ПРОГРАММИСТУ

Л.А.Мацяшек, Б.Л.Лионг

ПРАКТИЧЕСКАЯ программная инженерия

на основе учебного примера

Лаборатория ЗНАНИЙ

ПРОГРАММИСТУ

Л. А. Мацяшек, Б. Л. Лионг

ПРАКТИЧЕСКАЯ программная инженерия

на основе учебного примера

Перевод с английского
А. М. Епанешникова и В. А. Епанешникова

4-е издание (электронное)

Москва
Лаборатория знаний
2020

Leszek A. Maciaszek, Bruc Lee Liong
With contributions from Stephen Bills

PRACTICAL software engineering
A Case Study Approach

Harlow, England • London • New York • Boston • San Francisco • Toronto • Sydney • Singapore • Hong Kong
Tokyo • Seoul • Taipei • New Delhi • Cape Town • Madrid • Mexico City • Amsterdam • Munich • Paris • Milan

ПРАКТИЧЕСКАЯ
программная инженерия
на основе учебного примера

УДК 681.1.06
ББК 32.973-018.2
М36

Серия основана в 2005 г.

Мацяшек Л. А.

М36 Практическая программная инженерия на основе учебного примера / Л. А. Мацяшек, Б. Л. Лионг ; пер. с англ. — 4-е изд., электрон. — М. : Лаборатория знаний, 2020. — 959 с. — (Программисту). — Систем. требования: Adobe Reader XI ; экран 10". — Загл. с титул. экрана. — Текст : электронный.

ISBN 978-5-00101-783-7

Рассмотрены вопросы современных методов создания сложного программного обеспечения, использующего информацию, хранимую в базе данных. Подчеркнуты особенности создания такого программного обеспечения коллективом разработчиков: итеративный характер разработки, использование стандартных средств создания программ (стандартные компоненты, паттерны, Bean-компоненты и т. д.). Большое внимание уделено разработке структуры программного обеспечения, позволяющей наиболее просто организовать все стадии его жизненного цикла. Весь материал проиллюстрирован на одном достаточно сложном примере.

Для разработчиков сложного программного обеспечения, а также для студентов вузов, специализирующихся в вопросах создания современного ПО.

УДК 681.1.06
ББК 32.973-018.2

Деривативное издание на основе печатного аналога: Практическая программная инженерия на основе учебного примера / Л. А. Мацяшек, Б. Л. Лионг ; пер. с англ. — М. : БИНОМ. Лаборатория знаний, 2009. — 956 с. : ил. — (Программисту). — ISBN 978-5-94774-488-0.

ISBN 978-5-00101-783-7

ДЕВИЗ

Делайте все настолько просто, насколько возможно, но не проще.

Альберт Эйнштейн

ПОСВЯЩЕНИЯ

Диане, Доминике и Томашу

Лешек А. Мацяшек

Моим родителям, Эдисону и Тине

Брюс Ли Лионг

Оглавление

Экскурс в структуру книги

Глава 16

Проектирование и программирование пользовательского интерфейса

Основным «шагом» итерации 2 после итерации 1 учебного примера EM является замена текстового консольного интерфейса графическим и основанным на Web-технологии интерфейсом. Итерация 1 сосредоточена на введении системы надежного структурного проектирования. Это проектирование определяет внутреннюю структуру и внутренние операции системы. Однако в ПО ценятся **вид и поведение**, а не его внутренняя организация. Итерация 2 концентрируется на придании подсистеме EM надлежащего впечатления и ощущения от ее использования.

Как видно из вводных комментариев к главе 9, структурное проектирование имеет аналогии в строительной индустрии. Дом не может быть построен, если его сначала не спроектировал архитектор. Дом не может быть продан, если покупателю не понравится его вид и комфортность. Усилия по проектированию и программированию **пользовательского интерфейса** (user interface — UI) нацелены на создание привлекательного и полезного впечатления и ощущения от использования системы путем определения объектов пользовательского интерфейса и действий, которые позволяют пользователю реализовывать функции системы.

Промышленные приложения неизменно являются **клиент/серверными** (client/server — C/S) решениями. Часто они являются **многоярусными** решениями. Ярусы определяют ряд уровней серверов, типа Web-сервера, сервера приложения и сервера БД. Клиентом может быть рабочий стол, портативный компьютер, карманный компьютер, мобильный телефон, терминал ввода/вывода и т. д. Приложение может поддерживать многих клиентов.

Один из способов классификации клиентов состоит в разделении их на программируемых клиентов и клиентов-браузеров [94]. **Программируемый клиент** предполагает, что программа находится и выполняется на клиенте и имеет доступ к ресурсам, хранящимся на клиенте. Также важно, что программируемый клиент может загружать данные с сервера, отключаться от источника данных, если это потребуется, локально кэшировать данные, обрабатывать их и передавать их своему пользовательскому интерфейсу. Программируемый клиент иногда называется **толстым клиентом** или **богатым клиентом**.

Клиент-браузер нуждается в сервере, чтобы загрузить требуемые данные и получить инструкции для представления данных в пользовательском интерфейсе на основе Web-технологии. Кроме простых проверок пользова-

Начало главы задает место для детального обсуждения

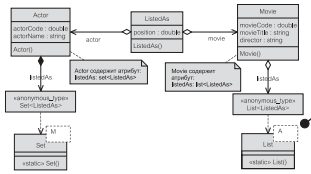


Рис. 12.4. Параметризованные типы C++

Например, `Actor` содержит переменную `listedAs`. Тип этой переменной задан как `Set<ListedAs>`. В действительности эта устанавливает связь агрегирования/ассоциации от `Actor` к коллекции ссылок (`Set`), указывающих на объекты класса `ListedAs`.

Листинг 12.4 показывает синтаксис C++, соответствующий модели на рис. 12.5. Определения классов для параметризованных типов задают параметры классов, на которых они определены. Точно также реализации всех методов, типа конструкторов `Set()` и `List()`, указывают на параметр класса. В этом случае параметризованный тип используется, чтобы инициализировать объекты коллекции, типа `Set<ListedAs> listedAs`.

Сравнение рисунков 12.2, 12.4 и связанных с ними листингов кода показывает, что использование параметризованных типов C++ лучше использования Java-интерфейса `Collection` для реализации ассоциаций с множественностью «многие». Создание новых типов, основанных на других типах, делает возможным определить переменные, имеющие тип классов коллекций, которые могут содержать только объекты определенного класса.

Листинг 12.4. Программирование параметризированных типов на языке C++

Программирование параметризованных типов на языке C++

```
template <class N> class Set
{
   public:
      static Set();
      ...
};
```

Многочисленные **снабженные комментариями образы экрана** для отображения ключевых моментов

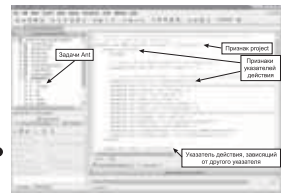


Рис. 3.33. Sun ONE Studio и Apache Ant: формирование системы
Источник: Sun Microsystems, Inc

Каждая задача Ant начинается с тега, называемого «target» (указатель действия) типа `«init»` (инициализировать) или `«compile»` (компилировать). Указатель действия `«init»` используется как основа для других указателей, задавая начальные свойства проекта. Указатель действия может ссылаться на другие указатели, чтобы выполнить свою работу. На рис. 3.33 указатель действия `«compile»` зависит от `«init»`. Эта зависимость заставляет Ant выполнить указатель `«init»` перед указателем `«compile»`.

3.4.4. Поддержка реинжиниринга

Самой объемной задачей программной инженерии является автоматическое преобразование проектов, представленных в существующих унаследованных системах, чтобы повторно реализовать их в новой форме. **Унаследованная система** — все еще используемое программное решение, обычно большая информационная бизнес-система, которая была реализована в давние времена в традиционной технологии и приобрела широкие дополнительные функциональные возможности в процессе **эволюционного сопровождения**.

Унаследованная система — история организации, которая остается с более раннего времени, но не может быть выброшена или заменена. Сама организация становится зависимой от этой системы, обеспечивающей ежедневные бизнес-операции. Эволюционное сопровождение системы, созданной много лет назад, нарушает ее первоначальный структурный проект. Добавление новых изменений необходимо из-за повышения требований пользовате-

Широкое использование **UML-диаграмм**

Листинг 18.28. Методы извлечения сообщений в CMsgSeeker

Методы `retrieveMessagesForCurrentEmployee()`, `retrieveMessagesFor-Department()` и `retrieveAllMessages()` в `CMsgSeeker`

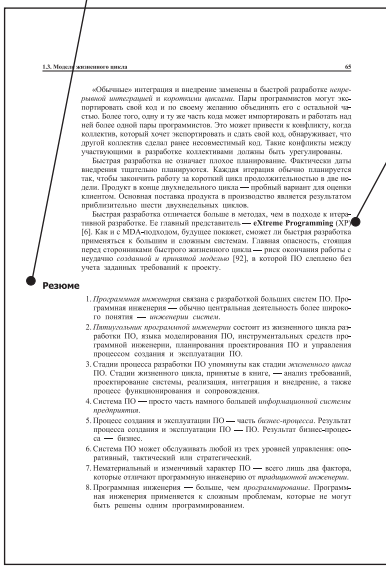
```
48:     public int retrieveMessagesForCurrentEmployee(
49:           Collection msgs,
50:           int numMsgs,
51:           Collection msgIDs) {
53:           int remainder = moderator.retrieveUnsentMessages(
                    emp, msgs, numMsgs);
56:           if (msgIDs == null || msgIDs.isEmpty())
57:              return remainder;
60:           Iterator it = msgs.iterator();
61:           while (it.hasNext()) {
62:              IAOutMessage msg = (IAOutMessage) it.next();
63:              Integer id = new Integer(msg.getMessageID());
64:              if (!msgIDs.contains(id)) {
65:                 it.remove();
66:                 remainder++;
67:              }
69:           }
70:           return remainder;
77:     }
91:     public int retrieveMessagesForDepartment(String
                                          departmentCode,
92:           Collection msgs,
93:           int numMsgs,
94:           Collection msgIDs) {
96:           int remainder = moderator.retrieveUnsentMessages(
                    departmentCode, msgs, numMsgs);
                 ...(то же самое, что и строки 56–77)
134:    public int retrieveAllMessages(
135:       Collection msgs,
136:       int numMsgs,
137:       Collection msgIDs) {
139:       int remainder = moderator.retrieveUnsentMessages(msgs,
                    numMsgs);
                 ...( то же самое, что и строки 56–77)
```

Фактический код показывает читателям, как концепции реализуются на практике

18.5. Пакет Entity

Наиболее заметные изменения в пакете `entity` (сущность) — это добавление интерфейса `IEObjectID` (интерфейс «идентификатор объекта» пакета `presentation`) и класса `EIdentityMap` (класс «коллекция идентичности

Резюме в конце каждой главы закрепляет знания



1.3. Модели жизненного цикла 65

«Обычная» интеграция и внедрение заменены в быстрой разработке *непрерывной интеграцией* и *короткими циклам*. Пары программистов могут экспортировать свой код и по своему желанию объединять его с остальной частью. Более того, одну и ту же часть кода может импортировать и работать над ней более одной пары программистов. Это может привести к конфликту, когда коллектив, который хочет экспортировать и сдать свой код, обнаруживает, что другой коллектив сделал ранее несовместимый код. Такие конфликты между участвующими в разработке коллективами должны быть урегулированы.

Быстрая разработка не означает плохое планирование. Фактически даты внедрения тщательно планируются. Каждая итерация обычно планируется так, чтобы закончить работу за короткий цикл продолжительностью в две недели. Продукт в конце двухнедельного цикла — пробный вариант для оценки клиентом. Основная поставка продукта в производство является результатом приблизительно шести двухнедельных циклов.

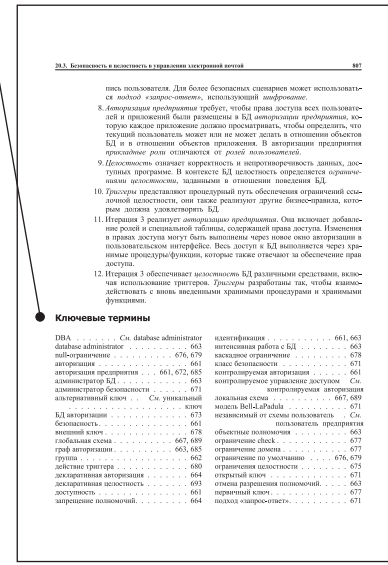
Быстрая разработка отличается больше в методах, чем в подходе к итеративной разработке. Ее главный представитель — **eXtreme Programming** (XP) [6]. Как и с MDA-подходом, будущее покажет, сможет ли быстрая разработка применяться к большим и сложным системам. Главная опасность, стоящая перед сторониками быстрого жизненного цикла — риск окончания работы с неудачно *созданной и принятой моделью* [92], в которой ПО слепленo без учета заданных требований к проекту.

Резюме

1. *Программная инженерия* связано с разработкой больших систем ПО. Программная инженерия — обычно центральная деятельность более широкого понятия — *инженерии систем*.
2. *Пятиуровневая программной инженерии* состоит из жизненного цикла разработки ПО, языка моделирования ПО, инструментальных средств программной инженерии, планирования проектирования ПО и управления процессом создания и эксплуатации ПО.
3. Стадии процесса разработки ПО уложились как стадии *экспертного цикла* ПО. Стадии жизненного цикла, принятые в книге, — анализ требований, проектирование системы, реализация, интеграция и внедрение, а также процесс функционирования и сопровождения.
4. Система ПО — просто часть намного больший *информационной системы предприятия*.
5. Процесс создания и эксплуатации ПО — часть *бизнес-процесса*. Результат процесса создания и эксплуатации ПО — ПО. Результат бизнес-процесса — бизнес.
6. Система ПО может обслуживать любой из трех уровней управления: оперативный, тактический или стратегический.
7. Нематериальный и изменчивый характер ПО — всего лишь два фактора, которые отличают программную инженерию *от традиционной инженерии*.
8. Программная инженерия — больше, чем *программирование*. Программная инженерия применяется к сложным проблемам, которые не могут быть решены одним программированием.

Ключевые термины выделены в тексте

Чтобы помочь найти **ключевые термины** и их **определения в тексте**, они отмечены в конце каждой главы

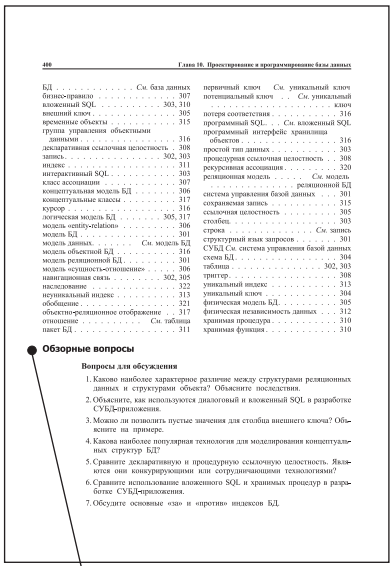


20.3. Безопасность и целостность в управлении электронной почтой 807

пись пользователя. Для более безопасных сценариев может использоваться *подход «запрос-ответ»*, использующий *шифрование*.

8. *Авторизация предприятия* требует, чтобы права доступа всех пользователей и приложений были размещены в БД *авторизации предприятия*, которую каждое приложение должно просматривать, чтобы определить, что текущий пользователь может или не может делать в отношении объектов БД и в отношении объектов приложения. В авторизации предприятия *присвоение роли* отличается от *ролей пользователей*.
9. *Целостность* означает корректность и непротиворечивость данных, доступных программе. В контексте БД целостность определяется *ограничениями целостности*, заданными в отношении поведения БД.
10. *Триггеры* представляют процедурный путь обеспечения ограничений ссылочной целостности, они также реализуют другие бизнес-правила, которым должна удовлетворять БД.
11. Итерация 3 реализует *авторизацию предприятия*. Она включает добавление ролей и специальной таблицы, содержащей права доступа. Изменения в правах доступа могут быть выполнены через новое окно авторизации в пользовательском интерфейсе. Весь доступ к БД выполняется через хранимые процедуры/функции, которые также отвечают за обеспечение прав доступа.
12. Итерация 3 обеспечивает *целостность* БД различными средствами, включая использование триггеров. *Триггеры* разработаны так, чтобы взаимодействовать с вновь введенными хранимыми процедурами и хранимыми функциями.

Ключевые термины

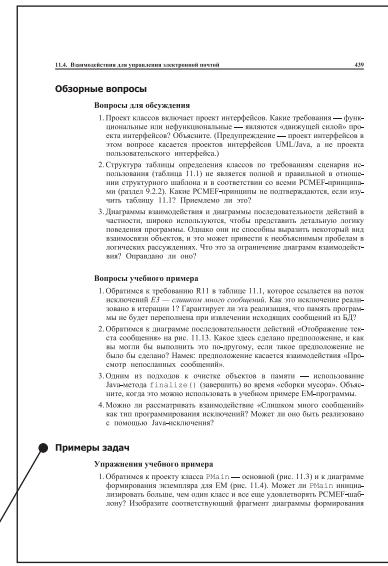


400 Глава 10. Проектирование и программирование базы данных

Обзорные вопросы

Вопросы для обсуждения

1. Каково наиболее характерное различие между структурами реляционных данных и структурами объекта? Объясните последствия.
2. Объясните, как используются диалоговый и вложенный SQL в разработке СУБД-приложения.
3. Можно ли позволить пустые значения для столбца внешнего ключа? Объясните на примере.
4. Какова наиболее популярная технология для моделирования концептуальных структур БД?
5. Сравните декларативную и процедурную ссылочную целостность. Являются они конкурирующими или сотрудничающими технологиями?
6. Сравните использование вложенного SQL и хранимых процедур в разработке СУБД-приложения.
7. Обсудите основные «за» и «против» индексов БД.



11.4. Взаимодействия для управления электронной почтой 439

Обзорные вопросы

Вопросы для обсуждения

1. Проект классов включает проект интерфейсов. Какие требования — функциональные или нефункциональные — являются «движущей силой» проекта интерфейсов? Объясните. (Предупреждение — проект интерфейсов в этом вопросе касается проектов интерфейсов UML/Java, а не проекта пользовательского интерфейса.)
2. Структура таблицы определения классов по требованиям сценария использования (таблица 11.1) не является полной и правильной в отношении структурного шаблона и в соответствии со всеми PCMEF-принципами (раздел 9.2.2). Какие PCMEF-принципы не подтверждаются, если изучить таблицу 11.1? Приемлемо ли это?
3. Диаграммы взаимодействия и диаграммы последовательности действий в частности, широко используются, чтобы представить детальную логику поведения программы. Однако они не способны выразить некоторый вид взаимосвязи объектов, и это может привести к необъяснимым пробелам в логических рассуждениях. Что это за ограничение диаграмм взаимодействия? Оправдано ли оно?

Вопросы учебного примера

1. Обратимся к требованию R11 в таблице 11.1, которое ссылается на поток исключений *E3 — слишком много сообщений*. Как это исключение реализовано в итерации 1? Гарантирует ли эта реализация, что память программы не будет переполнена при извлечении исходящих сообщений из БД?
2. Обратимся к диаграмме последовательности действий «Отображение текста сообщения» на рис. 11.13. Какое здесь сделано предположение, и как вы могли бы выполнить это по-другому, если такое предположение не было бы сделано? Намек: предположение касается взаимодействия «Просмотр непосланных сообщений».
3. Одним из подходов к очистке объектов в памяти — использование Java-метода `finalize()` (завершить) во время «уборки мусора». Объясните, когда это можно использовать в учебном примере EM-программы.
4. Можно ли рассматривать взаимодействие «Слишком много сообщений» как тип программирования исключений? Может ли оно быть реализовано с помощью Java-исключения?

Примеры задач

Упражнения учебного примера

1. Обратимся к проекту класса `PMain` — основной (рис. 11.3) и к диаграмме формирования экземпляра для EM (рис. 11.4). Может ли `PMain` инициализировать больше, чем один класс и все еще удовлетворить PCMEF-шаблону? Изобразите соответствующий фрагмент диаграммы формирования

Общие **обзорные вопросы** в конце главы
Обзорные вопросы в конце главы характерны для **текущего учебного примера**

Примеры задач в конце главы позволяют читателям лучше понять учебный пример
Дополнительные **небольшие примеры** в конце главы для проверки более глубокого понимания

Введение

История создания книги

История создания этой книги представляет собой итеративный и пошаговый процесс. Конечно, она соответствует всем четырем основным фазам популярной модели жизненного цикла: замысел, разработка, реализация и использование. Книга явилась плодотворным результатом работы двух авторов с учетом требований пользователей, вытекающих из их практики, с широким использованием непрерывной интеграции и рефакторинга, но, к сожалению, с достаточно длинным циклом разработки[1].

Замысел создания книги датируется публикацией в 1990 г. книги Мацяшека *Database Design and Implementation* (Prentice Hall) — *Проектирование и создание баз данных*. Многие читатели настаивали на продолжении этой книги с использованием законченных учебных примеров, маленьких и больших упражнений и с пошаговым (то есть итеративным и возрастающим) увеличением технической трудности и сложности содержания. Деловое предложение насчет книги было сделано, и проект вступил в стадию разработки — представление о книге было уточнено, риски устранены, требования и границы определены. Однако чтобы завершить целевую платформу, вместо работы над книгой пришлось заниматься… большим объемом учебной работы и консультаций в промышленности.

Десятью годами позже после работы над бесчисленными промышленными проектами количество собранного практического материала буквально кричало относительно необходимости публикации для широкой аудитории. Но аудитория изменилась — промышленность вступила в век Интернета. Требовался новый учебник, чтобы дать необходимые знания для разработки современного программного обеспечения (ПО). Таким учебником стал учебник Мацяшека *Requirements Analysis and System Design: Developing Information Systems with UML* (Addison-Wesley, 2001) — Анализ требований и проектирование систем: разработка информационных систем с использованием UML, — вышедший вторым изданием одновременно с этой книгой. Вскоре после этого книга, которую вы держите в руках, вступила в стадию создания.

[1] Под рефакторингом в программировании понимается улучшение структуры программного кода с целью более легкого его понимания, но без изменения внешнего поведения этого кода. В данном конкретном случае этот термин относится не к программному коду, а к самой книге, чем авторы подчеркивают некоторую схожесть процесса написания программного кода и данной книги. — *Прим. перев.*

Стадия *реализации* была ухабистой. Первоначально книга предполагалась как дополнение к учебнику Мацяшека 2001 г. или любой подобной книге. Позже акцент сместился в пользу самостоятельного учебника, который использует подход итеративного учебного примера к выработке навыков практической разработки ПО. Книга сконцентрирована на проектировании ПО, программировании и администрировании. Она посвящена вопросам современной практики разработки, методам, техническим приемам и инструментальным средствам.

Стадия *использования* этой книги находится в ваших, читатель, руках. Бета-тестирование (предварительное тестирование) этой книги проводилось и в классных комнатах, и в процессе проектирования ПО. Дальнейшее расширение использования — в воле читателя. Пожалуйста, присылайте пожелания по изменению, сведения о выявленных дефектах и предложения по совершенствованию книги в группу разработчиков.

Схема и организация книги

Отличительный характер этой книги проистекает из двух стремлений авторов. Во-первых, это книга о способе разработки ПО, проверенном *на практике*. Во-вторых, она о разработке ПО применительно к *промышленным приложениям*. Следующее описание того, что промышленные приложения включают и исключают, полностью применимо к этой книге: «Промышленные приложения включают платежную ведомость, регистрацию пациентов, отслеживание отгрузки, стоимостный анализ, оценку кредита, страхование, цепь поставок, бухгалтерский учет, обслуживание клиентов и международную биржевую торговлю. Промышленные приложения не включают вопросы, связанные с инжекцией автомобильного топлива, текстовыми процессорами, устройствами управления подъемниками, устройствами управления химическими заводами, телефонными коммутаторами, операционными системами, компиляторами и играми» [31].

Содержимое книги вращается вокруг одного основного **учебного примера**, двух **небольших проектов** с относящимися к ним упражнениями, большого количества сопутствующих **примеров**, **учебных средств** для рассмотрения основных концепций моделирования и программирования и **примеров задач** в конце каждой главы, которые содержат главным образом **упражнения, связанные с учебным примером**. Организация, на которую даются ссылки в учебном примере и в небольших проектах, а также в упражнениях, является компанией, специализирующейся в *оценке расходов на рекламу*. В книге эта организация по своей основной деятельности называется AEM (advertising expenditure measurement — оценка расходов на рекламу). Фактически же это ACNielsen's Nielsen Media Research в Сиднее, Австралия. Учебный пример называется *Email Management* (EM) — управление электронной почтой. EM является подсистемой системы *Contact Management* (CM) — управление деловыми партнерами AEM.

Как показано на диаграмме Венна, AEM занимается бизнесом, CM является одной из предметных областей бизнеса, а EM является учебным примером. Упражнения и небольшие проекты взяты из срезов областей AEM-биз-

неса, включая СМ. Некоторые примеры не связаны с АЕМ. Учебный пример обогащен дополнительными примерами и упражнениями. Учебные средства используются для того, чтобы быстро изучить вводимые темы, связанные с UML-моделированием, Java-программированием, реляционными базами данных (БД), конструкцией GUI (graphical user interface — графический интерфейс пользователя) и работой с бизнес-компонентами.

В книге делается попытка дать широкие знания по разработке ПО и представить исходную информацию до рассмотрения решений учебного примера. Другой же акцент книги — показать, как использовать эти знания при проектировании ПО. Для обеспечения данных требований книга итеративно развивает модели проектирования и реализации. Учебный пример, небольшие проекты, задачи и упражнения для решения задач тщательно отобраны, чтобы подчеркнуть многочисленные аспекты разработки ПО, как требуется в соответствии с уникальным характером различных приложений и целевых решений ПО.

Книга состоит из четырех частей. В части 1 (**Проектирование программного обеспечения**) обсуждаются жизненный цикл разработки ПО, языки моделирования, средства разработки, планирование проекта и управление процессом. Следующие три части (2-я, 3-я и 4-я) рассматривают учебный пример, небольшие проекты и примеры. Обсуждение в этих трех частях концентрируется на методах, технологиях, процессах и средствах разработки ПО.

Части 2, 3 и 4 соответствуют трем итерациям проекта (учебного примера). Каждая итерация начинается со спецификаций учебного примера, соответствующих начальной модели объекта. Базовая теория и практические знания, подкрепляющие каждую итерацию, объясняются в первую очередь демонстрацией проектирования и программных решений учебного примера. Любая информация, важная для решений учебного примера, но не обладающая существенным базовым значением, представлена внутри этого примера, либо как подраздел обсуждения учебного примера. Каждая итерация завершается полным решением и заканчивается вместе с главой, которая содержит исходный код с необходимыми комментариями, аннотациями и ссылками на объяснения в предшествующих главах.

Часть 2 (**От требований через структурное проектирование к готовому программному обеспечению**) начинается заданием бизнес-параметров для учебного примера ЕМ. Первые две главы этой части представляют модель бизнес-объектов для АЕМ и модель предметной области для СМ. Далее определены требования ЕМ и последовательно разрабатывается его первая итера-

ция. Краеугольный камень первой итерации — надежный структурный проект, поддающийся последовательным пошаговым улучшениям. «Подлежащим сдаче» итогом первой итерации является выпуск в свет ПО для пользователей (то есть читателей этой книги).

Часть 3 (**Рефакторинг программного обеспечения и разработка пользовательского интерфейса**) концентрируется на определении внешнего интерфейса системы и на отдельных частях приложения. В ней обсуждается проектирование графического пользовательского интерфейса (GUI — graphical user interface), включая Web-доступный внешний интерфейс. Переход от итерации 1 к итерации 2 обеспечивается структурным рефакторингом и разработкой дружественного пользовательского интерфейса.

Часть 4 (**Разработка данных и бизнес-компоненты**) перемещает внимание с внешнего интерфейса системы к ее сути и ее промежуточному логическому уровню. В этой части обсуждается хранение данных и манипуляция ими, реализация бизнес-правил, обработка транзакций и управление безопасностью. Она объясняет также, каким образом логика приложения может быть перемещена на сервер приложения в промежуточный логический уровень.

Хотя итерация 3 учебного примера является развитием итерации 2, части 3 и 4 книги можно изучать достаточно независимо. Читатель может сконцентрироваться на одной из этих частей и лишь вскользь просмотреть другие части. Например, проектировщика БД (программиста) может особо заинтересовать часть 3, в то время как GUI-проектировщика или Java-программиста может в первую очередь заинтересовать часть 4. Возможно, некоторые читатели сконцентрируют свое внимание на частях 1 и 2 книги и полностью определятся с частями 3 и 4 после экспериментирования и лучшей оценки знаний, содержащихся в первых частях. В продвинутых курсах, возможно, будет лучше начать использовать книгу с части 2.

Из общего количества 23 глав книги, 6 посвящены учебному примеру ЕМ (это главы 8, 13, 14, 18, 19 и 23). Образовательная цель этих шести глав — в понимании и анализе учебного примера. Это действительно метод обучения на примере. В противоположность этому первые пять глав (часть 1) объясняют основы разработки ПО и не обращаются к учебному примеру. Оставшиеся 12 глав имеют и теоретические части, и части, которые связывают теорию с учебным примером, небольшими проектами или другими примерами.

Отличительные особенности

Основная отличительная особенность книги выражена в ее подзаголовке: **«Подход с использованием учебного примера»**. Если вы считаете, как делают многие педагоги, что лучшая форма обучения — *обучать на примере*, то эта книга для вас. Если вы любите сомневаться и хотите *учиться на ошибках*, то эта книга дает вам много возможностей экспериментировать с вашими решениями и сравнивать их с ответами и объяснениями авторов. Если вдобавок ко всему вы хотели бы *настроить изучение на ваши текущие потребности и уровень ваших знаний*, тогда следует учесть, что каждая итерация имеет свой акцент, свою сложность моделирования и может потребовать отличный от других набор методов разработки и моделей.

Главная цель этой книги состоит в том, чтобы связать теорию с действительностью, уделяя особое внимание проектированию и реализации ПО (одновременно не пренебрегая анализом) и делая акценты на нетривиальных практических проблемах. В своей цели «пояснять на примерах» **эта** книга уникальна наличием следующих моментов:

1. *Цель — обучение*. Книга была написана для обучения. Учебный пример, отдельные примеры и упражнения не только взяты из реальной жизни, они сформированы, чтобы удовлетворить потребности обучения. Реальные решения являются частью сложного бизнеса в контексте создания ПО. Этот контекст, вероятно, может показаться читателю чрезвычайно обширным и неинтересным, так что он упрощен в максимально возможной степени. Представление GUI и проектов БД наряду с примерами программирования исключает несущественные зависимости, «информационный шум» и повторение задач.
2. *Аннотируемые решения*. В информационных системах не существуют «черное или белое», «истина или ложь», «ноль или один». Часто решение преследует конкретную специфическую цель и может показаться совершенно неправильным с точки зрения других применений. Поэтому ответы и решения тщательно аннотируются.
3. *Альтернативные решения*. Иногда отдельное решение, независимо от того, как оно аннотируется и объясняется, ненамного лучше, чем другие потенциальные решения. В этом случае часто даются и объясняются альтернативные решения.
4. *Списки ключевых терминов* в конце каждой главы, организованные в алфавитном порядке с указанием номеров страниц. Списки можно использовать для самостоятельного изучения основной терминологии, помещенной в каждую главу. Они могут также использоваться преподавателями для проверки знаний студентов по каждой главе.
5. *Обзорные вопросы* — для закрепления знаний читателей с помощью серьезных вопросов по каждой главе. Вопросы разделены, когда это удобно, на вопросы, касающиеся обсуждения проблемы, и вопросы по учебному примеру. Ответы на все обзорные вопросы преподаватели могут получить на Web-сайте книги.
6. *Упражнения, связанные с решением конкретных проблем*, имеющие целью стимулировать читателя исследовать проблемы перед попыткой их решения и попробовать получить расширенные или альтернативные решения учебного примера, небольших проектов или других примеров. Типовые решения всех обзорных вопросов доступны преподавателям на Web-сайте книги.
7. *Web-сайт* с полным набором сопутствующего материала, включая модели и код программ (главным образом код UML, Java и БД Oracle). Весь код программ, включая код, не представленный в тексте, доступен на Web-сайте книги.
8. *Акцент на основы*. Имеются определенные четкие основы (модели, структуры, стандарты, библиотеки и т. д.), позволяющие разрабатывать хорошие ПО и системы. В книге определяются и объясняются эти основы и даются источники информации.

9. *Сбалансированная смесь профессиональной глубины и образовательных возможностей*. В общем-то, писать ПО и писать учебники – несколько разные вещи. Хотелось бы надеяться, что данная книга противоречит этому тезису.
10. *Использование вместо профессионального образования и обучающих курсов*. Занятые профессионалы часто решают обычные задачи и могут быстро отстать в искусстве и даже в практике своей дисциплины. Найти время и средства для получения дорогого профессионального образования и посещения обучающих курсов с использованием учебных примеров, подобных тем, которые рассмотрены в этой книге, может оказаться затруднительным. Есть надежда, что эта книга может предоставить профессионалам возможность ознакомиться с самыми последними разработками как альтернативу этим вариантам или даже в процессе выполнения обычных рабочих обязанностей.

Для кого предназначена эта книга

Эта книга нацелена на широкую аудиторию студентов и IT-профессионалов (профессионалов в области информационных технологий). Идеальный читатель — это студент курса разработки ПО или конструктор ПО (а также руководитель проекта/менеджер). Книга написана так, чтобы она могла быть абсолютно понятна студентам и профессионалам, которые обладают элементарными знаниями по информационным системам и основными навыками программирования (желательно на объектно-ориентированном языке и с некоторыми навыками использования БД). Для большинства читателей это соответствует первому году университетского курса по компьютерным наукам или информатике (информационные системы, информационная технология).

Для *студентов* основное использование этого учебника — в курсах разработки ПО с компонентами его проектирования. Книга может также использоваться как учебник в курсах разработки информационных систем, проектирования ПО, или анализа и проектирования систем, которые преподаются на более старших курсах обучения, или как пособие для более подготовленных студентов. Кроме того, книга может быть рекомендована в курсах по объектной технологии, объектному программированию, системам на основе Web-технологий, проектированию и программированию БД и подобных курсах.

Практики, особенно заинтересованные в книге, — это в первую очередь системные проектировщики, программисты, проектировщики ПО, бизнес- и системные аналитики, руководители проектов и менеджеры, Web-разработчики и разработчики по наполнению Web-ресурсов, рецензенты, испытатели, менеджеры по обеспечению качества и промышленные испытатели. Книга может использоваться для профессионального образования, обучающих курсов и семинаров. Она может также быть взята в качестве источника информации для команд проектировщиков. Для практиков, уже использующих UML, Java и реляционные БД в проектах ПО, эта книга может служить как средство проверки правильности своей практической разработки ПО и как источник идей и направлений проектирования.

Сопутствующие материалы

Полный пакет сопутствующих материалов находится на Web-сайте компаньонов. Большая часть содержания Web-сайта свободно доступна всем читателям, но некоторый материал защищен паролем и предназначен для преподавателей, которые будут использовать книгу в обучении. Домашняя страница этой книги находится на сайтах:

http://www.comp.mq.edu.au/books/pse
http://www.booksites.net/maciaszek

Web-сайт этой книги содержит два вида ресурсов: для всех читателей и для преподавателей, использующих эту книгу для обучения. Ресурсы преподавателей защищены паролем.

Ресурсы преподавателей на Web-сайте включают (но только этим не ограничиваются):

1. *Руководство для преподавателя*, содержащее:
   а) *вопросы и ответы* ко всем обзорным вопросам в конце каждой главы и упражнения по решению проблем;
   б) дополнительные *проекты и решения*, не содержащиеся в учебнике и доступные на Web-сайте, чтобы помочь преподавателям выбрать для студентов задания и проекты.
2. *Слайды для лекций* в PowerPoint и в модифицируемом pdf-формате (Acrobat).
3. *UML-модели и исходный код Java/БД* для:
   а) упражнений и небольших проектов;
   б) проектов, находящихся на Web-сайте книги;
   в) альтернативных подходов к проектированию/программированию учебного примера книги.

Ресурсы для всех читателей включают (но не только это):

1. *Слайды лекций* — в pdf-формате (Acrobat) только для чтения.
2. *Опечатки и дополнения*.
3. *UML-модели и исходный код Java/БД* для учебного примера книги и законченных примеров с инструкциями о том, как компилировать и запускать код.

Ваши комментарии, исправления, предложения по усовершенствованию, дополнения и т. д. очень ценны. Пожалуйста, направляйте любую корреспонденцию по адресу:

Leszek A. Maciaszek
Department of Computing
Macquarie University
Sydney
NSW 2109, Australia
email: leszek@ics.mq.edu.au
web: http://www.comp.mq.edu.au/~leszek
телефон: +61 2 9850-9519
факс: +61 2 9850-9551
курьерская почта: North Ryde, Herring Road, Bid. E6A, Room 319

Благодарности

Благодарности авторов

Эта книга потребовала для написания значительного времени, но это время пренебрежимо мало по сравнению со временем, потребовавшимся, чтобы получить знания и навыки, необходимые для ее написания. Наша особая благодарность — нашим друзьям и коллегам в **ACNielsen, Сидней, Австралия**, которые обеспечили начальную «испытательную среду» для многих идей, предложенных читателям в этой книге. Наша благодарность прежде всего Стивену Биллсу (который является участником создания этой книги) и его команде разработчиков, включая Пола Антоуна, Бруно Бейру, Сью Дэйз, Стивена Гротта, Джефа Хонга, Йиджуна Ли, Кевина Мати, Денизу Маккрей, Шанталь О'Коннел, Джеймса Риса, Джована Споа, Эрика Цурхера.

Написание книги — немалый проект. Как сказано в главе 4 и еще в нескольких местах книги, успешное создание проекта требует большой работы и материальных ресурсов для решения его задач. Рабочие ресурсы состоят из людей и оборудования, включая аппаратное и программное обеспечение. Материальные ресурсы — расходные материалы и товарно-материальные ценности. Авторы этой книги распределили себя по задачам, но проект потерпел бы серьезную неудачу без всей сопутствующей работы и материальных ресурсов. Ресурсы обеспечивались **Университетом Маккуэри, Сидней, Австралия** и **Университетом Экономики, Вроцлав, Польша**. Мы обязаны нашим друзьям, коллегам и студентам в этих двух университетах за их советы, поддержку и помощь, за все средства, которые они предоставили этому проекту.

Говоря о ресурсах, следует отметить, что эта книга не могла бы быть написана без интенсивного и всестороннего использования инструментального ПО и сред программирования. ПО, необходимое для книги, было получено различными способами — от приобретения демонстрационных копий до использования свободно распространяемых источников ПО. Люди, связанные с этим ПО, полученным таким образом, неизвестны нам, так что мы не можем передать благодарность конкретно им. Наши благодарности, однако, вместо них адресуются продавцам ПО, которые отвечали на наши запросы, как получить ПО бесплатно для использования в учебных целях нами и нашими студентами. Мы особенно обязаны фирмам Oracle Corporation (Oracle и JDeveloper), Rational Software Corporation, а в настоящее время IBM (Rational Suite), Sybase (PowerDesigner) и yWorks (yDoc).

Особо мы благодарны **Кейту Мансфилду** из Pearson Education, редактору этой книги, а также и редактору книги Мацяшека *Requirements Analysis and Systems Design.* Спасибо, Кейт, за профессиональное видение обеих книг и за Вашу упорную работу от начала до производства и передачи в продажу. Книжное производство на самом деле является усилием целой команды. Рискуя упустить ряд фамилий, мы хотели бы выразить особую благодарность **Аните Аткинсон** (главный редактор), **Элен Макфадьен** (корректор), **Рут Фристон Кинг** (выпускающий редактор) и **Оуэн Найт** (помощник редактора). Спасибо вам за все исправления, улучшения, понимание, советы и тесное сотрудничество.

Благодарности издателя

Мы благодарны за разрешение воспроизвести авторский материал:

Таблица 1.1 основана на информации из *Fundamentals of Software Engineering,* Prentice Hall — Основы разработки ПО (Pearson Education, Inc.), [34]; рисунок 1.10 преобразован из рисунка *Rational Unified Process* (рациональный унифицированный процесс), перепечатанного по разрешению © Copyright 2003 by International Business Machines Corporation. All Rights Reserved с сайта http:// www.rational.com/products/rup/; рисунки 3.1, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 4.10, 4.11, 4.13, 4.14, 4.16, 4.17, 4.19, 4.20, 4.21, 4.22, 4.23, 4.25, 4.26, 4.28 и 4.29 — с изображений экрана Microsoft® Office Project (2003), перепечатанные по разрешению Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation; рисунок 3.2 — с изображения экрана Manage-Pro™ 6.1 из сайта www.managepro.net, March 2004, Performance Solutions Technology, LLC, перепечатанный с любезного согласия Performance Solutions Technology, LLC; рисунок 3.3 — с изображения экрана eRoom из сайта www.eroom.net/eRoomNet/, July 2003, Documentum, Inc., перепечатанный с любезного разрешения Documentum, Inc.; рисунок 3.4 — с изображения экрана eProject Enterprise, July 2003, перепечатанный с любезного разрешения eProject, Inc.; рисунки 3.5 и 3.6 — с изображений экрана *Small Worlds*, перепечатанные с www.thesmallworlds.com/ по разрешению © Copyright 2003 by International Business Machines Corporation. All Rights Reserved; рисунок 3.8 — с изображения экрана @Risk из сайта www.palisade-europe.com, перепечатанный с любезного разрешения Palisade Corporation; рисунки 3.9, 3.10, 3.12, 3.14, 3.16, 3.30, 3.31, 5.14, 5.15 — с изображений экрана *IBM Rational Suite*, перепечатанные по разрешению *Rational Suite Tutorial, Version 2002.05.00,* © Copyright 2002 by International Business Machines Corporation. All Rights Reserved; рисунок 3.9 — с изображения экрана Microsoft® Word, перепечатанный по разрешению Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation; рисунок 3.11 — с изображения экрана DOORS® из сайта www.telelogic.com, перепечатанный по любезному разрешению Telelogic UK Ltd., Copyright © 2004 Telelogic AB; рисунок 3.13 — с изображения экрана Enterprise Architect из сайта www.sparxsystems.com.au, August 2003, Sparx Systems Pty Ltd., перепечатанный по любезному разрешению Sparx Systems Pty Ltd.; рисунок 3.15 — с изображения экрана Gentleware's Poseidon Sales Application model, August 2003, перепечатанный

с любезного разрешения Gentleware AG; рисунки 3.16, 3.34 и 3.35 — с изображений экрана No Magic's MagicDraw™, перепечатанный по разрешению из сайта www. magicdraw.com, July 2003, © Copyright No Magic, Inc. 1998–2004, All Rights Reserved; рисунок 3.17 — с изображения экрана Sybase® PowerDesigner® version 9.5 из сайта www.sybase.com, July 2003, © Copyright 2002, Sybase, Inc., All Rights Reserved; рисунок 3.25 — с изображения экрана *Borland® Together® ControlCenter® example* (2003), www.borland.com,, перепечатанный по разрешению Borland Software Corporation; рисунок 3.27 — с изображения экрана Oracle JDeveloper из сайта www.otn.oracle.com, перепечатанный по разрешению Oracle Corporation; рисунок 3.29 — с изображения экрана Perforce из сайта www.perforce.com, перепечатанный с любезного согласия Perforce Software, Inc.; рисунок 3.32 — с изображения экрана Microsoft® Visual SourceSafe из сайта http://msdn.microsoft.com/ssafe/default.asp, (2003), перепечатанный по разрешению Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation; таблицы 4.2 и 4.3 основаны на информации из *Software Cost Estimation with COCOMO II* — оценка стоимости программного обеспечения с помощью COCOMO II, Prentice Hall (Pearson Education, Inc.) [11]; рисунки 5.11 и 20.4 — из *Requirements Analysis and Systems Design with UML* — анализ требований и проектирование систем с помощью UML, Addison Wesley, (Pearson Education), (Maciaszek, L.A., 2001); рисунок 12.11 — с изображения экрана Microsoft® DOS, перепечатанный по разрешению Microsoft Corporation, Copyright © 1990–2003 Microsoft Corporation; рисунок 17.18 преобразован из *Mastering Jakarta Struts* by James Goodwill. Copyright © 2002 by Ryan Publishing Group. Перепечатан здесь по разрешению Wiley Publishing, Inc. All rights reserved; рисунки 22.3 и 22.4 — с изображений экрана Oracle Business Component Browser из сайта www.otn.oracle.com, перепечатанный по разрешению Oracle Corporation.

В некоторых случаях было невозможно проследить владельцев авторского права, и мы будем признательны за любую информацию, которая позволила бы нам сделать это.

Часть 1

Проектирование программного обеспечения

Начинать с определения основной терминологии — хороший стиль в области передачи знаний (типа написания книги или производства ПО). Термины нужно только определять и разъяснять, но по возможности не изобретать. Действительно, большинство терминов, с которыми мы работаем, было изобретено до нас.

Эта книга о ПО и системах. Она посвящена программной инженерии и разработке систем ПО. В ней рассматриваются большие проекты ПО и информационные системы предприятий. В книге используются следующие определения, данные видными экспертами в рассматриваемой области.

«*Программная инженерия* — область информатики, имеющая дело с созданием систем ПО, которые являются настолько большими или настолько сложными, что создаются коллективом или коллективами инженеров» [34]. В этом определении имеется ряд важных моментов.

«Создание систем ПО» — другими словами можно сказать «*разработка систем*». На самом деле программная инженерия — это несколько большее, чем собственно разработка ПО. Она включает также и *сопровождение ПО*. Подобно любому техническому изделию, типа моста или дома, ПО должно поддерживаться. В отличие от случая типичного технического изделия, сопровождение ПО включает его развитие (добавление новых модулей) и глубокие изменения в самих основах продукта. В этом смысле разработка ПО дополняется его обслуживанием.

«*Система* — целенаправленное собрание находящихся во взаимосвязи компонентов, которые работают вместе для достижения некоторой цели» [96]. Разработка системы (и обслуживание) связана с процессами, методами и инструментальными средствами производства ПО. Поскольку ПО является моделью действительности, его разработка связана с *моделированием* продуктов ПО.

Система — это больше, чем просто ПО. «*Инженерия систем* связана со всеми аспектами разработки и развития сложных систем, в которых ПО играет главную роль» [96]. «Инженерия систем концентрируется на разнообразии элементов, анализе, проектировании и организации этих элементов в систему, которая может быть изделием, сервисом или технологией преобразования информации или управления» [81]. А так как ПО играет главную роль в большинстве современных систем, эта книга также и об инженерии систем.

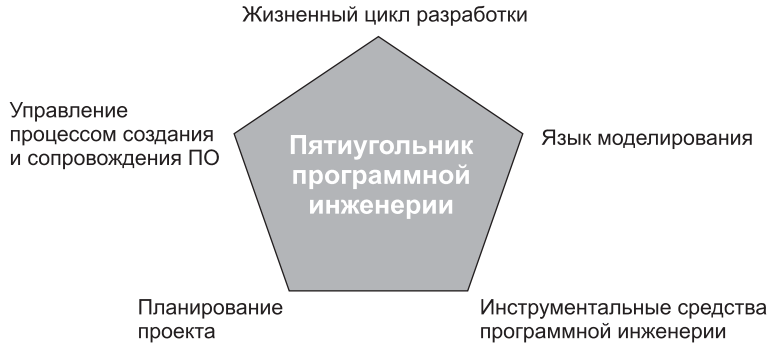
Большинство проектов ПО выполняется в связи с организационными потребностями заняться задачами идентификации в бизнес-процессе или улучшением этого процесса из-за конкуренции. *Проектирование ПО* — запланированная операция, предназначенная для создания программного продукта или сервиса и выполняющаяся в течение определенного времени. В этой книге особое внимание уделяется проектам ПО, предназначенным для *информационных систем предприятий*. Это большие и сложные системы. Как отмечено Фаулером [31], «Промышленные приложения часто содержат сложные данные, и большинство их работает на основе бизнес-правил, которым не хватает логических проверок».

Standish Group [98] исследует причины *отказов ПО*. Оригинальный *Отчет о хаосе* (Chaos Report), выполненный Standish Group в 1994 г., сообщил, что только 16.2 процентов проектов ПО были закончены вовремя и

в пределах выделенных средств. Насчет более крупных и сложных систем сведения оказались даже еще хуже: только 9 процентов таких проектов были закончены вовремя и без перерасхода средств. *Отчет о хаосе* за 2003 г. показал улучшение в этом деле — 34 процента проектов были закончены вовремя и в пределах выделенного бюджета. Хотя улучшение существенно, все это выглядит мрачно по сравнению с традиционными техническими дисциплинами типа архитектуры или электротехники.

Успех реализации проекта ПО обусловлен пятью взаимосвязанными аспектами — см. *пятиугольник программной инженерии*, изображенный ниже. Эти аспекты следующие:

- жизненный цикл разработки ПО — глава 1;
- язык моделирования ПО — глава 2;
- инструментальные средства программной инженерии — глава 3;
- планирование работ над проектом ПО — глава 4;
- управление процессом разработки и сопровождения ПО — глава 5.



Основные учебные цели части 1 состоят в том, чтобы получить знания в следующих вопросах:

- сущность программной инженерии;
- стадии жизненного цикла и модели; в частности, итеративная и пошаговая разработка;
- языки моделирования ПО, в частности, UML — объектно-ориентированный язык моделирования;
- инструментальные средства программной инженерии для управления проектом, моделирования систем, интегрированного коллективного программирования, управления изменением и конфигурацией проекта;
- планирование работы над проектом и оценка бюджета;
- технологии отслеживания хода выполнения проекта;
- управление людскими ресурсами, привлеченными к проекту;
- управление рисками проекта;
- управление качеством ПО;
- управление изменением и конфигурацией.

Глава 1

Жизненный цикл разработки программного обеспечения

В обычном использовании термин «жизненный цикл» означает «изменения, которые происходят в жизни животного или растения» [18]. В программной инженерии термин «**жизненный цикл**» (на английском языке lifecycle — обычно пишется единым словом) применяется к искусственным системам ПО и означает изменения, которые происходят в «жизни» программного продукта. Различные стадии между «рождением» изделия и его возможной «смертью» известны как **стадии жизненного цикла**.

Изменения, а следовательно, и стадии, являются последовательными. Изделие *создается* поэтапно, за ряд стадий. Таким образомь, разработка является повторяющейся и пошаговой. В конечном счете, изделие поэтапно *выводится из работы* — его использование постепенно прекращается. Следовательно, и прекращение его работы является пошаговым. Разумно думать, что ПО в любое время, кроме стадии непосредственного внедрения, находится или в фазе создания, или в фазе снятия с производства. Сопровождение ПО, несмотря на его эволюционный характер, одновременно начинает и процесс снятия с эксплуатации.

Рис. 1.1 показывает типичные стадии жизненного цикла ПО (объясненные более подробно в разделе 1.2). Эти стадии следующие:

1. анализ требований;
2. проектирование системы;
3. реализация;
4. интеграция и внедрение;
5. процесс функционирования и сопровождения.

Рис. 1.1 демонстрирует, что как только программный продукт внедрен в организацию, он остается там навсегда, хотя и под различными «перевоплощениями». Организация уже не может вернуться к ручному способу ведения бизнеса. Однажды включенный в действие, программный продукт поддерживается «до смерти».

Сопровождение, даже если оно развивает систему, ведет, в конечном счете, к ухудшению ее первоначальной структуры. Система становится **унаследованной системой** — она не может быть больше «усовершенствована», и даже вспомогательное и корректирующее сопровождение становится большой проблемой. Вся система или основные ее компоненты должны постепенно удаляться. Осознание, что система является унаследованной, приводит к

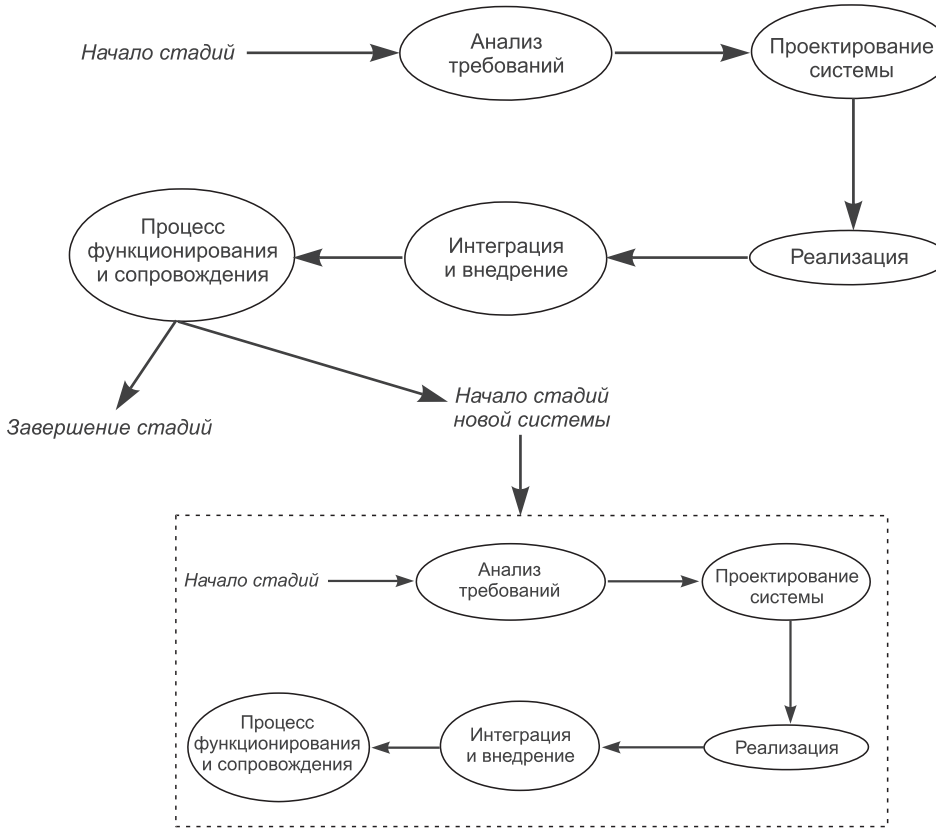


Рис. 1.1. Стадии жизненного цикла ПО

решению разработать новую систему. Это стимулирует новый жизненный цикл, показанный в нижней части рис. 1.1. Постепенное сокращение старой системы и синхронизация с новой системой проводится в параллель, пока новая система не будет полностью развернута для пользователей. Но даже и после развертывания старая система может оставаться в эксплуатации в течение некоторого времени, пока новая система не продемонстрирует свою полноценность.

Особенностью рис. 1.1 является отсутствие *тестирования* как стадии жизненного цикла. Тестирование, как и действия по управлению проектом, включая сбор *системы показателей* проекта, является всеобъемлющей деятельностью, которая выполняется на всех стадиях жизненного цикла.

1.1. Сущность программной инженерии

Понимание жизненного цикла ПО является необходимым условием понимания сущности программной инженерии — ее фундаментальной природы, контекста формирования ПО. Суть **программной инженерии** отражается в следующих ключевых выводах:

- система ПО меньше, чем информационная система предприятия;
- процесс создания и эксплуатации ПО является частью бизнес-процесса;
- программная инженерия отличается от традиционной инженерии;
- программная инженерия больше, чем программирование;
- программная инженерия напоминает моделирование;
- система ПО сложна.

1.1.1. Система ПО меньше, чем информационная система предприятия

Система ПО просто является частью намного большей **информационной системы предприятия**. Это означает, что разработка системы ПО является только частью (хотя и фундаментальной) разработки информационной системы предприятия. Диаграмма Венна на рис. 1.2 демонстрирует включение системы ПО в информационную систему предприятия. Она показывает также, что информационная система предприятия является компонентом предприятия как целого, и что предприятие является частью бизнес-среды.

Информационная система обеспечивает формирование и управление информацией в интересах людей. Часть этой информации генерируется автоматически компьютерными системами. Другая информация вводится людьми вручную. Дело в том, что информационные системы — социальные системы, которые включают и используют ПО и другие компоненты в интересах предприятия. Бенсон и Стандинг [7] перечисляют следующие компоненты информационной системы:

- люди;
- данные/информация;
- процедуры;
- ПО;
- аппаратное обеспечение;
- линии связи.

Рис. 1.2

Например, система управления счетами банка состоит из кассиров банка, данных/информации о клиентах и их счетах, процедур, управляющих изъятием и внесением денег, ПО, способного обработать данные/информацию, аппаратных средств, на которых может работать ПО (включая банковские автоматы), а также персональных и автоматизированных каналов связи, которые объединяют все эти компоненты в единую систему.

1.1.2. Процесс создания и эксплуатации ПО является частью бизнес-процесса

Процессом можно назвать нечто, где запланированы, организованы, скоординированы и выполнены в определенный период времени некоторые виды деятельности по производству продукта или сервиса. Различие между процессом создания и эксплуатации ПО, с одной стороны, и бизнес-процессом, с другой, определяется и связано с продуктом или сервисом, которые ожидается получить от этих процессов. Результат **процесса создания и эксплуатации ПО** — ПО. Результат **бизнес-процесса** — бизнес.

Имеются четкие отношения между ПО и бизнесом. ПО — потенциально главный вкладчик в бизнес-успех. ПО — часть бизнеса, но не наоборот. Фактически это отношение старшинства было отображено на рис. 1.2. Предприятие на рис. 1.2 — это другой термин для бизнеса. Цель функционирования предприятия состоит в том, чтобы сформировать цепочку создания ценностей, которая обеспечивает реализацию бизнес-назначения, задач и целей.

Различие между процессом создания, эксплуатации ПО и бизнес-процессом сродни различию между эффективностью процесса и результативностью. **Эффективность** (efficiency) означает делать что-то правильно. **Результативность** (effectiveness) означает делать правильную вещь. В организационных терминах результативность подразумевает достижение бизнес-назначения, задач и целей. Все они — то, что нужно получить как результат процесса *стратегического планирования*, проводимого на предприятии. Частью стратегического планирования является *бизнес-моделирование*. Следовательно, целью бизнес-процесса является обеспечение результативности.

В противоположность этому процесс создания и эксплуатации ПО должен обеспечить эффективность. Следовательно, возможна ситуация, когда процесс создания и эксплуатации ПО даст очень *эффективный* программный продукт или сервис, который будет *нерезультативен* для бизнеса. В лучшем случае нерезультативность может означать нейтральный результат с точки зрения бизнеса. В худшем случае это может сделать бизнес уязвимым для конкурентов и даже привести к банкротству.

Поэтому ясно, что процесс создания и эксплуатации ПО является характерной частью бизнес-процесса, жизненно важной для успеха предприятия. Чтобы обеспечить результативность наряду с эффективностью, процесс создания и эксплуатации ПО должен быть составной частью бизнес-процесса. В конце концов, решение разрабатывать ли программный продукт или сервис в первую очередь будет результатом стратегического планирования и бизнес-моделирования.

Процесс программной инженерии устанавливает соответствие ПО и бизнес-процессов. С одной стороны, разработка ПО все более и более внедряет-

ся в среду бизнес-моделирования. Главы 6 и 7 этой книги иллюстрируют явное проявление этой тенденции. С другой стороны, разработка ПО предназначена для поставки программных продуктов и сервисов, увеличивая для предприятия стоимость бизнеса. Это имеет отношение к трем **уровням управления**, которые бизнес-процессы обслуживают: оперативный, тактический и стратегический.

Помещение разработки ПО в среду бизнес-моделирования означает, что процесс создания и эксплуатации ПО получен из более широкой бизнес-модели, и он старается поддерживать и реализовывать конкретный бизнес-процесс в этой модели. Отсюда следует, что программный продукт/сервис не может быть только информационным сервисом. Он должен также реализовывать бизнес-операции или содействовать им. Проект информационной системы должен или явно определить бизнес-процесс, который он обслуживает, или, что лучше, он должен быть частью *системы управления знаниями*. Одним из аспектов такого проекта является координация между автоматическими информационными действиями, ручными действиями и творческими действиями по принятию решения.

Обычно система ПО обслуживает один уровень управления — оперативный, тактический или стратегический (рис. 1.3). *Оперативный уровень* обрабатывает оперативные бизнес-данные и документы, типа заказов и счетов. Это — царство OLTP-систем (**online transaction processing** — оперативная обработка транзакций), сопровождаемых обычной технологией *баз данных (БД)*. *Тактический уровень* обрабатывает информацию, полученную от анализа данных, типа ежемесячных тенденций в заказах продуктов. Это — царство OLAP-систем (**online analytical processing** — аналитическая обработка в реальном времени), которые сопровождает технология *хранилищ данных*. *Стратегический уровень* обрабатывает организационные знания, типа правил и фактов, обусловливающих высокий уровень выгодной продажи изделий. Это — царство систем знаний, которые сопровождает технология *баз знаний*.

Программные продукты/сервисы на оперативном уровне обязательны для предприятия. Без них современное предприятие не может функционировать.

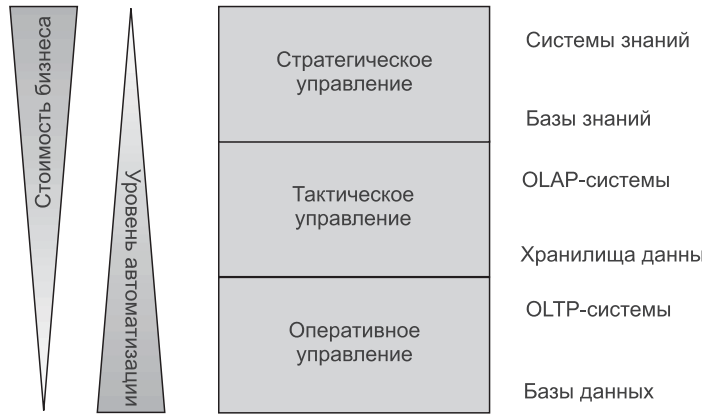


Рис. 1.3. ПО различных уровней управления

Однако одно оперативное ПО не дает предприятию никакого конкурентного преимущества. Ведь конкуренты также работают с подобными системами ПО. Бизнес-ценность ПО увеличивается с более высокими уровнями управления, к которым оно применяется.

Интересно, что программные продукты/сервисы, которые потенциально обладают наибольшей бизнес-ценностью для предприятия, наиболее трудно автоматизировать. И это понятно. Стратегическое управление — из области организационных *знаний* и *мудрости*. Как отмечено Бенсоном и Стандингом: «Мудрость и знания существуют только в умах людей. Когда люди говорят относительно знания чего-то, типа номера телефона, на самом деле они говорят о данных. Понимание того, как использовать эту часть данных — знание. Решение не вызывать кого-то в 3.00 утра — пример мудрости» [7]. Обработка и передача знаний (не будем упоминать мудрость) являются и будут оставаться, главным образом, социальным явлением, мотивационным и интуитивным, но не техническим и не предсказуемым.

1.1.3. Программная инженерия отличается от традиционной инженерии

Факт, что система ПО является компонентом информационной системы, подразумевает, что программная инженерия — лишь часть более широкой дисциплины — инженерии систем. Следовательно, инженер ПО должен понимать требования всей системы и должен быть компетентен в ее предметной области, чтобы проектировать интерфейсы, которыми ПО должно снабдить внешние устройства системы. Инженер ПО должен также понимать, что получение некоторых данных или обработку информации лучше реализовать с помощью аппаратных средств ЭВМ, чем с помощью ПО, и что некоторую обработку не нужно автоматизировать вообще.

Инженерия систем связана с изучаемыми принципами, которые определяют внутреннюю работу сложных систем. Существует длинная история инженерии систем в **традиционных** технических дисциплинах, типа проектирования механических или электрических систем. Разработанные принципы инженерии систем формализованы в математических моделях. Модели утверждаются и используются в технических изделиях. Эти изделия материальны по своей природе — мосты, строения, электростанции.

С программными продуктами дело обстоит иначе. Как выявлено в оригинальной работе Брукса [15], ПО нематериально по природе. Классические математические модели применяются к некоторым, но не ко всем аспектам ПО. ПО определено в нечетких терминах — «хороший», «плохой», «приемлемый», «удовлетворяющий требованиям пользователя» и т. д. Подобные качества используются в области обслуживания, где качество связано с нечеткими терминами типа «хорошее обслуживание», «удобство клиента», «компетентность», «знание работы» и т. д. Программная инженерия может заниматься «нечеткими» проблемами, но это не подразумевает, что они должны быть менее строгими или недоказуемыми. Очевидно, что программная инженерия должна использовать различные области математики, типа нечеткой логики или нечетких множеств, обеспечивающие строгость и доказательность.

В этом контексте следует обратить внимание, что *строгость* — это не то же самое, что *формальность*. Процесс создания и эксплуатации ПО может быть строгим даже в том случае, когда он не доказан формально в соответствии с математическими законами. Действительно, дело обстоит так даже в классической математике. Как указано Гецци и др. [34]: «Учебники по функциональному исчислению строги, но редко формализованы: доказательства теорем провсдены очень осторожно, как последовательности промежуточных выводов, которые ведут к заключительному утверждению; каждый дедуктивный шаг полагается на интуитивное оправдание, которое должно убедить читателя в его законности. И почти никогда мы не видим доказательств, выполненных формальным способом в терминах математической логики». Заинтересованных читателей отсылаем к оригинальной статье о социальных процессах и доказательствах теорем и программ Де Милло и др. [24].

Программная инженерия не должна быть бедной падчерицей традиционной инженерии. Это совсем другое. Никто в традиционной инженерии не ожидает, что мост, построенный по математическим моделям, разрушится. Точно так же «хорошее» ПО, «удовлетворяющее требованиям пользователя», не должно терпеть неудачу. Однако имеется одно «но» — все это при условии, что тем временем решительно не изменятся требования и ожидания пользователя или внешние обстоятельства.

Никто не ожидает, что мост будет перемещен на десять метров после того, как он был построен. Точно так же не следует ожидать, что программный продукт успешно выполнит различные задачи после того, как будет создан. Если это то, что нам нужно, тогда ПО создано удачно. ПО только тогда должно стать непригодным или недопустимым, когда бизнес создает новые условия или меняется внешняя среда. Если речное русло переместится на десять метров из-за недавнего наводнения, инженер-строитель не может быть обвинен, и не следует ожидать, что существующий мост можно будет легко переместить, чтобы приспособить к новому руслу.

Все сказанное означает, что разработчик ПО должен быть готов создавать ПО, которое можно приспосабливать к изменениям. Этого требует сама природа ПО. ПО должно иметь **возможность сопровождения**: понятно, ремонтопригодно и расширяемо. Это то, что отличает ПО от моста и делает программную инженерию отличной от традиционной инженерии.

Каждая система ПО уникальна, и процесс ее создания уникален. В отличие от традиционных технических дисциплин прикладной программный продукт не *производится*, он *реализуется*. Это — не автомобиль или рефрижератор. Программный продукт должен быть реализован, чтобы приспособить его к окружающей среде. Каждый случай системы ПО уникален: либо она построена на пустом месте, либо переделана из имеющегося в наличии коммерческого пакета программ (COTS — commercial off-the-shelf software — коммерческие коробочные программные продукты). Только системное ПО и инструментальные средства ПО, типа операционных систем (ОС) и текстовых процессоров, были когда-то произведены в широком масштабе. *Прикладное ПО*, которое является предметом этой книги, реализуется, а не производится.

1.1.4. Программная инженерия больше, чем программирование

В начальных страницах части 1 этой книги программная инженерия была определена как «область информатики, имеющая дело с созданием систем ПО, которые являются настолько большими или настолько сложными, что создаются коллективом или коллективами инженеров» [34]. Определение делает акцент на двух понятиях: «коллективы людей» и «сложные системы».

В **программировании** используется термин «разделение кода» — написание серий инструкций, чтобы заставить компьютер выполнить специфическую задачу. Если задача большая, для программирования может быть использован коллектив программистов, но каждый акт программирования — прежде всего персональная деятельность. Программирование — навык. Учитывая определение и спецификацию задачи, программист применяет свои навыки, чтобы выразить проблему на языке программирования.

Программная инженерия больше, чем программирование. Она обращается к сложным проблемам, которые не могут быть решены, используя одно программирование. Сложные системы должны быть разработаны прежде, чем они будут запрограммированы. Подобно строительной индустрии, над сложной системой должен поработать *архитектор*, прежде чем она будет построена. Она должна быть *разбита на модули*, используя обобщение и метод «разделяй и властвуй». Каждый модуль затем должен быть тщательно специфицирован и определены его интерфейсы к другим модулям, прежде чем его отдавать программистам для кодирования.

Программист имеет ограниченное понимание всей системы. Он/она кодирует одновременно один программный модуль — **компонент ПО**, который должен быть объединен (инженером ПО) с другими компонентами, чтобы сконфигурировать рабочую систему. (Конечно, это различие между программистом и инженером ПО приведено только для того, чтобы проиллюстрировать проблему. Практически различие может быть, а может и не быть.)

Часто инженеру ПО доступны различные **версии** одного и того же компонента. **Конфигурация** ПО выполняется объединением определенных версий различных компонентов. По этой причине можно иметь различные конфигурации одной и той же системы.

Прежде чем система будет разработана, инженер ПО должен разобраться с требованиями к ней. Это означает, что должен быть сделан и определен на некотором языке моделирования анализ требований. Стандартный язык моделирования в современной практике — UML (**Unified Modeling Language** — унифицированный язык моделирования). И анализ, и синтез моделей выполняются в UML.

Инженер ПО создает такую UML-модель системы, по которой может быть создан исходный код программы. Программисты могут начинать работу с этого момента, но инженер ПО остается ответственным за циклическое проектирование между проектом и кодом. **Циклическое проектирование** — итеративный процесс, представляющий как прямое (от проекта к коду), так и обратное (от кода к проекту) проектирование.

Наконец, программная инженерия — работа коллектива. Коллективом нужно управлять. Следовательно, программная инженерия требует **управле-**

ния проектом и воздействует на него. Это руководство включает планирование, составление бюджета и разработку графика, управление качеством и управление рисками, управление конфигурацией и изменениями.

Резюмируя все, можно сказать, что программная инженерия связана с обеспечением структурного решения системы, с проектированием структурных компонентов, с объединением компонентов в рабочую систему, с прямым и обратным проектированием, с руководством проектом и т. д. Программная инженерия — сложный процесс, в пределах которого программирование является полезным ремеслом.

1.1.5. Программная инженерия напоминает моделирование

Программная инженерия напоминает моделирование [56]. **Модели** — абстракции реального мира. Они являются абстрактными представлениями действительности. Так что же такое компьютерная программа — модель или действительность? Ну, хорошо, можно утверждать, что программа, находящаяся в памяти компьютера или напечатанная на бумаге — действительность. Однако цель программирования — не код программы сам по себе; скорее эта цель — функциональные возможности, которые он обеспечивает. Является ли футбольная игра, сыгранная на компьютере, реальностью? Что, действительно, это реальность? Ясно, что это риторические вопросы.

Абстракция — мощная технология в программной инженерии. Позволяя концентрироваться на важных аспектах проблемы и игнорировать аспекты, которые являются в настоящее время несущественными, абстракция позволяет систематически справляться со сложностью проблемы (раздел 1.1.6). Абстракция применяется также и к программным продуктам и процессу создания ПО. **Модель процесса создания ПО** является абстрактным представлением этого процесса. В практических терминах модель процесса создания ПО определяет стадии жизненного цикла и то, как они взаимодействуют (раздел 1.2). **Модель программного продукта** — это его абстрактное представление. Она определяет дискретный продукт в дискретные стадии жизненного цикла.

Модель процесса создания ПО определяет, какие программные продукты, требуемые для обеспечения стадий жизненного цикла, создавать на различных уровнях абстракции. Далее приведен список моделей базовых программных продуктов:

- **Модель требований** — сравнительно неформальная модель, которая охватывает требования пользователя и описывает систему в терминах ее бизнес-ценности.
- **Модель спецификаций** — модель, которая определяет требования к более формальному использованию терминов, применяя язык моделирования типа UML.
- **Структурная модель** — модель, которая определяет желаемую структуру системы.
- **Детальная модель проекта** — модель, которая определяет характеристики программного/аппаратного обеспечения, необходимые для реализации программирования.

- **Программная модель** — конструктивная модель, которая представляет окончательную выполнимую модель ПО.

Каждая из этих моделей программного продукта может быть разделена далее для ее детализации. Например, детальная модель проекта может включать модель пользовательского интерфейса, модель БД, модель программной логики и т. д.

Наконец, подход к программной инженерии, используемый при создании системы, влияет на абстракции моделирования. Два главных подхода — в старом стиле функциональной (процедурной, командной, структурной) разработки и в современном стиле объектно-ориентированной разработки.

Функциональный подход разбивает сложную систему до управляемых единиц, используя прием, известный как функциональная декомпозиция. Для этой цели используется технология, называемая моделированием потока данных. Модель ПО последовательно делится на процессы (при уменьшении уровня абстракции), связанные с потоками данных.

Объектно-ориентированный подход разбивает систему на пакеты/компоненты классов, связанные различными отношениями. Абстракция может применяться, формируя вложенные структуры, то есть пакет/компонент может содержать многие уровни других пакетов/компонентов. Это книжная квинтэссенция объектно-ориентированной программной инженерии.

1.1.6. Система ПО сложна

Системы ПО *сложны*. В прошлом ПО было монолитно и процедурно по своей природе. Типичная программа прошлого, написанная на КОБОЛе, была единственной сущностью, использующей подпрограммы, вызываемые по мере необходимости. Логика программы была последовательна и предсказуема. Сложность такого ПО была просто следствием его размера.

Современное объектно-ориентированное ПО является распределенным (оно может находиться на многих узлах компьютерной сети) и его выполнение случайно и непредсказуемо. Размер современного ПО — сумма размеров его компонентов. Каждый компонент разработан так, чтобы быть ограниченного управляемого размера. В результате размер не является главным фактором в сложности современного ПО.

Сложность современного ПО заключается в «проводах», то есть в связях и коммуникационных путях между компонентами. Связи между компонентами создают зависимости между распределенными компонентами, которые могут быть сложны для понимания и управления. Трудность усугубляется тем, что компоненты часто разрабатываются и управляются людьми и коллективами, даже не известными друг другу.

Рис. 1.4 представляет возможную объектно-ориентированную систему, в которой объекты различных пакетов связываются без разбора. Это создает сеть внутренней связи объектов. В диаграмме сложность в пределах отдельных пакетов (компонентов) все еще управляема из-за ограниченного размера пакетов. Однако зависимости, созданные связями между пакетами, будут расти по экспоненте с добавлением новых пакетов. Кто должен обеспечивать управление такими зависимостями, не всегда ясно, поскольку обязанности обеспечения управления для пакетов остаются за различными коллективами.

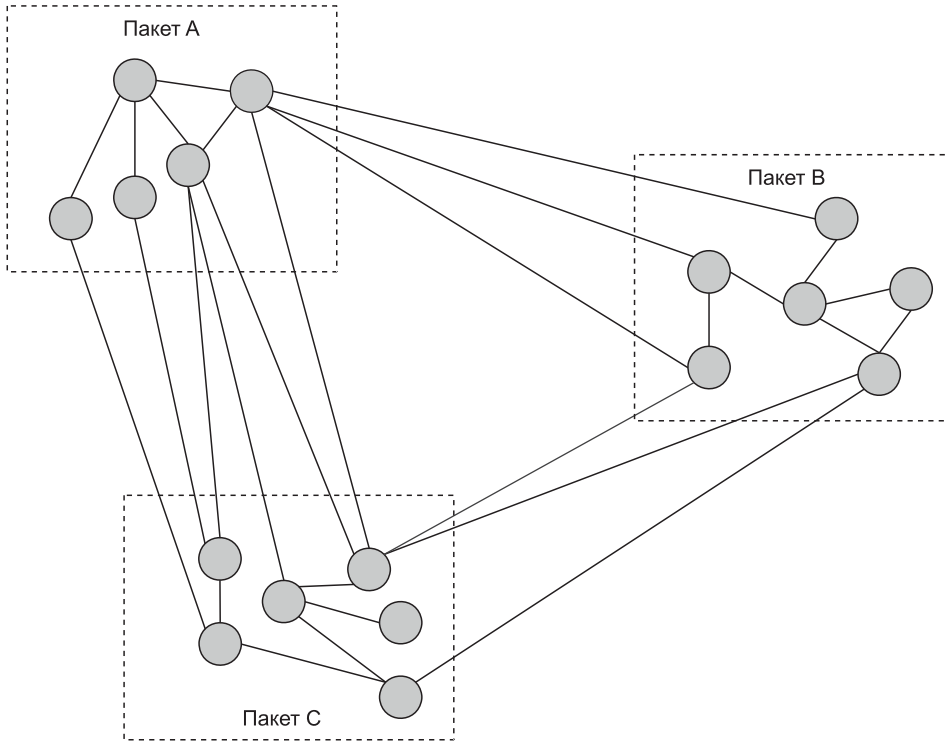


Рис. 1.4. Сложность в «проводах»

Более важен тот факт, что любой объект в одном пакете может связываться с любым объектом в другом пакете; это создает *потенциальные зависимости* между всеми объектами в системе. Такой факт означает, что изменение какого-то объекта может потенциально повлиять (вызвать «*эффект ряби*») на любой другой объект в системе.

Более формально: совокупной мерой зависимости объектов с неограниченными межпакетными (межкомпонентными) коммуникационными связями является число различных комбинаций пар объектов. Такая мера может быть вычислена, используя метод теории вероятностей, известный как *правило комбинаторики*. Ниже приводится формула для вычисления совокупной зависимости класса (`CCD` — cumulative class dependency) в системе с `n` классами объектов. Для 5 классов `CCD` равняется 10. Для 57 классов (например) CCD равняется 1596. Такой рост сложности быстро становится *неприемлемым*.

$$_{n}CCD_{2} = \frac{n!}{2!(n-2)!}$$

Формула вычисляет наихудшую сложность, когда каждый объект связывается со всеми другими объектами. Хотя самый плохой вариант практически маловероятен, он должен быть принят при любом анализе влияния зависимости, проводимом в системе (просто потому, что реальные зависимости заранее

не известны). Если изменение в классе может потенциально воздействовать на любой другой класс, то этот факт должен быть проверен для гарантии, что изменение не вызовет неприятных последствий. Системы, допускающие произвольную сеть связи объектов, подобно изображенной на рис. 1.4, считаются неприемлемыми с точки зрения возможности программной инженерии. Они непонятны, неремонтопригодны и нерасширяемы.

Решение дилеммы находится в замене сетей объектов иерархиями (древовидными структурами) объектов. Все приемлемые сложные системы имеют *иерархическую* форму. Эта тема настолько важна, что ей посвящена отдельная глава (глава 9). На рис. 1.5 просто показано, как можно уменьшить сложность системы, допуская только единственный канал связи между двумя пакетами. Каждый пакет определяет интерфейсный объект (это может быть интерфейс Java-стиля или так называемый доминантный класс), через который осуществляется вся связь с пакетом. Несмотря на добавление трех дополнительных объектов, сложность системы, изображенной на рис. 1.5, явно уменьшена по сравнению с той же самой системой, изображенной на рис. 1.4.

Обратите также внимание на то, что проект на рис. 1.4 выбивает из рук инженера ПО основной инструмент — механизм **абстракции**. Абстракция позволяет нам рассуждать относительно отобранных частей, не учитывая несущественные детали (абстрагируясь от них). Хотя объекты на рис. 1.4 сгруп-

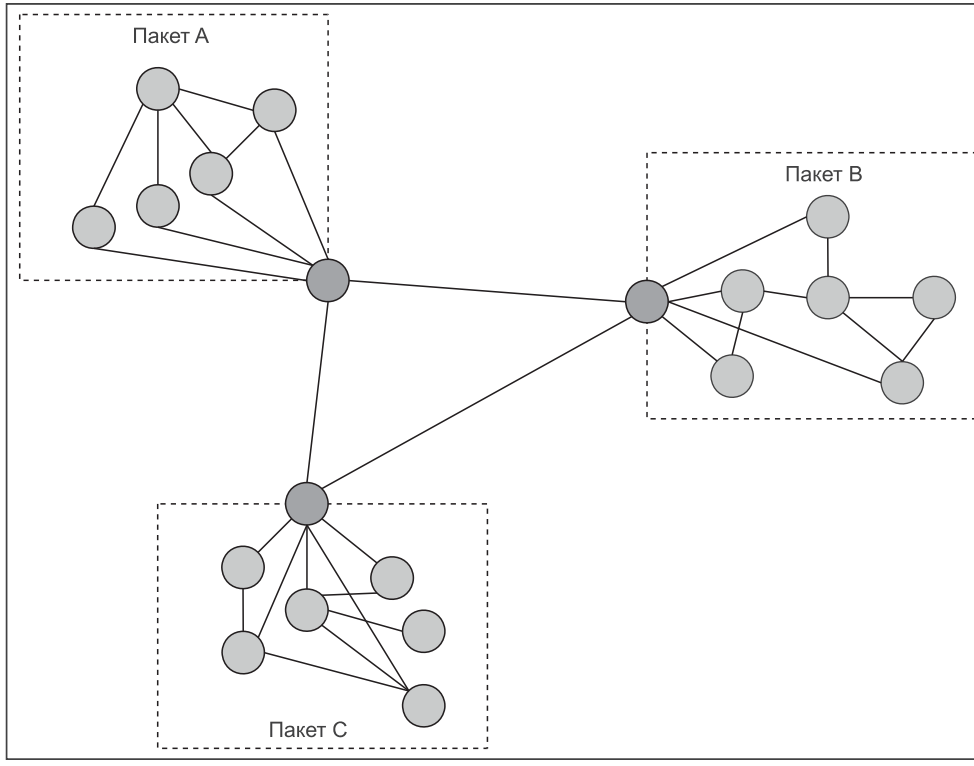


Рис. 1.5. Понижение сложности добавлением интерфейсов между пакетами

пированы в три пакета, сложность системы (измеренная как совокупная зависимость классов) та же самая, как и в подобной системе совсем без пакетов. Пакеты на рис. 1.4 не дают никакого полезного уровня абстракции. Они могут вообще отсутствовать.

1.2. Стадии жизненного цикла

Жизненный цикл ПО — это абстрактное представление процесса создания и эксплуатации ПО. Он определяет для проекта разработки ПО стадии, шаги, действия, методы, инструменты, а также то, что ожидается сдавать. Все это определяет стратегию разработки ПО.

Существует ряд полезных моделей жизненного цикла (раздел 1.3), которые находятся друг с другом в общем согласии по стадиям жизненного цикла, но отличаются важностью отдельных стадий и взаимодействиями между ними. Стадии жизненного цикла, принятые в этой книге, были определены в начале этой главы. Они следующие:

1. анализ требований;
2. проектирование системы;
3. реализация;
4. интеграция и внедрение;
5. процесс функционирования и сопровождения.

1.2.1. Анализ требований

«**Требования пользователя** — это утверждения на естественном языке плюс диаграммы, содержащие сведения, какие услуги ожидаются от системы, и ограничения, при которых система должна работать» [96]. **Анализ требований** включает действия по их определению и составлению их списка. В современной практике анализу требований помогает хорошая степень технической строгости, и поэтому эти требования иногда отождествляют с **техническими требованиями**.

Определение требований, оказывается, одна из самых больших проблем любого жизненного цикла разработки ПО. Пользователям часто неясно, что они требуют от системы. Часто они не знают реальные требования, преувеличивают их, предъявляют требования, которые противоречат требованиям коллег и т. д. Имеется также риск, как и в любом общении между людьми, что истинное значение требования будет неправильно истолковано. Разработчики часто сталкиваются с подобными анонимными заявлениями: «Я знаю, что Вы полагаете, что Вы поняли то, что Вы думаете, что я сказал, но я не уверен, что Вы сделали то, что Вы слышали, и не то, что я подразумевал».

Имеется много методов и технологий выявления требований. Они включают [64]:

- интервьюирование пользователей и экспертов в предметной области;
- анкетные опросы пользователей;
- наблюдение, как пользователи выполняют свои задачи;
- изучение существующих документов системы;

- изучение подобных систем ПО, чтобы выяснить состояние в предметной области;
- изучение опытных образцов рабочих моделей для определения и подтверждения требований;
- объединенные совещания разработчиков и клиентов по разработке приложения.

Спецификация требований следует за выявлением требований. В настоящее время UML — стандартный язык моделирования для спецификации требований (так же как и для проектирования системы). Требования определяются как в графических моделях, так и с помощью текстовых описаний. Поскольку сложную систему нельзя понять с единственной точки зрения, модели снабжаются дополнениями и при необходимости перекрестными точками зрения на систему.

И графические представления, и текст размещаются в хранилище специального CASE-средства (**computer assisted software engineering** — автоматизированная программная инженерия). Данное средство облегчает изменения в моделях, если это потребуется. Оно позволяет интегрировать различные модели с перекрывающимися концепциями. CASE-средство позволяет также выполнять преобразования между моделями анализа (где это возможно) и помогает в преобразованиях моделей проектирования.

Анализ требований завершается созданием **технического задания** [64]. Большинство организаций использует некоторые шаблоны для технического задания. Шаблон определяет структуру документа и дает руководящие принципы того, как его писать. Основная часть технического задания содержит модели и описания сервисов и ограничений системы. *Сервисы системы* (то, что система должна делать) часто делятся на функциональные требования и требования к данным. *Ограничения системы* (то, чем система ограничена) включают соображения, связанные с пользовательским интерфейсом, работой, безопасностью, эксплуатационными условиями, политическими и юридическими ограничениями и т. д.

Как уже мимоходом упоминалось, результат каждой стадии жизненного цикла должен быть утвержден и проверен. Профессиональный подход к тестированию требует внутри организации создания группы по гарантии качества ПО (SQA — **software quality assurance**). Коллектив этой группы состоит из профессиональных системных испытателей. Он работает достаточно независимо от разработчиков. Чтобы обеспечить целиком всю работу процесса, именно коллектив SQA-группы, а не разработчики, является ответственным за качество программного продукта (то есть SQA отвечает за все невыявленные несоответствия и дефекты в ПО).

Тестирование абстрактных моделей затруднено, поскольку большую часть времени оно не может быть автоматизировано. *Сквозной контроль* и *инспекции* — вот две популярные и эффективные технологии. Данные технологии схожи. Это предварительные встречи разработчиков и пользователей, на которых «проходятся» по техническому заданию и документам. Обсуждение, которое происходит на встречах, вероятно, раскроет некоторые проблемы. Сущность этих технологий заключается в том, что в течение встреч идентифициру-

ются проблемы, но их решение не определено, и нет никакого «указующего перста» на людей, потенциально ответственных за эти проблемы.

1.2.2. Проектирование системы

«**Проектирование ПО** — это описание структуры ПО, которое будет реализовано, данных, которые являются частью системы, интерфейсов между компонентами системы и, иногда, используемых алгоритмов» [96]. Это определение совместимо с определением системы ПО как объединения структур данных и алгоритмов. В информационных системах предприятий структуры данных подразумевают БД. Алгоритмы не всегда полностью описываются во время проектирования, чтобы оставить некоторый уровень свободы выполнения программистам (ведь говоря прямо, проектировщики — это не программисты, и они не в состоянии выбрать умные алгоритмические решения).

Проектирование начинается там, где заканчивается анализ. Столь же истинно и тривиально утверждение, что линия, отделяющая анализ от проектирования, во многих проектах не столь уж и ясна. Теоретически проблема проста. Анализ — моделирование, не ограниченное никакой реализацией (аппаратного/программного обеспечения). Проектирование — моделирование, которое учитывает платформу, на которой должна быть реализована система.

На практике различие между анализом и проектированием размыто. Этому имеются две главные причины. Во-первых, современные модели жизненного цикла являются *итеративными* и *пошаговыми* (раздел 1.3.2). В большинстве таких моделей при разработке в любой момент времени имеются многочисленные разнообразные версии ПО. Некоторые из версий находятся в процессе анализа, другие — в процессе проектирования; некоторые в разработке, другие в производстве и т. д. Во-вторых, и это более важно, для анализа и проектирования используется один и тот же язык моделирования (UML). Переход от анализа к проектированию «по готовности» предпочтительней, чем переход между различными представлениями. Модель анализа уточняется в модели проекта простым дополнением деталей спецификации. Провести линию раздела между анализом и проектированием в таких обстоятельствах очень трудно.

Проектирование, обсужденное выше, более точно называется **детальным проектированием**, то есть проектированием, которое добавляет детали к моделям анализа. Но имеется другой аспект проектирования системы, а именно структурное проектирование. **Структурное проектирование** связано с определением структуры системы, которая должна быть детально спроектирована и которой следует твердо придерживаться, а также с принципами и образцами внутренних коммуникаций между компонентами.

Структурное проектирование задает «красоту» системы. Главная цель структурного проектирования состоит в том, чтобы получить систему, которая является *приемлемой* — понятной, ремонтопригодной и расширяемой. Детальный проект должен соответствовать структурному проекту. Из-за расплывчатой линии раздела между анализом и детальным проектированием некоторые ранние структурные решения, может быть, придется заново выбрать внутри технического задания или даже раньше (но после определения требований).

Тестирование структурного проекта двулико. Во-первых, преимущества структуры, предложенной проектировщикам, должны быть продемонстрированы. Нужно показать, что структура поддерживает сложность ПО, гарантирует возможность сопровождения, упрощает разработку и т. д. Во-вторых, тестирование структурного проекта должно подтвердить, что проект компонентов соответствует принципам и шаблонам принятой структуры.

Тестирование детального проекта также имеет два аспекта. Во-первых, чтобы быть тестируемым, должна быть возможность трассировать детальный проект. **Управление трассировкой** — целая отрасль программной инженерии, занимающаяся поддержанием связей между продуктами ПО в различных стадиях разработки. В случае детального проекта каждый продукт проекта должен быть связан с требованиями в техническом задании, которое мотивировало производство того продукта. Наличие продукта еще не подразумевает, что это требование обеспечено. Следовательно, второй аспект тестирования проекта использует сквозной контроль и инспекции, чтобы оценить качество изделия проекта.

1.2.3. Реализация

Реализация в большей мере связана с программированием. Но программирование подразумевает не только группу людей, сидящих в общем помещении и кодирующих на некотором языке программирования в соответствии со спецификацией проекта. Программирование предполагает намного более интеллектуальные требования, чем это. Как указано в предыдущих разделах, проекты будут «недоопределены» в некоторых областях, когда они попадают к программистам, особенно в области проектирования алгоритмов. Завершение спецификаций требует дополнительного проектирования прежде, чем можно будет начать кодирование. В этом смысле программист — тоже проектировщик.

Программист — *инженер, имеющий дело с компонентами*. Сегодняшнее программирование редко выполняется на пустом месте. Большая часть программирования основана на многократном использовании уже созданных компонентов. Это означает, что программист должен иметь знание о компонентах ПО и должен знать, как найти это ПО, чтобы добавить к нему новые закодированные компоненты приложения. Это трудный вопрос.

Программист — *инженер, работающий в двух направлениях*. Программирование начинается с преобразования проекта в код. Начальный код не должен программироваться вручную. Используя CASE-средства и IDE-средства (**integrated development environments** — интегрированные средства разработки), код начинает формироваться (прямое проектирование) из моделей проекта. После того как эта работа будет сделана, произведенный код должен быть скорректирован вручную, чтобы заполнить отсутствующие части (эти «части» существенны и наиболее трудны в программировании). После того как код будет модифицирован программистом, он может обратно воздействовать на модели проекта, корректируя их. Эти технические операции в прямом и обратном направлении называются **циклическим проектированием**.

Если все это звучит просто, на самом деле это не так. Циклическое проектирование несовершенно. Существующие инструментальные средства мощны и умны, но они все еще не приспособлены должным образом для выполнения некоторых видов работы. Чтобы сохранить проект и реализацию синхронными, и проектировщики, и программисты должны знать ограничения и выполнять ручные исправления и дополнения, когда это необходимо. Большая часть ответственности в этой задаче падает на менеджеров проекта. Они должны планировать и контролировать работу проектировщиков и программистов так, чтобы те не наступали на ноги друг другу. Практическое требование заключается в том, чтобы прямое и обратное проектирование не наложились по времени [62].

Во многих проектах реализация — самая длинная из стадий разработки. В некоторых моделях жизненного цикла, типа быстрой разработки ПО (раздел 1.3.2), реализация является доминирующей стадией разработки. Реализация — подверженная ошибкам деятельность. Время, потраченное на творческое написание программ, может быть меньше, чем время, потраченное на отладку программы и ее тестирование.

Отладка — это процесс удаления из ПО «блох» — ошибок в программах. Ошибки в синтаксисе программы и некоторые логические ошибки могут быть определены и исправлены коммерческими средствами отладки. Другие ошибки и дефекты должны быть обнаружены во время тестирования программы. **Тестирование** может иметь форму просмотра кода (сквозной контроль и инспекции) или может основываться на выполнении программы (наблюдение за поведением программы во время ее выполнения). Управление трассировкой поддерживает возможность использования тестирования, если программы удовлетворяют требованиям пользователя.

Имеются два вида *тестирования, основанного на выполнении программы*: **тестирование на основе технических требований** (тестирование черного ящика) и **тестирование на основе кода** (тестирование белого ящика). Оба вида используют ту же самую стратегию задания программе входных данных и наблюдения, тот ли выходной результат получается, который ожидался. Различие заключается в том, что при тестировании на основе технических требований программе задаются данные без какого-либо учета логики работы программы. Считается, что программа должна вести себя разумно при любых входных данных. В тестировании на основе кода используются такие входные данные, которые позволяют проверить определенные пути выполнения программы — столько путей и настолько разнообразных, насколько это возможно. Поскольку тестирование на основе технических требований и тестирование на основе кода обнаруживают различные виды ошибок и дефектов, нужно использовать их оба.

1.2.4. Интеграция и внедрение

«Целое — больше чем сумма его частей». Этот оригинальный афоризм Аристотеля (384–322 гг. до н. э.) охватывает сущность интеграции системы и ее внедрения. **Интеграция** собирает приложение из набора компонентов, предварительно созданных и проверенных. **Внедрение** — передача системы клиентам для использования в производстве.

Интеграция ПО означает переход от «программирования в малом» к «программированию в большом» [34]. Информационные системы предприятий — все достаточно большие и сложные системы (раздел 1.1.6) и для них интеграция — существенная стадия в жизненном цикле. По этому поводу есть другое высказывание: «Для каждой сложной проблемы имеется простое решение, которое не будет работать» (Х. Л. Менкен). Интеграция не может быть проигнорирована. Это отдельная стадия по праву, даже если ее иногда трудно отделить от реализации, как, например, в случае *непрерывной интеграции* в быстрой разработке (раздел 1.3.2).

Интеграцию также трудно отделить от тестирования. Фактически, стадия интеграции жизненного цикла часто упоминается и обсуждается под термином **тестирования интеграции**. При широком использовании итеративных моделей жизненного цикла (раздел 1.3.2) ПО создается как последовательность быстрых пошаговых реализаций. Каждый шаг — интеграция компонентов, до этого проверенных индивидуально, однако при этом до внедрения саму эту интеграцию системы необходимо сначала проверить.

В значительной степени интеграция определяется структурным проектом системы. В свою очередь, структура системы определяет ее компоненты и зависимости между ними. Особенно важно, чтобы структурное решение было в виде *иерархии* или древовидной структуры (раздел 1.1.6). Иерархия (древовидная структура) означает устранение любых циклических зависимостей между компонентами. В случае циклических зависимостей тестирование интеграции отдельных шагов создания ПО (конструкций) может оказаться невозможным.

Рассмотрим структуру зависимых компонентов, где компонент C_i использует (зависит от) компонент C_j, а C_j использует компонент C_k. Предположим, что C_i и C_j уже были реализованы и индивидуально протестированы, а компонент C_k должен быть еще создан. Задача состоит в том, чтобы объединить C_i и C_j. Эта задача требует программирования **испытательной заглушки** для C_k, то есть части кода, которая моделирует поведение отсутствующего компонента C_k. Заглушка обеспечивает среду интеграции для выполнения компонента C_j. Обычный путь создания заглушки — сделать ее такой, чтобы она обеспечивала те же зависимости входа/выхода, что и в окончательном компоненте C_k, и формировать результаты, ожидаемые для компонента C_j, жестким кодированием их или читая их из файла.

Все это работает, если только между C_j и C_k нет никаких циклических зависимостей. В присутствии циклических зависимостей оба компонента должны быть полностью реализованы и индивидуально проверены до интеграции. Но даже и тогда циклические зависимости создадут кошмар тестирования. С большими циклами в структурном проекте тестирование интеграции должно быть выполнено как единая операция, называемая *тестированием «одним махом»*. Тестирование «одним махом» может быть успешно выполнено только для небольших программных решений. Это не очень разумно в современной разработке ПО.

Возвращаясь к примеру, предположим теперь, что C_j и C_k были реализованы и индивидуально протестированы, но C_i должен быть еще разработан. Как мы должны интегрировать C_j и C_k, чтобы объединенный экземпляр

(конструкция) получал данные и другой контекст, которые стандартно давал бы компонент C_i, и так, чтобы мы могли утверждать, что этот экземпляр работает таким образом, как ожидается при наличии C_i? Чтобы сделать это, для C_i должен быть запрограммирован **тестовый драйвер**, чтобы «управлять» интеграцией.

В целом интеграция требует написания дополнительного ПО, заглушек и драйверов, которые полезны только во время интеграции. Это дополнительное тестирующее ПО называется **средствами тестирования** или «*лесами для тестирования*», по аналогии с временными лесами, используемыми в строительной индустрии.

Интеграция может проводиться *сверху вниз* (от корня иерархии зависимостей) или *снизу вверх* (от компонентов в листьях иерархии). Нисходящий подход требует реализации заглушек. Восходящий подход требует драйверов. В действительности интеграция редко следует только одному из этих подходов. Смешанный подход, иногда называемый «*из середины*», является преобладающим.

Подобно интеграции, внедрение — не одноразовая операция. ПО внедряется своими версиями. Каждая *версия* объединяет ряд экземпляров (конструкций), которые предлагаются совместно и функционально полезны пользователям. Перед внедрением ПО *системно тестируется* разработчиками в реальных условиях. Оно иногда называется *альфа-тестированием*. За альфа-тестированием следуют **приемочные испытания** специалистами пользователя. Это иногда называется *бета-тестированием* (альфа- и бета-тестирование — термины, в большей мере используемые при тестировании системного ПО от имени продавцов ПО, в противоположность разработке прикладного ПО).

Кроме самой системы и приемочных испытаний внедрение включает ряд других действий. Наиболее важное из этих действий — *обучение пользователей*. Практически обучение пользователей может начаться задолго до того, как система будет подготовлена к выпуску. Обучение совпадает по времени с производством *документации для пользователя*.

1.2.5. Процесс функционирования и сопровождения

Процесс функционирования представляет стадию жизненного цикла, когда программный продукт ежедневно используется, а работа предыдущей системы (ручной или автоматизированной) постепенно сокращается. *Постепенное сокращение* — обычно поэтапный процесс. В ситуациях, где это возможно и выполнимо, организация обычно использует новую и старую системы параллельно в течение некоторого времени. Это обеспечивает отход назад, если новое изделие не удовлетворяет бизнес-требованиям.

Процесс функционирования совпадает с началом **сопровождения изделия**. В программной инженерии сопровождение имеет несколько отличное значение по сравнению с обычным использованием этого слова. Во-первых, сопровождение — это не только незапланированная фиксация возникающих проблем. Сопровождение планируется и оплачивается на ранних стадиях жизненного цикла. Во-вторых, сопровождение включает развитие изделия. В некоторых итеративных моделях жизненного цикла (раздел 1.3.2), могут даже оказаться трудноразличимыми разработка и сопровождение.

В литературе [60, 34] сопровождение обычно разделяется на:

- *корректирующее* (действия по обслуживанию) — выявление дефектов и ошибок, обнаруженных при работе;
- *адаптивное* — изменение ПО в ответ на изменения в вычислениях или в бизнес-среде;
- *совершенствующее* — развитие изделия путем добавления новых особенностей или улучшения его качества.

Стоимость сопровождения значительна за время жизни ПО. Она значительна потому, что изделие остается в действии в течение долгого времени. Большие системы предприятий настолько фундаментальны, что они сохраняются действующими с помощью использования любых доступных технологий «поддержки жизни». Такие системы называются **унаследованными системами**. Конечно, их следовало бы удалить, но для них нет никакой замены. Действительно, когда решение об *удалении* системы, в конечном счете, принято, это делается не потому, что унаследованная система больше не полезна для организации, а потому, что нет технических возможностей дальше сохранять ее в работе. Не должно быть сюрпризом, что сопровождение является главной причиной, почему это технически невозможно — через какое-то время сопровождение настолько уничтожает структурную ясность ПО, что оно дальше не может поддерживаться.

1.3. Модели жизненного цикла

Жизненный цикл ПО определяет «что», но не «как» выполнять в процессе программной инженерии. Программная инженерия в значительной степени — социальное явление, определяемое внутренней организационной культурой предприятия. Предприятие может выбрать базовую **модель жизненного цикла**, но специфические особенности жизненного цикла и то, как работа будет сделана, уникальны для каждой организации и могут даже значительно отличаться от проекта к проекту. Это согласуется с наблюдениями, сделанными в разделе 1.1.3: программный продукт не изготавливается, он реализуется. Процесс создания и эксплуатации ПО — не эксперимент, который может быть повторен много раз с той же самой степенью успеха.

Далее приведен краткий список причин, почему специфические особенности жизненного цикла должны быть приспособлены к организационной культуре и почему они отличаются от проекта к проекту:

- Опыт программной инженерии, навыки и знания коллектива разработчиков (если их недостаточно, должно быть включено в процесс разработки и время в соответствии с «кривой обучения»).
- Деловой опыт и знания (это намного более неприятно, чем предыдущий момент, потому что деловой опыт и знания легко не приобретаются).
- Вид предметной области (необходимы различные процессы, чтобы совершенствовать бухгалтерский учет или систему контроля электростанции).
- Изменения деловой атмосферы (изменения внешнеполитических, экономических, социальных, технологических и конкурентных факторов).

- Внутренние деловые изменения (изменения в управлении, условиях работы, финансовое здоровье предприятия и т. д.).
- Размер проекта (большой проект требует различных процессов, начиная с небольших; очень маленький проект может даже вообще не нуждаться в каких-либо процессах, поскольку разработчики могут сотрудничать и обмениваться информацией неофициально).

При условии вышеупомянутого обсуждения подходы к жизненному циклу ПО могут быть грубо разделены на две главные категории:

- «водопад с обратной связью»;
- итеративный пошаговый.

1.3.1. Жизненный цикл «водопад с обратной связью»

Модель водопада — традиционный жизненный цикл, реализованный и популяризируемый в 1970-х годах. Об этой модели было сказано, что она успешно использовалась во многих больших проектах прошлого. Большинство этих проектов было связано с пакетными (то есть не диалоговыми) системами, реализованными на языке КОБОЛ. Сегодня жизненный цикл «водопад» используется менее часто.

Используя стадии жизненного цикла, принятые в этой главе, жизненный цикл «водопад с обратной связью» может быть представлен так, как изображено на рис. 1.6. Это линейная последовательность стадий, из которых предыдущая стадия должна быть закончена прежде, чем может начаться следующая. Завершение каждой стадии отмечается *подписанием* документа для этой стадии проекта. Обратные связи (обратные стрелки на рис. 1.6) между стадиями возможны и даже целесообразны. Обратная связь показывает не документированное, но необходимое изменение в более поздней стадии, которое должно быть завершено соответствующим изменением в предыдущей стадии. Такой возврат может (хотя это бывает редко) продолжаться до начальной стадии — анализа требований.

Имеется много вариантов модели жизненного цикла «водопад». Один такой вариант, изображенный на рис. 1.7, допускает наложение стадий, то есть следующая стадия может начинаться прежде, чем предыдущая полностью за-

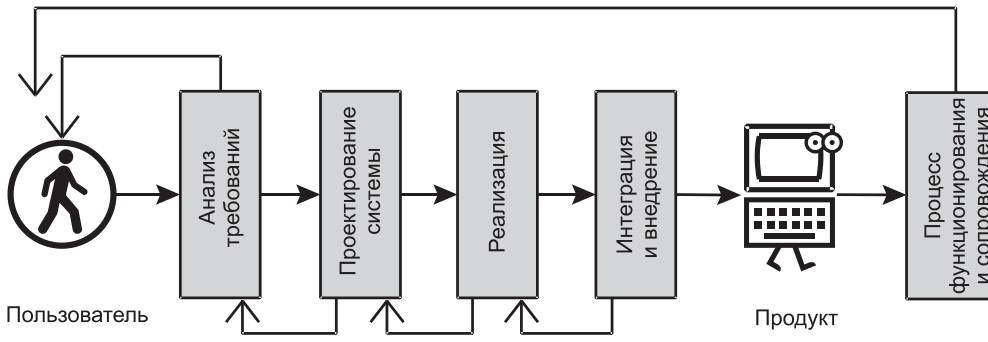


Рис. 1.6. Жизненный цикл «водопад с обратной связью»

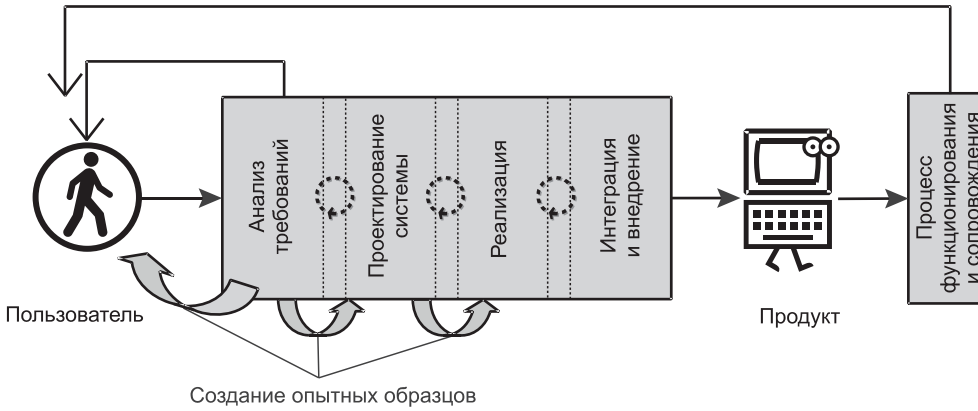


Рис. 1.7. Жизненный цикл «водопад с обратной связью» с наложениями и опытными образцами

кончится, будет задокументирована и подписана. Это обозначено на рис. 1.7 окружностями со стрелками между стадиями. Другой популярный вариант, иногда называемый **моделью с опытными образцами**, учитывает создание опытных образцов ПО на стадиях, предшествующих стадии реализации. Это также изображено на рис. 1.7.

Критический момент относительно методов водопада — то, что они являются *монолитными*: они применяются при разработке по принципу «выполнять одновременно» ко всей системе в целом и задают единственную дату поставки системы. Пользователь подключается только на ранних стадиях анализа требований и заканчивает техническим заданием. Позже в течение жизненного цикла пользователь находится в полном неведении, пока изделие не может быть протестировано им до его внедрения. Поскольку *время задержки* между началом проекта и поставкой ПО может быть существенным (месяцы или даже годы), доверие между пользователями и разработчиками может быть утеряно, и разработчикам придется все сложнее защищать проект и оправдывать израсходованные ресурсы.

Использование *наложения* стадий может привести к другому недостатку жизненного цикла, а именно к остановкам в некоторых частях проекта, потому что разработчики различных коллективов ждут другие коллективы, чтобы можно было завершить зависимые задачи. Наложение также приводит к большим обратным связям между соседними стадиями.

Использование опытных образцов в любом жизненном цикле является положительным, но оно дает особые преимущества для модели водопада, добавляя некоторую гибкость в ее монолитную структуру и смягчая риск поставки изделия, не отвечающего требованиям пользователя. **Опытный образец ПО** — частный, «сляпанный наспех» пример решения проблемы. Из этого решения могут быть получены последовательные формы программного продукта.

В программной инженерии опытные образцы использовались с большим успехом, выявляя и делая понятными требования пользователя к изделию. После того как опытный образец один раз будет использован в этом качестве,

возникает вопрос, что делать с его ПО. Одна из возможностей состоит в том, чтобы забросить его подальше, как только цель подтверждения правильности требований будет достигнута. Оправданием этого действия является то, что сохранение опытного образца может привести к переработке «сляпанного наспех» решения в конечный продукт. В конце концов, люди не ездят на опытных образцах автомобилей!

Однако реализация ПО — это не производство автомобилей. С помощью средств разработки динамического ПО на основе компонентов возможно из опытного образца создать хороший конечный продукт без каких-либо следов «сляпанного наспех» решения. На рис. 1.7 это отражено направленными вперед стрелками. Опытный образец, созданный по требованиям анализа, может служить для улучшения проекта системы, а улучшенный опытный образец проекта может быть преобразован в конечный продукт.

В табл. 1.1 приведены основные характеристики жизненных циклов «водопада», объединенные в группы преимуществ и недостатков [92, 34]. Иногда характеристика является одновременно и преимуществом, и недостатком. Это отражено комментариями, помещенными в оба столбца таблицы.

Таблица 1.1. Жизненный цикл «водопад с обратной связью»

Преимущества	Недостатки
• Предписывает дисциплинированный подход к разработке ПО. Определяет ясные вехи в стадиях жизненного цикла, облегчая таким образом руководство проектом	• Критерии завершения анализа требований и проектирования системы часто не определены или неопределенны. Сложно определить, когда останавливаться. Опасность превышения крайних сроков
	• Монолитный подход, обращаясь ко всей системе целиком, может потребовать много времени до получения конечного продукта. Это может быть недопустимо для современного предприятия, требующего короткого срока «возвращения инвестиций»
	• Никаких возможностей для абстракции. Никакой возможности «разделять и властвовать» в предметной области проблемы, чтобы справиться со сложностью системы
• Производит полную документацию для системы	• Документация может давать ложное представление относительно прогресса в проектировании. Ее сухие, формальные утверждения могут быть легко извращены. Имеется также риск бюрократизировать работу
• Требует завершения проектной документации перед переходом в последующие стадии, разъясняет юридические позиции коллективов разработчиков	• Замораживание результатов каждой стадии идет вразрез с программной инженерией как социальным процессом, в котором требования могут меняться, нравится ли нам это или нет
• Требует тщательного планирования проекта	• Планирование проекта проводится на ранних стадиях жизненного цикла, когда имеется только ограниченное понимание проекта. Риск неверной оценки требуемых ресурсов

Источник: Основано на информации Гецци и др. [34].

1.3.2. Итеративный пошаговый жизненный цикл

Итерация в разработке ПО (в противоположность итерации в программе) — повторение некоторого процесса с целью улучшить программный продукт. Каждый жизненный цикл имеет некоторые элементы итеративного подхода. Например, обратные связи и наложения в модели водопада представляют своего рода итерацию между отдельными фазами, стадиями или действиями. Однако модель водопада не может считаться итеративной, потому что итерация означает движение от одной версии изделия к его следующей версии. Подход модели водопада монолитен, с формированием только одной заключительной версии изделия.

Итеративный жизненный цикл предполагает **шаги** — улучшенные или расширенные версии изделия в конце каждой итерации. По этой причине итеративная модель жизненного цикла иногда называется эволюционной или пошаговой.

Итеративный жизненный цикл также предполагает наличие **конструкций** — исполняемого кода, полученного в итерации. Конструкция — вертикальный срез системы. Это не подсистема. Область действия конструкции — вся система, но с некоторыми уменьшенными функциональными возможностями, с упрощенными пользовательскими интерфейсами, с ограниченной многопользовательской поддержкой, меньшим объемом работы и другими подобными ограничениями. Конструкция — это нечто, что может демонстрироваться пользователю как версия системы, полученная в процессе продвижения к конечному продукту. Каждая конструкция фактически является новым шагом по отношению к предыдущей. В этом смысле понятия конструкции и шага не различимы.

Итеративный жизненный цикл предполагает *короткие итерации* между шагами, в недели или дни, но никак не в месяцы. Это позволяет осуществлять непрерывное планирование и надежное управление. Работа, выполненная на предыдущих итерациях, может быть измерена и может дать ценную информацию для планирования выполнения проекта. Наличие работоспособных двоичных кодов, подлежащих сдаче в конце каждой итерации, позволяет надежно управлять проектом.

Рис. 1.8 демонстрирует, что каждая итерация — маленький «водопад» типичных стадий жизненного цикла. Различия лишь в масштабе, как сказано выше. Теперь пользователь может часто и полностью использовать основной цикл стадии функционирования и сопровождения. Уроки предыдущей итерации немедленно используются в следующей итерации. Пользователь, вооруженный опытом использования предыдущей конструкции, может быть очень полезен в уточнении требований для следующей итерации. Текущий проект конструкции — отправная точка для проекта системы в следующей итерации. Внедрение итерации 2 продукта является началом итерации 3 и т. д.

Классическая модель итеративного жизненного цикла — спиральная модель [10]. Современным представителем итеративного жизненного цикла является IBM Rational Unified Process® (RUP®) [91], который создан из Rational Unified Process (RUP) [53]. Более современные представители итеративной модели жизненного цикла — Model Driven Architecture (MDA®) [51, 67] и agile development — быстрая разработка [1, 66].

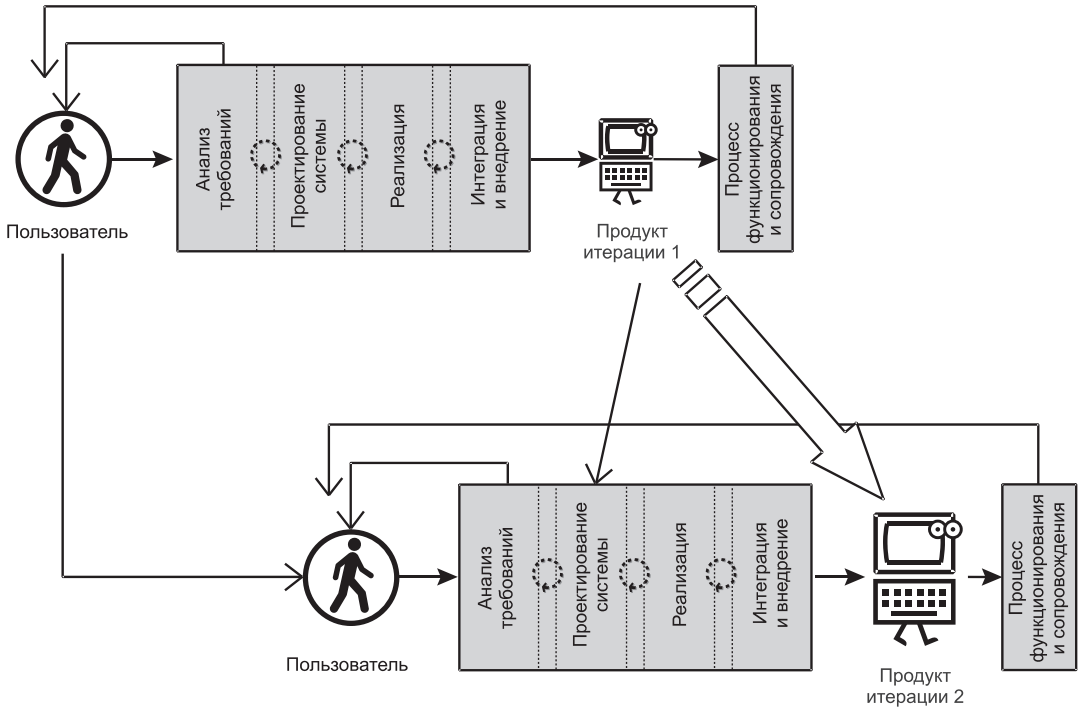


Рис. 1.8. Итеративный пошаговый жизненный цикл

Спиральная модель

Как справедливо замечено Гецци и др. [34], **спиральная модель** [10] в действительности является *метамоделью*, в которой могут содержаться все другие модели жизненного цикла. Модель состоит из четырех секторов диаграммы в декартовых координатах (рис. 1.9). Сектора следующие: планирование, анализ рисков, инженерия и оценка проекта клиентом. Первая петля спирали начинается в секторе *планирования* и связана с начальным сбором требований и планированием проекта. Затем проект входит в сектор *анализа рисков*, где проводится анализ стоимости/выгоды и угроз/благоприятных случаев, чтобы принять решение «да-нет» относительно того, следует ли входить в сектор *инженерии* (или отказаться от проекта как слишком опасного). Сектор инженерии — это то, где происходит разработка ПО. Результат этой разработки (конструкция, опытный образец или даже конечный продукт) подвергается *оценке клиентом*, после чего начинается вторая петля спирали.

Фактически спиральная модель имеет отношение к разработке ПО только в одном из этих четырех секторов — секторе инженерии. Акцент на повторном планировании проекта и оценке проекта клиентом дает всему этому явно итеративный характер. Анализ рисков уникален в спиральной модели. Частые исследования рисков позволяют провести раннюю идентификацию любых появляющихся рисков в проекте. Риски могут быть внутренние (находящиеся под управлением организации) и внешние (риски, которыми организа-

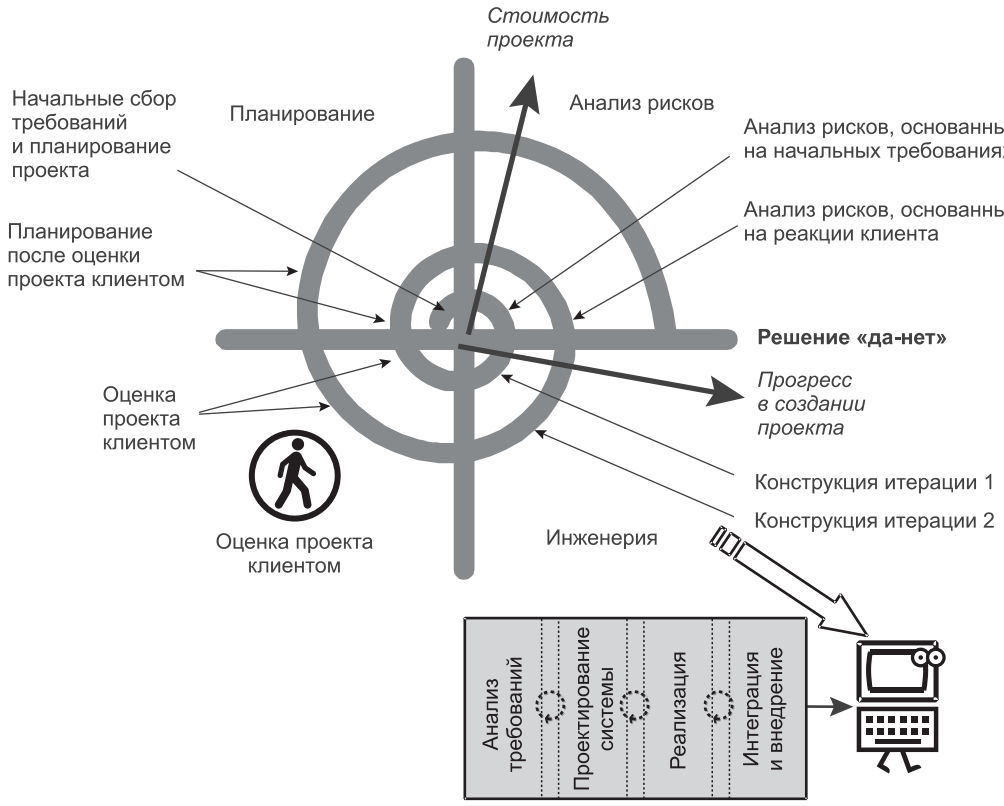


Рис. 1.9. Спиральная модель жизненного цикла

ция не может управлять). В любом случае задача анализа рисков состоит в том, чтобы смотреть в будущее, и если ясно, что риски неустранимы, проектирование должно быть прекращено безотносительно к затратам, уже израсходованным.

Каждый фрагмент петли в пределах сектора инженерии может быть итерацией, заканчивающейся созданием конструкции. Любой последовательный фрагмент петли в направлении наружу представляет собой шаг. Возможны и другие интерпретации фрагментов петли инженерии. Например, вся петля спирали может быть связана с анализом требований. В таком случае фрагмент петли инженерии может быть связан с моделированием требований и созданием раннего опытного образца, чтобы выявить требования. С другой стороны, спиральная модель может содержать монолитную модель водопада. В таком случае, будет только одна петля спирали.

Спиральная модель — скорее модель ссылок или метамодель для других моделей. Привлекательная и реалистическая, насколько это возможно, она не может быть перенесена на конкретный задокументированный жизненный цикл, который организации используют при разработке ПО. Спиральная модель — «способ мышления» относительно принципов разработки ПО.

Rational Unified Process (RUP)

IBM Rational Unified Process® — рациональный унифицированный процесс (RUP®) [91] — больше, чем модель жизненного цикла. Это также и среда поддержки (называемая RUP-платформой), помогающая разработчикам в использовании и приспособлении к RUP-жизненному циклу. Эта поддержка реализуется в форме HTML и другой документации, обеспечивающей диалоговую помощь, шаблоны и руководство. Среда поддержки RUP составляет неотъемлемую часть набора инструментальных средств программной инженерии производства IBM (ранее Rational), но она может использоваться и как модель жизненного цикла для любой разработки ПО.

Степень признания RUP страдала от неясной структуры RUP-процесса, которая была представлена как двуразмерная. *Горизонтальное измерение* представляет динамический аспект жизненного цикла — время, протекающее в процессе. Это измерение было разделено на четыре разворачивающихся аспекта жизненного цикла: начало, разработка, конструирование и переход. *Вертикальное измерение* представляет статический аспект жизненного цикла — дисциплины разработки ПО. Это измерение было разделено на действия жизненного цикла: бизнес-моделирование, требования, анализ и проектирование, реализация, тестирование, внедрение, а также вспомогательные действия по управлению конфигурацией и изменениями, управлению проектированием и окружающей средой.

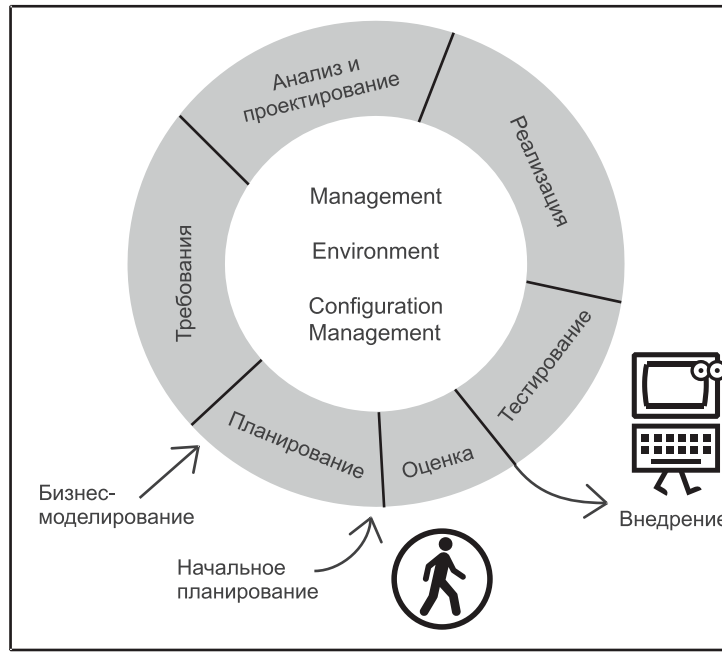


Рис. 1.10. Процесс RUP

Источник: взято с рисунка RUP и перепечатано по разрешению с http://www.rational.com/products/rup/, © Copyright 2003 by International Business Machines Corporation. All Rights Reserved

При первом обращении процесс, представленный как комбинация двух RUP-измерений, недостаточно понятен. Каково различие между конструированием и реализацией, между началом и бизнес-моделированием или между переходом и внедрением? На эти вопросы трудно ответить. Чтобы устранить эти проблемы (так кажется), была предложена упрощенная визуализация RUP®. Рис. 1.10 — это упрощенное представление [91].

Кроме некоторых небольших изменений в стадиях жизненного цикла, RUP-процесс очень хорошо совпадает с базовым итеративным жизненным циклом, представленным на рис. 1.8. Основное различие — явная стадия тестирования после стадии реализации. Как уже неоднократно упоминалось, стадии жизненного цикла, принятые в этой книге, рассматривают тестирование как всеобъемлющую деятельность, которую нужно применять к каждой из них, а не только к реализации. В RUP тестирование приводит к внедрению итерации (шага).

Следуя спиральной модели, RUP старается быть явным относительно управления рисками. Один аспект управления рисками в RUP — явная стадия оценки. Хотя более важно то, и это совпадает со спиральной моделью, что RUP поддерживает выбор «да-нет» в конце каждой стадии жизненного цикла.

Model Driven Architecture (MDA)

Model Driven Architecture® — архитектура, управляемая моделями (MDA®) [51, 67] — структура жизненного цикла от Object Management Group — группы управления объектами (OMG). MDA использует UML на следующей характерной стадии — выполнимых спецификаций. Идея **выполнимых спецификаций**, то есть превращения моделей спецификаций в выполнимый код, не нова, но MDA пользуется преимуществом существующих стандартов и современной технологии, чтобы реализовать эту идею.

MDA — современный представитель *модели преобразования* [34], которая в свою очередь ведет свое происхождение от **formal systems development** — формальной разработки систем [96]. Модель преобразования рассматривает разработку систем как последовательность преобразований от формальных спецификаций задачи через более детальные (менее абстрактные) стадии проектирования к выполняемой программе. Каждый шаг преобразования тщательно проверяется, чтобы гарантировать, что каждый результат — правильное представление исходных данных.

Эта модель предполагает широкую автоматизацию преобразований, однако признает, что некоторые преобразования могут быть выполнены только вручную. Соответственно, предполагалось использование машинно-считываемых представлений моделей, созданных и размещенных в среде автоматизированной программной инженерии (computer assisted software engineering — CASE). Возможность машинного считывания обеспечивает итеративную природу модели преобразования. В отличие от жизненного цикла водопада модель преобразования поддерживает эволюцию рассматриваемых моделей через многократные итерации.

Жизненный цикл MDA изображен на рис. 1.11. MDA использует две спецификации: неформальную Requirements (требования) и более формальную Analysis (анализ). Это разделение позволяет нам исключать Requirements из

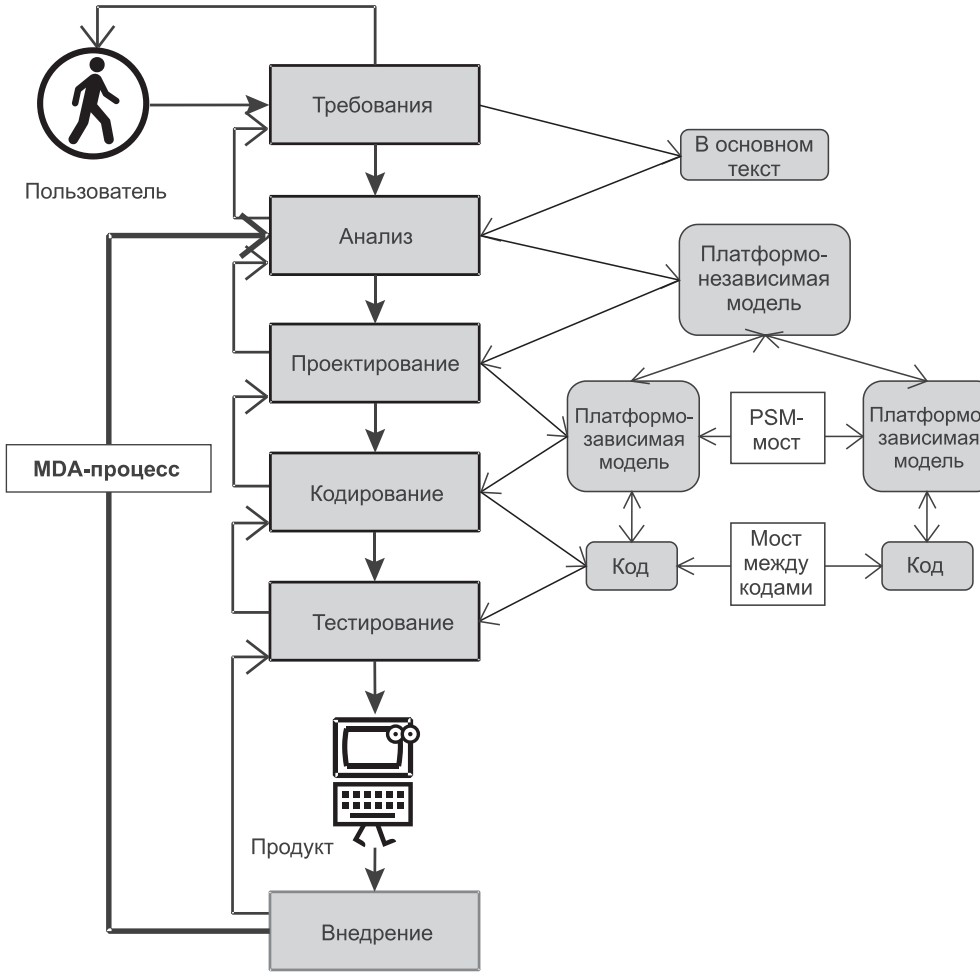


Рис. 1.11. Жизненный цикл MDA

преобразований MDA-процесса. Остающиеся элементы (модели) процесса позволяют выполнять преобразования в машинно-считываемой форме.

Результатом использования Analysis является платформо-независимая модель (Platform Independent Model — PIM) — высоко абстрактная и не зависимая от каких-либо программных/аппаратных ограничений. Результатом использования Design является платформо-зависимая модель (Platform Specific Model — PSM) — менее абстрактная и ограниченная платформой программной/аппаратной реализации. Для каждой платформы формируется своя PSM. Если система в процессе разработки охватывает несколько платформ, то формируются дополнительные PSM, объединенные связующими MDA-*мостами*. Разные PSM формируют разные коды, которые также связываются мостами.

MDA обещают расширить в **технологию компонентов** и целую область конструирования систем из многократно используемых блоков — моделей и

программ (такой подход иногда называется *жизненным циклом на основе компонентов* [81]). Структура находится вне выполнимых UML-моделей и охватывает диапазон OMG-определяемых сервисов типа складских сервисов, перманентных объектов, транзакций, обработчиков событий и сервисов, обеспечивающих безопасность. Цель состоит в том, чтобы создать многократно используемые и поддающиеся преобразованию модели для ряда основных отраслей типа телесвязи или больниц. Время скажет, окажется ли MDA практической. Для этого MDA должна быть способна применяться к большим и сложным системам.

Быстрая разработка ПО с короткими итерациями

Процесс **быстрой разработки ПО**, предложенный в 2001 г. некоммерческой организаций энтузиастов Agile Alliance, является смелым новым подходом к производству ПО. В Manifesto of Agile Alliance [1] дух быстрой разработки представлен четырьмя рекомендациями:

1. Индивидуальные характеристики и взаимодействия процессов и инструментальных средств.
2. Рабочее ПО с полной документацией.
3. Сотрудничество с клиентами после заключения контракта.
4. Определение, что нужно изменить после планирования.

Быстрая разработка подчеркивает, что производство ПО — творческая деятельность, которая зависит от сотрудничества людей и коллективов гораздо больше, чем от различных процессов, использования инструментальных средств, документации, планирования и других формальных операций. Быстрая разработка подтверждает принцип, что «большое ПО делают большие коллективы людей», все остальное вторично. «Люди» включают всех участников создания проекта — разработчиков и клиентов. В отличие от других процессов создания ПО, при быстрой разработке клиенты (пользователи и собственники системы) тесно работают с коллективом разработчиков на протяжении всего жизненного цикла, а не только в его начале. Постоянная обратная связь клиента позволяет легче подписать формальный контракт на все изделие. Интенсивное сотрудничество облегчает также получение той части документации, которая обеспечивает передачу знаний.

Несмотря на все эти «революционные» суждения, быстрая разработка хорошо выглядит среди других итеративных жизненных циклов. Как показано на рис. 1.12, быстрый жизненный цикл не может использовать терминологию обычных стадий жизненного цикла, однако на самом деле его терминология соответствует обычному циклу анализа, проектирования, реализации и внедрения.

«Обычный» анализ требований заменен в быстрой разработке *историями пользователей*, где клиент отмечает особенности, которые, по его мнению, система должна поддерживать. Коллектив разработчиков оценивает, сколько потребуется времени для реализации каждой истории, и сколько реализация каждой истории будет стоить. После этого клиент может выбрать те истории, которые будут реализованы на первой и последующих итерациях.

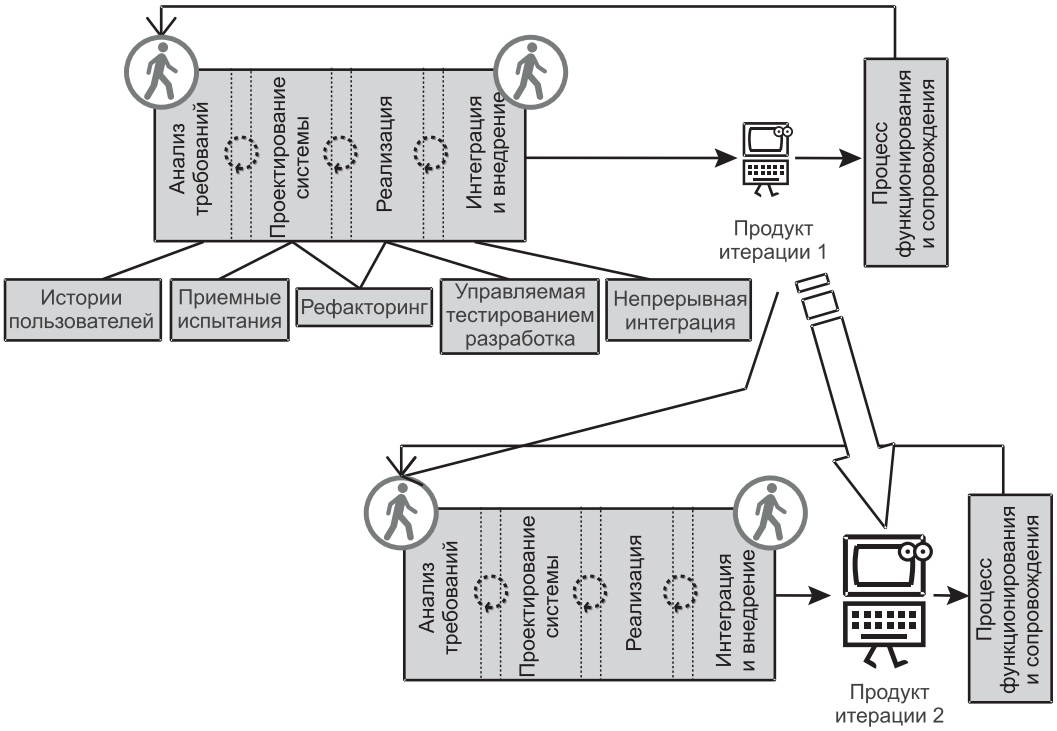


Рис. 1.12. Быстрая разработка ПО

«Обычные» виды проектирования систем и их реализация заменены в быстрой разработке комбинацией приемочных испытаний, рефакторинга и управляемой тестированием разработки. *Приемочные испытания* — это специальные программы, через которые разрабатываемая прикладная программа должна пройти, чтобы быть принятой клиентами. Этот процесс называется «**управляемой тестированием разработкой**» (Test-Driven Development — TDD) или *программированием намерений* (Intentional Programming — IP) — разработчик программирует свои намерения по приемочным испытаниям до того, как использует их. Этому подходу сопутствует частое использование **рефакторинга** (refactorings) — усовершенствования в структуре системы без изменения ее поведения.

Быстрая разработка поддерживает и другие концепции типа **парного программирования** и *коллективной собственности*. Все программирование выполняется парами программистов — двумя программистами, работающими вместе на одной рабочей станции. Один программист пишет код, а другой наблюдает за процессом и задает вопросы. Роли меняются всякий раз, когда один из программистов хочет «поставить точку с помощью клавиатуры». Пары программистов также меняются местами, по крайней мере, один раз в день. Результатом является *коллективная собственность*. Никто индивидуально не владеет кодом. Считается, что высокий дух коллективизма и желание выдать продукт перевесят любые требования индивидуальной ответственности.

«Обычные» интеграция и внедрение заменены в быстрой разработке *непрерывной интеграцией* и *короткими циклами*. Пары программистов могут экспортировать свой код и по своему желанию объединять его с остальной частью. Более того, одну и ту же часть кода может импортировать и работать над ней более одной пары программистов. Это может привести к конфликту, когда коллектив, который хочет экспортировать и сдать свой код, обнаруживает, что другой коллектив сделал ранее несовместимый код. Такие конфликты между участвующими в разработке коллективами должны быть урегулированы.

Быстрая разработка не означает плохое планирование. Фактически даты внедрения тщательно планируются. Каждая итерация обычно планируется так, чтобы закончить работу за короткий цикл продолжительностью в две недели. Продукт в конце двухнедельного цикла — пробный вариант для оценки клиентом. Основная поставка продукта в производство является результатом приблизительно шести двухнедельных циклов.

Быстрая разработка отличается больше в методах, чем в подходе к итеративной разработке. Ее главный представитель — **eXtreme Programming** (XP) [6]. Как и с MDA-подходом, будущее покажет, сможет ли быстрая разработка применяться к большим и сложным системам. Главная опасность, стоящая перед сторонниками быстрого жизненного цикла — риск окончания работы с неудачно *созданной и принятой моделью* [92], в которой ПО слеплено без учета заданных требований к проекту.

Резюме

1. *Программная инженерия* связана с разработкой больших систем ПО. Программная инженерия — обычно центральная деятельность более широкого понятия — *инженерии систем*.
2. *Пятиугольник программной инженерии* состоит из жизненного цикла разработки ПО, языка моделирования ПО, инструментальных средств программной инженерии, планирования проектирования ПО и управления процессом создания и эксплуатации ПО.
3. Стадии процесса разработки ПО упомянуты как стадии *жизненного цикла* ПО. Стадии жизненного цикла, принятые в книге, — анализ требований, проектирование системы, реализация, интеграция и внедрение, а также процесс функционирования и сопровождения.
4. Система ПО — просто часть намного большей *информационной системы предприятия*.
5. Процесс создания и эксплуатации ПО — часть *бизнес-процесса*. Результат процесса создания и эксплуатации ПО — ПО. Результат бизнес-процесса — бизнес.
6. Система ПО может обслуживать любой из трех уровней управления: оперативный, тактический или стратегический.
7. Нематериальный и изменчивый характер ПО — всего лишь два фактора, которые отличают программную инженерию от *традиционной инженерии*.
8. Программная инженерия — больше, чем *программирование*. Программная инженерия применяется к сложным проблемам, которые не могут быть решены одним программированием.

9. Программная инженерия — вид *моделирования*. Все продукты программной инженерии, включая программы, являются моделями действительности. Моделирование использует *абстракцию*, чтобы представлять концепции с различными уровнями детализации.
10. Системы ПО *сложны*. Сложность современного ПО заключается в «проводах» — в связях и коммуникационных путях между компонентами.
11. *Анализ требований* — действия по определению и составлению списка требований пользователя. Соответственно, анализ требований делится на *определение требований* и *техническое задание*. *Разработка требований* включает все наиболее строгие и формальные задачи поддержки анализа требований.
12. *Проектирование системы* следует за анализом требований, и это — моделирование, которое учитывает платформу, на которой должна быть реализована система. Имеются два различных аспекта проектирования системы: *структурное проектирование* и *детальное проектирование*. Главная цель структурного проектирования состоит в том, чтобы создать систему, которая является приемлемой — понятной, ремонтопригодной и расширяемой. Детальное проектирование должно соответствовать структурному проектированию.
13. *Реализация* главным образом является программированием, но она включает и другие технические действия типа инженерии компонентов и прямого и обратного проектирования. Отладка и тестирование — неотъемлемая часть реализации.
14. *Интеграция* собирает приложение из набора компонентов, предварительно реализованных и проверенных. *Внедрение* — передача системы клиентам для использования в производстве. *Тестирование интегрированной системы* — важная часть успешной интеграции, а *приемочные испытания* должны быть сделаны до внедрения.
15. *Процесс функционирования* является важной стадией жизненного цикла, когда программный продукт используется в ежедневной работе, а использование предыдущей системы (ручной или автоматизированной) постепенно сокращается. *Постепенное сокращение* — обычно организованный процесс. Процесс функционирования совпадает с началом *сопровождения* изделия. Сопровождение может быть корректирующим, адаптивным или совершенствующим.
16. Имеются различные *модели жизненного цикла*, которые могут быть приняты для разработки ПО. Они грубо разделены на модели водопада с обратной связью и итеративные пошаговые модели.
17. *Модели водопада* характеризуются линейной последовательностью стадий, в которых предыдущая стадия должна быть закончена прежде, чем может начаться следующая. Модели водопада не подходят для современного производства ПО.
18. Имеются четыре главных представителя *итеративных моделей*: спиральная, Rational Unified Process (RUP), Model Driven Architecture (MDA) и модель быстрой разработки.

19. *Спиральная модель* — на самом деле базовая метамодель, которая охватывает все итеративные модели. Модель состоит из четырех секторов жизненного цикла: планирование, анализ риска, инженерия и оценка проекта клиентом. Анализ рисков — наиболее характерная особенность спиральной модели.
20. *RUP* — больше, чем модель жизненного цикла. Это также и среда поддержки (называемая RUP-платформой), чтобы помочь разработчикам в использовании и приспособлении к жизненному циклу RUP. Подобно спиральной модели RUP использует управление рисками.
21. *MDA*-модель основана на идее выполнимых спецификаций. Это — современный представитель модели преобразования, которая в свою очередь происходит от формальной разработки систем. Технология компонентов — сердце MDA-модели.
22. *Быстрая разработка* подчеркивает, что производство ПО — творческая деятельность, которая зависит от сотрудничества людей и коллективов гораздо больше, чем от различных процессов, использования инструментальных средств, документации, планирования и других формальных операций.

Ключевые термины

Обзорные вопросы

1. Объясните, как программная инженерия и инженерия систем соотносятся друг с другом. Является ли это отношением включения или пересечения? Может быть, эти две концепции вообще не имеют друг к другу никакого отношения?
2. Каковы пять главных аспектов программной инженерии? Можете ли вы сказать, что любой случай программной инженерии обязательно охватывает все эти аспекты?
3. Какие факторы определяют, что система является унаследованной? Может ли унаследованная система быть преобразованной в современную систему? Как это может быть сделано, если вообще возможно?
4. Что мы подразумеваем, когда говорим, что информационные системы являются социальными системами? Может ли быть система социальной? Рассмотрите следующее объяснение термина «социальный» — «касающийся действий, когда вы встречаетесь и проводите время с другими людьми, и которые происходят в то время, когда вы не работаете» [18].
5. В науке управления существует строгое различие между эффективностью и результативностью систем и людей. В обычном использовании, термин «эффективность» означает «когда кто-то или что-то хорошо использует

время и энергию без каких-либо пустых затрат» [18]. Термин «результативность» означает «насколько успешно это ... в достижении результатов, которые вы хотите получить» [18]. Как эти два термина применимы к программной инженерии? Является ли программная инженерия эффективной или результативной или и тем, и другим? Объясните. Приведите примеры.

6. Каковы основные различия между БД и хранилищем данных? Как эти две концепции связаны с уровнями управления в организации?
7. Что такое приемлемая система? Может ли быть унаследованная система приемлемой? Объясните.
8. Что мы подразумеваем под прямым и обратным проектированием? Как это касается программирования?
9. Программная инженерия связана с моделированием систем. Что моделируется в процессе программной инженерии? Как это соотносится с понятием абстракции? Имеет ли смысл говорить о моделировании программ?
10. Как вы понимаете различие между функциональным и объектно-ориентированным подходами к разработке системы?
11. Какой фактор наиболее важен в определении сложности современной объектно-ориентированной системы? Какова основная технология управления сложностью и ее уменьшением? Объясните.
12. Объясните отношение между анализом требований и техническими требованиями.
13. Каково различие между определением требований и техническим заданием? Объясните.
14. Проведите линию раздела между анализом требований и проектированием системы. Когда анализ становится проектированием?
15. Как детальное проектирование и структурное проектирование относятся друг к другу?
16. Каковы главные подходы в технологии тестирования?
17. Объясните использование заглушек и драйверов в тестировании интеграции.
18. В современной программной инженерии модель водопада для жизненного цикла заменена итеративными моделями. Имеются ли какие-либо аспекты модели водопада, отсутствующие или не выполнимые в итеративных моделях, которые принесли бы пользу итеративному подходу? Обсудите.
19. Что такое управление рисками? Какая из моделей жизненного цикла наиболее явно использует управление рисками? Объясните.
20. Что такое выполнимые спецификации? Какая модель жизненного цикла использует выполнимые спецификации как свой основной момент? Объясните.
21. Каковы наиболее необычные аспекты быстрой разработки (необычные по сравнению с другими итеративными подходами)? Объясните.

Глава 2

Язык моделирования программного обеспечения

Язык является средством связи между людьми. ПО создано людьми и для людей. Поэтому нужно, чтобы модели ПО были коммуникабельны. Моделирование ПО требует наличия языка как средства выражения процессов разработки ПО и созданных продуктов. Язык должен быть универсальным (должен существовать его стандарт) и понятным большому сообществу людей. Он должен быть богат семантически, чтобы передать требуемый смысл минимальным количеством синтаксических конструкций. Наконец, язык должен обладать мощными абстрактными концепциями, чтобы выразить идеи на различных уровнях абстракции без потребности использовать другие языки, лучше удовлетворяющие тому или иному уровню абстракции.

Как отмечено в разделе 1.1.5, программная инженерия сродни моделированию. Результат программной инженерии — программа, являющаяся просто выполняемой моделью. Следовательно, язык программирования является языком моделирования, хотя и на очень низком уровне абстракции, представляя биты и байты вычислительной системы. В идеале один язык моделирования ПО должен служить объединенным целям анализа, проектирования и реализации. Сегодня это фактически является предпосылкой использования стандартного языка моделирования **Unified Modeling Language** (UML) — унифицированного языка моделирования [108].

Вообще-то говоря, ни UML, ни любой другой язык моделирования не достиг еще статуса языка для полной разработки жизненного цикла. В частности, запутанность программирования мешала искушению реализовать суждение «один язык годен на все случаи жизни». Программисты работают с различными языками, которые лучше удовлетворяют их по той или иной причине. Это означает, что, когда начинается программирование, UML-модели должны быть переведены на язык программирования. Кроме того, когда выполняется детальное проектирование для различных предметных областей и различных технологий, UML имеет недостаточную мощь и разнообразие, чтобы обслужить все цели проектирования. **UML-профили**, типа UML для моделирования сети или UML для бизнес-моделирования [107], несколько смягчают эту проблему, но есть еще много нерешенных вопросов.[1]

«Одно изображение стоит тысячи слов». Языки моделирования ПО используют эту старую истину. Эти языки *визуальны*, но с добавлением *текста*. Иногда графическая модель должна быть далее разъяснена, чтобы избе-

[1] UML-профиль — подмножество UML-шаблонов, используемых для определенных целей (*Прим. перев.*)

жать неверного толкования. В других случаях одна графическая модель не дает целую картину, и в тексте обеспечивается дополнительная информация. В некоторых случаях текст доминирует над визуальным представлением. Интересно, что определение требований в начале жизненного цикла и программирование в конце его преобладающе состоят из текста. Средние стадии анализа и проектирования главным образом визуальны.

Следовательно, спецификации ПО состоят из большого числа *визуальных моделей*, снабженных текстом, определяющих различные стадии разработки продукта и обеспечивающих различные (часто перекрывающиеся) точки зрения на продукты ПО. Грубая классификация моделей различает модели анализа, проектирования и реализации. Каждая **модель** состоит из одной или большего количества диаграмм и всевозможной дополнительной информации, размещенной в хранилище проекта.

Диаграмма — графическое представление модели, которое отображает символы ПО. Каждая диаграмма содержит различные аспекты модели или представляет модель на различных (более или менее детальных) уровнях абстракции. Основная классификация различает диаграммы, содержащие состояние системы, ее поведение или изменения состояния.

Диаграммы и текстовые описания моделей размещаются в хранилище проекта. **Хранилище** — главный компонент любого *инструмента визуального моделирования*. Это — БД продуктов проекта. Термин «база данных» (БД) подразумевает сохраняемое и интегрированное размещение данных, допускающее параллельный доступ к ним многих пользователей. Разработка ПО — совместное усилие многих разработчиков, и все они должны иметь параллельный доступ к моделям разработки. Большинство инструментальных средств визуального моделирования использует всю мощь технологии БД или даже применяет коммерческие системы управления БД (СУБД), чтобы реализовать их хранилища. Традиционно средства визуального моделирования называются CASE-инструментами (computer assisted software engineering — автоматизированная программная инженерия).

Эта глава посвящена UML. UML — аббревиатура средств и методов разработки ПО для разработчиков и пользователей. Для законченности и для контраста глава начинается с объяснения языка моделирования прошлого — языка структурного моделирования. Понятно, что эта глава не является полной детализацией языков моделирования; так, например, детальное описание UML в настоящее время состоит из 736 страниц [108]. Глава представляет только основной «словарь» и приводит примеры моделирования, используя небольшую предметную область — приложение «Фильм-актер» (MovieActor), используемое также и в других местах книги. Части со второй по четвертую объясняют и используют UML детально и строго.

2.1. Язык структурного моделирования

Структурное моделирование было популярно в конце 1970-х в результате широкого использования структурного программирования. **Структурное программирование** представляет собой программирование без использова-

ния операторов `goto` с циклами и операторами `if` как основными управляющими конструкциями и с нисходящим подходом к проектированию программ. Нисходящий подход структурного программирования использовался на стадиях анализа и проектирования, в результате чего были разработаны различные методы **структурного анализа и проектирования**. Ряд методов был известен под именами авторов книг, которые предложили эти методы — Де Марко, Йордана–Константина, Гейна–Сарсона, Джексона.

Структурное моделирование — нисходящий функционально-ориентированный подход к разработке ПО, который расчленяет систему на ряд взаимодействующих в общем процессе функций и известный как **функциональная декомпозиция**. Функциональная декомпозиция — пошаговый процесс, который основан на хорошем использовании абстракции. Модели системы представляются на ряде уровней абстракции, начинающихся с вершины — «птичьего глаза», дальнейшего обзора и заканчивающихся на простейших функциональных модулях, определенных в деталях и существенных для реализации. Структурное моделирование выражает монолитный и процедурный характер систем прошлого, например, стиля КОБОЛ.

Структурное моделирование задает диапазон технологий визуализации, наиболее популярными из которых были DFD (data flow diagrams — диаграммы потока данных), ERD (entity-relationship diagrams — диаграммы «сущность-отношение») и диаграммы структур. Первые две будут рассмотрены ниже.

2.1.1. Моделирование потока данных

Несомненно, **диаграммы потока данных** (data flow diagrams — DFD) являются одной из наиболее популярных технологий моделирования в истории программной инженерии. Сегодня они используются нечасто из-за того, что не соответствуют объектно-ориентированному подходу. Однако в некоторых обстоятельствах DFD остается приемлемой технологией. Важно отметить, что ранние версии UML пытались включать разновидность DFD для моделирования поведения систем. Эта попытка была отвергнута после критики со стороны объектно-ориентированных «пуристов».

Краеугольный камень DFD — функциональная декомпозиция. Рис. 2.1 иллюстрирует эту идею. Функциональная декомпозиция — метод постепенного определения функциональных процессов системы. Это — нисходящая операция, которая начинается с контекстной диаграммы и заканчивается спецификациями модулей.

Контекстная диаграмма представляет только один **процесс** — процесс, который соответствует разрабатываемой системе. Цель ее состоит в том, чтобы определить место системы по отношению к окружающей среде. Контекстная диаграмма задает границы системы, определяя ее входы и выходы по отношению к внешним сущностям. **Внешние сущности** — это организации, отделы, люди, другие системы и т. д., которые при разработке системы находятся вне нее и которые поставляют входные данные системе или ожидают от нее выходные данные, или делают и то, и другое. Потоки данных изображаются стрелками, которые так и называются **потоками данных**.

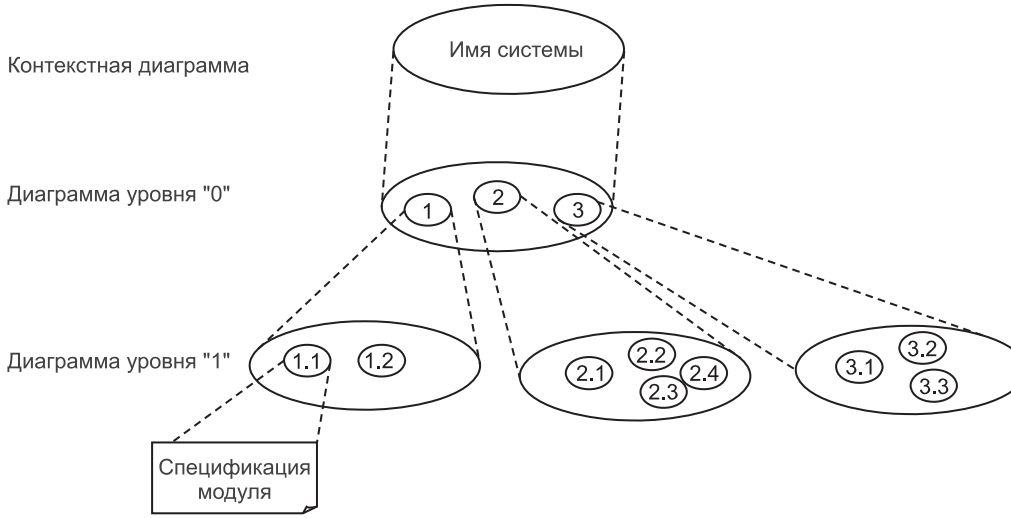


Рис. 2.1. Функциональная декомпозиция в DFD

Рис. 2.2 — пример контекстной диаграммы для системы (контекстный процесс), названной `Movies on the Web` (фильмы на Web-сайте). Система использует сеть кинотеатров для рекламы фильмов в Интернете, обеспечивая сведения о фильмах и характеристиках показов, чтобы кинозрители были информированы и могли в диалоге заказывать билеты. Имеются две внешние сущности, представленные на рис. 2.2: `Customers` (клиенты) и `Distributors` (дистрибьюторы). `Distributors` передает в систему значения `MovieDetails` (детальная информация о фильмах). Система выдает для `Customers` `ScreeningDetails` (характеристики показов), а `Customers` может размещать в системе `TicketOrder` (заказ билетов). Все они — потоки данных вне контекстного процесса. Вся другая обработка сделана внутри «пузыря», то есть внутри контекстного процесса.

Детализация контекста «пузыря» обеспечивается *диаграммой уровня «0»*, называемой также **диаграммой обзора**. Декомпозиция контекстной диаграммы в диаграмму уровня «0» показана на рис. 2.3. Система состоит из трех процессов: `CRUD MovieActor` («Фильм-актер»), `CRUD Screenings` (показы) и `Manage Ticketing` (управление распространением билетов) (`CRUD` — сокращение для Create, Read, Update, Delete — создать, читать, корректировать, удалить — четыре основных действия, выполняемые над данными в приложениях, работающих с БД). Внешние потоки данных назначаются процессам, но без дублирования внешних сущностей из контекстной

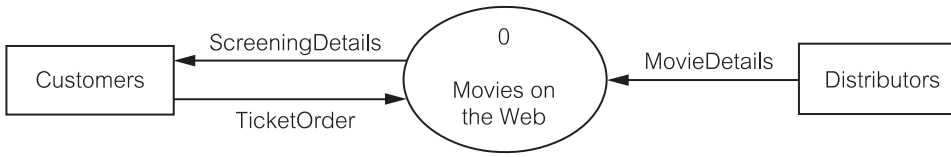


Рис. 2.2. Контекстная диаграмма

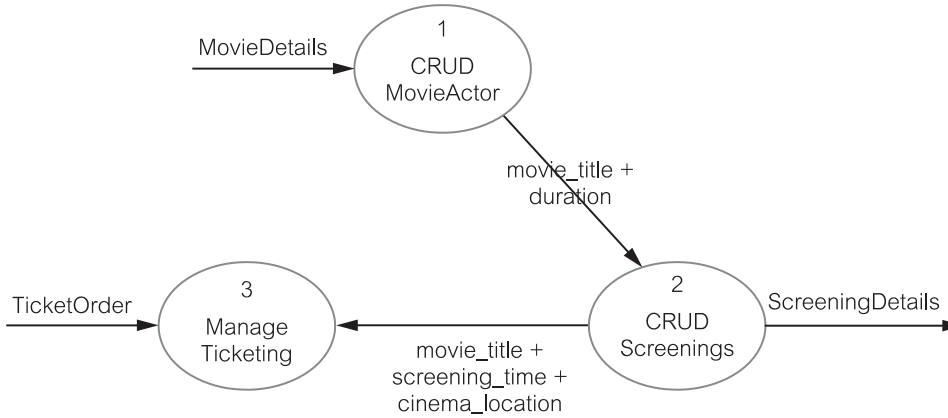


Рис. 2.3. DFD-диаграмма уровня «0»

диаграммы. Внутренние потоки данных теперь более определенные. Процесс 1 формирует `movie_title` (название фильма) и `duration` (продолжительность фильма) для процесса 2. Процесс 2 определяет продолжительности и места показов и передает эту информацию процессу 3. Процессы пронумерованы, но последовательность нумерации не имеет значения, кроме как для соединения процесса с его дальнейшей декомпозицией.

Рис. 2.4 представляет декомпозицию процесса 1 (`CRUD MovieActor`) из рис. 2.3. Диаграмма на рис. 2.4 — одна из трех возможных *диаграмм уровня «1»*. Потоки данных, входящие в процесс 1 и выходящие из него, рассчитываются для рис. 2.4 (это один из аспектов так называемого *баланса потоков*). Диаграмма уровня «1» на рис. 2.4 является последним элементом моделирования в DFD и отображает информационный склад, представленный здесь элементом `MovieActor Database` (БД «Фильм-актер»). **Информационный склад** — хранилище данных для потоков данных. К потоку данных, помещенному одним процессом в информационный склад, может обращаться другой процесс в другое время.

Теоретически функциональная декомпозиция может дать любое число уровней иерархии (дерева). Однако практически достаточны три или четыре уровня декомпозиции даже для весьма больших систем. Глубина декомпозиции может быть разной для разных процессов, идентифицированных на диаграмме уровня «0».

Процессы на нижнем уровне иерархии (узлы-листья дерева) определены в тексте далее. Это называется **спецификацией модуля** (рис. 2.1). Текст на самом деле представляет псевдокод, называемый **Structured English** (структурированный английский). Определение Structured English использует типичные концепции языка программирования типа оператора присваивания, условного оператора и конструкции цикла.

От спецификаций Structured English только один шаг к кодированию. DFD с их многоуровневыми абстрактными моделями занимают определенное место между анализом и проектированием. Они предназначены для процессов, но одновременно идентифицируют структуры данных (потоки данных). Хотя здесь это и не рассмотрено, но потоки данных в DFD тщательно определены.

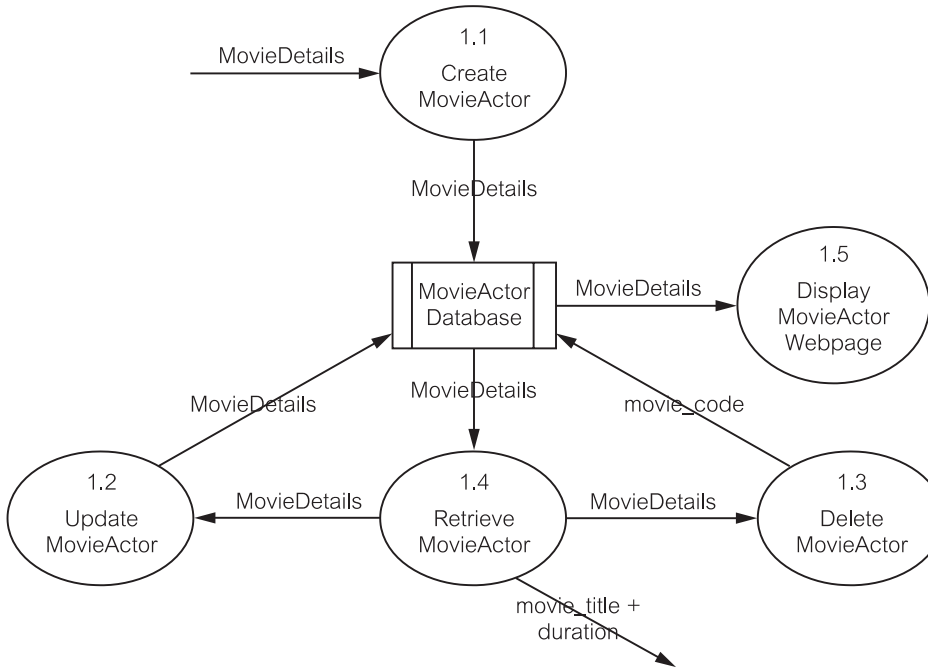


Рис. 2.4. DFD-диаграмма уровня «1»

DFD представляют весьма полное описание системы, но они все же должны быть дополнены другими технологиями, по крайней мере, в двух областях. Одна область находится на пересечении проектирования и реализации. Это область, заполненная *диаграммами структуры*, не рассмотренная здесь. Вторая область — моделирование данных. Эта область заполнена моделями «сущность–отношение», которые рассмотрены ниже.

2.1.2. Моделирование сущностей и отношений

ER-моделирование (Entity–relationship — «сущность–отношение») является технологией моделирования данных, которую популяризировал структурный анализ и синтез. Оно и сейчас остается популярной технологией для моделирования структур БД на высших (концептуальных) уровнях абстракции. Технология имеет много вариантов, из которых наиболее популярным является так называемая *нотация «вороньей лапы»*.

ER-модели представляются визуально как ERD (**entity–relationship diagrams** — диаграммы «сущность–отношение»). Диаграммы определяют только три элемента моделирования — сущности, отношения и атрибуты.

Сущность — концептуальная структура данных, которая представляет деловой факт или правило и которая может быть четко идентифицирована (как правило). Идентификация позволяет различать сущность как концепцию (*сущность-множество* или *сущность-тип*) и *экземпляры сущности* (*реализации сущности*). ERD визуализируют сущности-множества, но при условии

соответствующей интерпретации моделей можно учитывать и экземпляры сущностей.[2]

Отношения представляют связи между экземплярами сущностей различных сущностей-множеств, а в некоторых важных случаях — и одной сущности-множества. Отношения представляются в виде линий, соединяющих сущности-множества. Линии изображаются и снабжаются комментариями разнообразными способами, чтобы определить различные свойства отношений.

Сущности — контейнеры данных. Данные в этих контейнерах снабжены **атрибутами**. Атрибут — это пара параметр-значение. В современной практике, подверженной влиянию технологии реляционных БД, атрибуты имеют *единственное значение*. Атрибуты *с несколькими величинами* и *составные* атрибуты (то есть атрибуты, содержащие другие атрибуты) обычно не поддерживаются в ER-средствах визуального моделирования.

Рис. 2.5 использует ER-нотацию «вороньей лапы», чтобы представить (весьма упрощенно) концептуальную модель для БД MovieActor («Фильм-актер»). Диаграмма состоит из трех именованных сущностей и двух неименованных отношений. Свойства *множественности* отношений показываются графически и нотацией в виде двух цифр. Нотация `0,n` на отношении между `movie` (фильм) и `listed_as` (перечисление) означает, что экземпляр сущности `movie` может быть связан минимум с нулем (показано графически маленьким кружком) и максимум со многими (показано графически тремя маленькими линиями, известными как « воронья лапа») экземплярами сущности `listed_as`.

Нотация `1,1` означает минимум один и максимум один, то есть, точно один. Это также означает, что `listed_as` *принудительно присутствует* в отношениях с `movie` (каждый экземпляр `listed_as` должен быть связан с одним экземпляром `movie`). Маленькая черта на линии отношения показывает это принудительное присутствие графически.

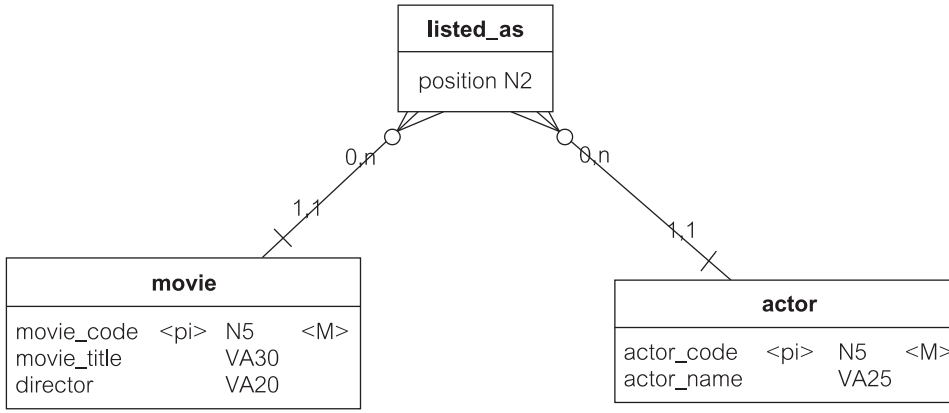


Рис. 2.5. Диаграмма «сущность–отношение» в нотации «вороньей лапы»

[2] В отечественной литературе сущности-множества или сущности-типы обычно называются просто *сущностями*, а представители сущностей — *экземплярами сущностей*. (*Прим. перев.*)

Атрибуты представлены списками внутри прямоугольников сущностей. Атрибутам дают имена (например, `movie_code` — код фильма), типы (например, `N5` — целое число максимум из пяти цифр), а также задается, является ли атрибут специальным идентификатором (`<pi>` — primary identifier — первичный идентификатор), и всегда ли он должен присутствовать (`<M>` — mandatory — принудительный).

ERD должна быть совместима со структурами данных (потоками данных и информационными хранилищами), размещенными в соответствующей DFD. Чтобы гарантировать соответствие, исходная ERD может быть получена из DFD. К сожалению, эта возможность автоматического формирования ERD из DFD никогда не была в нужной мере реализована в коммерческих CASE-средствах.

2.2. Язык объектно-ориентированного моделирования

Моделирование современных (объектно-ориентированных) систем производится с помощью **Unified Modeling Language** (UML) — унифицированного языка моделирования. UML «является языком для определения, визуализации, создания и документирования компонентов систем ПО, а также для бизнес-моделирования и других систем, не являющихся ПО. UML представляет собрание лучших технических методов, которые доказали свою эффективность в моделировании больших и сложных систем» [108].

В UML визуальное моделирование обеспечивается так называемыми **классификаторами**. Классификатор — это элемент модели, который описывает поведение или структуру системы, и обычно имеет визуальное представление. Примеры классификаторов включают class (класс), actor (актор), use case (сценарий использования), relationship (отношение).

Объект, согласно объектно-ориентированному подходу, определен как часть ПО, которая имеет состояние, поведение и индивидуальность [61]. **Состояние объекта** определяется значениями его атрибутов. **Поведение объекта** определено сервисами (операциями), которые объект может выполнять, когда он вызывается другими объектами. **Индивидуальность объекта** — свойство объекта, когда любые два объекта в системе рассматриваются различными, даже если они обладают теми же самыми значениями атрибутов и операциями. Это означает, что два объекта могут быть равны, но они никогда не идентичны. По этой же причине объект может существенно изменять свое состояние и поведение, но при этом оставаться тем же самым объектом с той же самой индивидуальностью.

В соответствии с вышеупомянутыми характеристиками объектов существует классификация объектно-ориентированных моделей, включающая *модели состояний*, *модели поведения* и *модели изменения состояний*. Каждая из этих моделей представляется одной или несколькими диаграммами.

Спецификация UML определяет шесть видов диаграмм: state structure (структуры состояний), use case (сценариев использования), interaction (взаимодействия), statechart (состояния), collaboration (сотрудничества), activity

(деятельности), а также диаграммы implementation (выполнение). Эти диаграммы рассматриваются ниже.[3]

2.2.1. Диаграммы классов

Статические структуры моделей, называемые также *моделями состояний*, отображаются в UML диаграммами классов. **Диаграмма классов** (class diagram) отображает классы (и интерфейсы), их внутреннюю структуру и их отношение к другим классам.

UML определяет **класс** как описание «множества объектов, которые обладают одинаковыми спецификациями особенностей, ограничений и семантики» [106]. Другое определение — «класс — описатель множества объектов с подобной структурой, поведением и отношениями» [108].

Особенности — это атрибуты и операции. **Атрибут** — особенность (типизированная) структуры класса. **Операция** — особенность поведения класса, которая объявляется как сервис, выполняемый экземплярами (объектами) этого класса.

В языках программирования типа Java особенности называются **элементами класса**. Атрибут — элемент данных, часто называемый **элементом-переменной**, **переменной экземпляра** или **полем** (обратите внимание, что локальная переменная, определенная внутри операции, не является элементом данных). Операция — это **элемент-функция**, обычно называемый **методом**. Вызов сервиса, оформленного в виде метода, называется посылкой **сообщения** объекту или **передачей сообщения** между объектами.

Отношение — значимая связь между классификаторами (то есть классами в случае диаграммы классов). При моделировании классов могут использоваться отношения ассоциации (association), агрегирования (aggregation) и обобщения (generalization).

Рис. 2.6 показывает основные элементы моделирования, используемые в диаграмме классов. Классы представлены прямоугольниками с тремя отделениями. Верхнее отделение содержит имя класса, среднее — список атрибутов класса, и нижнее — список операций класса. Отношения изображаются линиями, соединяющими классы.

---

[3] К сожалению, в отечественной литературе и среди российских программистов терминология унифицированного языка моделирования UML еще не устоялась. Особенно это касается диаграмм сценариев использования (use case). Само название диаграмм имеет несколько вариантов: прецедентов (по-видимому, первый перевод термина use case и поэтому широко распространенный), вариантов использования (наиболее распространенный), сценариев использования, использования и т. д. либо просто не переводится. Наиболее подходит для названия этих диаграмм термин «диаграммы сценариев использования». К сожалению, термин «сценарий» (scenario) уже используется в UML для других целей. Тем не менее, мы будем здесь использовать термин «диаграммы сценариев использования» по той причине, что термин scenario (сценарий) в книге используется нечасто, и при необходимости будут комментироваться различия между терминами «сценарий» и «сценарий использования». Все сказанное относится и к компоненту этих диаграмм, имеющему то же название — use case. Его мы тоже будем называть «сценарием использования», так как этот компонент представляет в некотором роде сценарий выполняемых им действий. Другой компонент диаграммы, который имеет много вариантов перевода, — actor: актер, исполнитель, действующее лицо, инициатор и т. д. Мы будем переводить его как «актор», также используемый в качестве названия этого компонента. Разновидности перевода других терминов не столь существенны. (*Прим. перев.*)

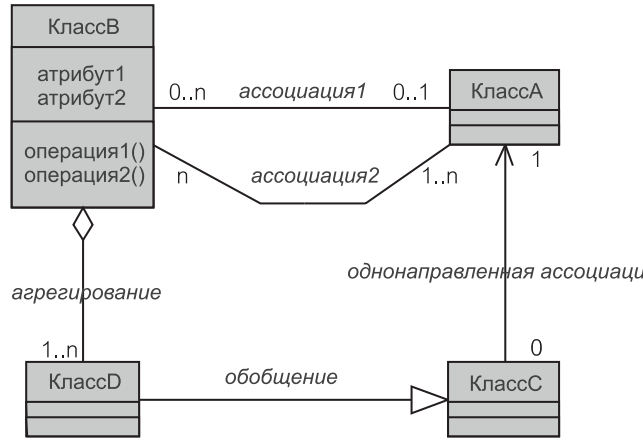


Рис. 2.6. Элементы моделирования диаграммы классов

Хотя **ассоциация** определена для классов (типов), правильнее сказать, что она является отношением между экземплярами этих классов (типов). Следовательно, множественность обеих *сторон ассоциации* (ведь ассоциация связывает два класса) очень важна. Рис. 2.6 демонстрирует возможное использование **множественности**. Множественность определяет, сколько экземпляров одного класса относится к одному экземпляру другого класса. Множественность должна быть определена на обоих концах линии, представляющей ассоциацию. Присутствие здесь нуля (`0..n` или `0..1`) указывает, что **участие** экземпляра класса в ассоциации необязательно.

Агрегирование в UML представляет специальный вид ассоциации, когда экземпляр одного класса (целое; **класс-супермножество**) содержит экземпляры другого класса (часть; **класс-подмножество**). В большинстве ситуаций модель классов предметной области может быть достаточно выразительной, чтобы не использовать агрегирование. Решение использовать агрегирование в модели классов предметной области может быть предпочтительным в чрезвычайных случаях, когда это действительно является (в UML) особой формой ассоциации.

Обобщение — отношение классификаторов, то есть классов (типов) в модели классов предметной области. Здесь каждый экземпляр более конкретного класса (**подкласса**) является также экземпляром общего базового класса (**суперкласса**). Следовательно, это не отношение экземпляров данных классов. Согласно определению обобщения, множественность к нему не применяется.

Вышеупомянутые определения должны прояснить, что диаграмма классов выражает не только статическую структуру системы. Диаграмма классов представляет также и динамическое поведение системы. Во многих случаях роль диаграмм классов в объектно-ориентированном моделировании напоминает доминирующую роль DFD в структурном моделировании. Обе определяют процессы (операции) структуры данных. Различие заключается в том, что движущей силой в DFD являются процессы, а движущая сила диаграмм

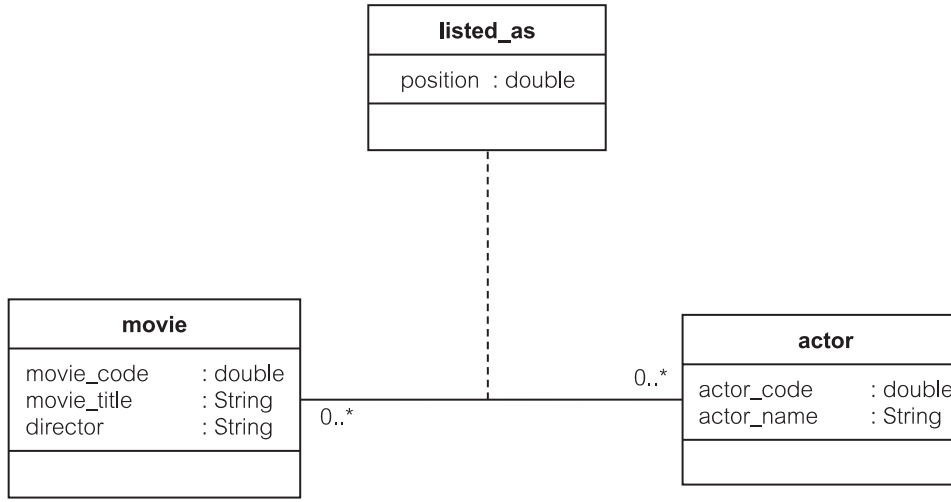


Рис. 2.7. Диаграмма классов как статическая диаграмма структуры

классов — структуры данных. Можно строить диаграммы классов и без визуализации операций. В этом случае диаграмма класса является чисто статической диаграммой структуры.

Диаграмма классов на рис. 2.7 соответствует ER-диаграмме на рис. 2.5. Интересный аспект этого примера заключается в том, что он ведет к UML-концепции класса ассоциации. **Класс ассоциации** — это класс, который представляет отношение ассоциации между другими классами. Моделирование с классом ассоциации необходимо, когда сама ассоциация имеет атрибуты, и требуется класс, чтобы хранить эти атрибуты. Класс `listed_as` (перечисление) — класс ассоциации. Он помечен пунктирной линией от него самого до отношения между двумя другими классами.

На рис. 2.8 показан UML-проект класса `Movie` (фильм), содержащий и особенности состояния, и особенности поведения. Класс состоит из трех элементов данных, метода-*конструктора* для инициализации объектов класса, и ряда методов экземпляра с их *сигнатурами* — списками формальных параметров и их типов. Обратите внимание, что *типы, возвращаемые* методами, в UML-нотации класса не отображаются.

Рис. 2.7 и 2.8 отличаются также по используемому уровню абстракции. На рис. 2.7 представлена диаграмма уровня анализа, а на рис. 2.8 изображена модель уровня проектирования класса Java. Следующий рисунок — рис. 2.9 — является моделью уровня проектирования для приложения MovieActor («Фильм-актер»). Модель состоит из пяти классов и интерфейса из библиотеки `java.util` (по имени `Collection` — коллекция).

Из трех главных видов UML-отношений — **ассоциация**, **агрегирование** и **обобщение** — первые два представлены на рис. 2.9. Все отношения — **однонаправленные** (обозначены стрелками на линиях). Это означает, например, что `MovieSearcher` (поиск фильма) содержит атрибут, названный `conn` (связывает), тип которого — `Connection` (связь) и значение которого у лю-

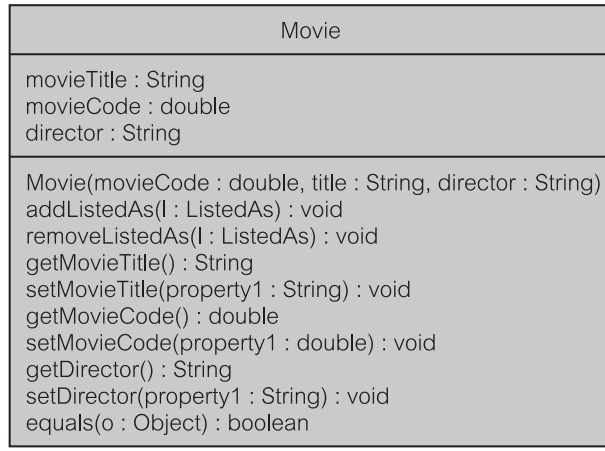


Рис. 2.8. Класс уровня проектирования с особенностями состояния и поведения

бого объекта `MovieSearcher` представляет собой *идентификатор объекта* (object identifier — OID) `Connection`. Однако с другой стороны `Connection` не содержит атрибут, связывающий его с `MovieSearcher`.

`Connection` не имеет никаких атрибутов (элементов-переменных). Это возможно в классах, предназначенных выполнять вычисления или другую обработку, не используя *объекты сущности* (объекты бизнес-данных). Атрибуты в `MovieSearcher` — *константы* (статические величины). Практически `MovieSearcher` имеет также два переменных атрибута: `conn` и `listedAs` (перечисление), полученные из его отношений с двумя другими классами. Более детальное объяснение рис. 2.9 здесь не существенно, однако дается в главе 12.

2.2.2. Диаграммы сценариев использования

Диаграммы сценариев использования (use case diagrams) — основная технология моделирования поведения на уровне анализа в UML. Можно создать много таких диаграмм, представляющих различные аспекты системы на различных уровнях абстракции. Вообще-то, эти диаграммы не создают иерархические структуры, напоминающие DFD.

Основная особенность этих диаграмм заключается в том, что они используют не только графические представления. Главное достоинство диаграмм сценариев использования определяется текстовыми спецификациями, размещенными в хранилище. Эти текстовые *спецификации сценариев использования*, наподобие заданных для учебного примера EM (Email Management — управление электронной почтой) в главах 8, 13 и 19, являются детальным описанием требований пользователя. Спецификации сценариев использования сопровождают разработчиков на всех стадиях жизненного цикла и в большинстве задач моделирования. Они управляют задачами анализа, проектирования и реализации. Их используют для разработки контрольных приме-

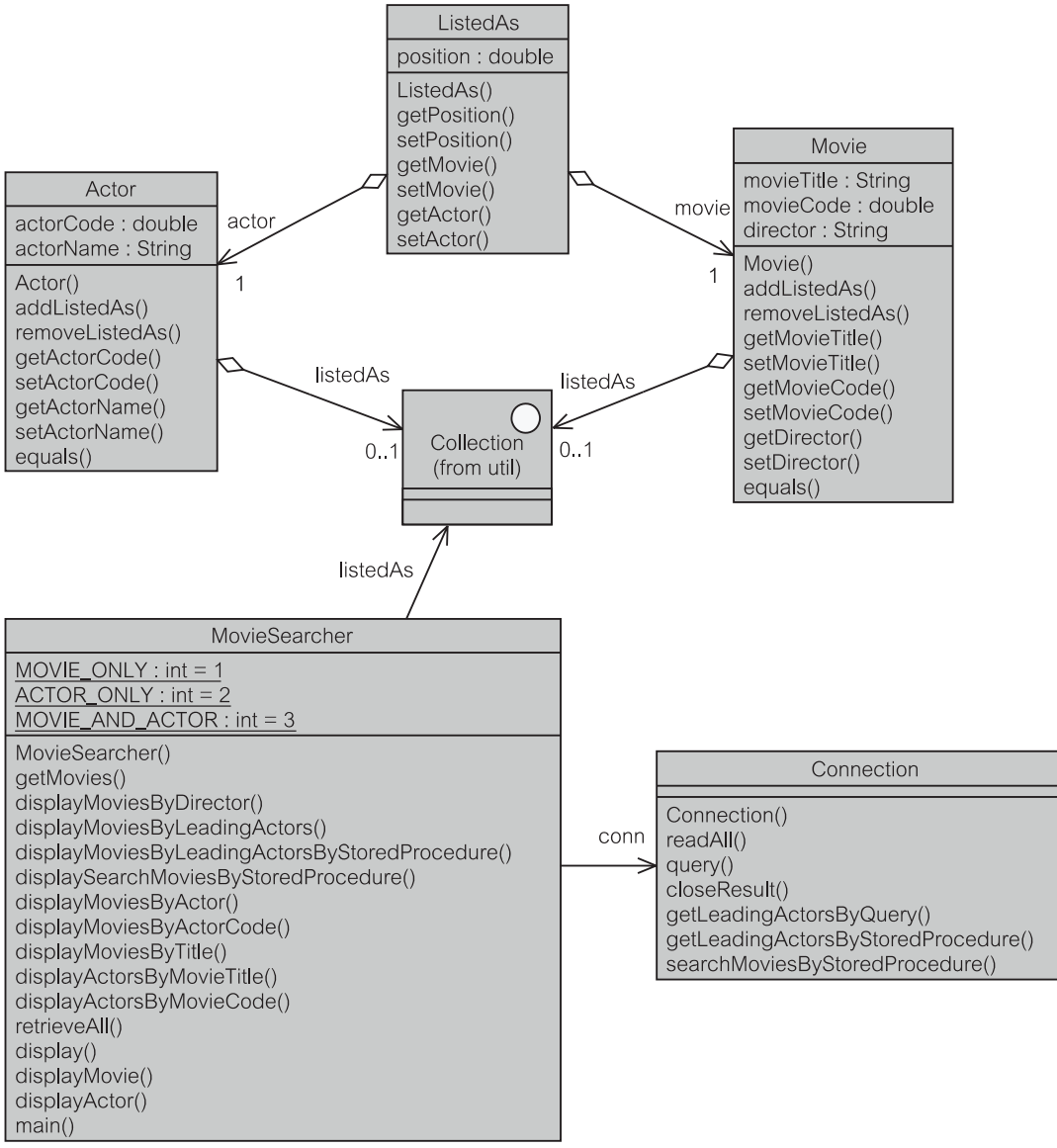


Рис. 2.9. Диаграмма классов уровня проектирования с учетом состояния и поведения

ров, фиксации ошибок и дальнейшей корректировки. Спецификации необходимы для определения задач развития и сопровождения.

Сценарий использования представляет главную часть функциональных возможностей системы. Актор (действующее лицо) представляет роль, которую кто-то или что-то играет в отношении сценариев использования. Актор связывается со сценарием использования (через отношение, определяемое стереотипом «`communicate`», то есть, связующее отношение) и ожидает от

него некоторую обратную связь — величину или наблюдаемый результат. (В UML имя, помещенное в двойные угловые скобки типа «communicate», называется стереотипом. UML-**стереотип** используется, чтобы ввести новую концепцию в модель, если она первоначально не поддерживалась в UML.)

В диаграмме сценариев использования размещаются элементы — сценарии использования (изображенные в виде овалов) и акторы (изображенные в виде человечков). Графическое размещение элементов — сценариев использования и акторов — не имеет никакого значения. Общепринято размещать сценарии использования в середине диаграммы, а акторы — на ее границах.

Иногда желательно рисовать несколько диаграмм сценариев использования для одной и той же предметной области, при этом каждая из них подчеркивает различные аспекты модели или ее части. Две типичные точки зрения (кроме, конечно, основной модели):

- *никаких акторов* — отображаются только сценарии использования и их отношения;
- *точка зрения одного конкретного актора* — отображаются все сценарии использования для каждого важного актора.

Акторы в модели могут быть разбиты на категории с помощью отношений обобщения. Никакие другие UML-отношения между акторами не используются. Сценарии использования могут быть связаны отношениями ассоциации и обобщения либо двумя стереотипными отношениями, называемыми «include» (включение) и «extend» (расширение). Акторы и сценарии использования связаны специальным видом ассоциации, называемой «communicate», — **связующим отношением**.

Рисунок 2.10 показывает, как классификаторы (элементы модели) могут быть связаны в модели сценариев использования. Сценарии использования

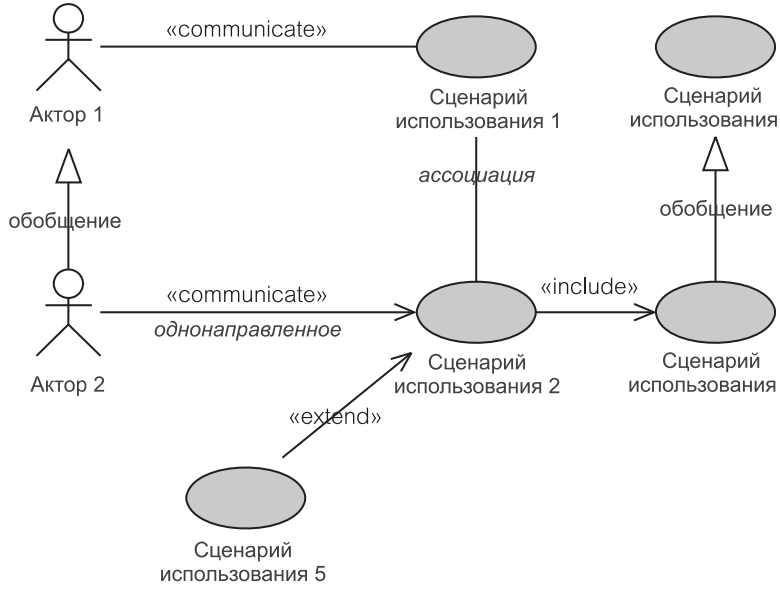


Рис. 2.10. Элементы моделирования диаграмм сценариев использования

связаны, потому что для решения задач системы они обычно должны взаимодействовать. Однако рекомендуется ограничивать отношения между ними. Показ всех отношений затеняет смысл модели. При показе только отобранных отношений возникает вопрос, почему эти отношения более важны, чем другие. Здесь нужно помнить, что, в конечном счете, диаграммы сценариев использования играют вторичную роль по отношению к текстовым документам, определяющим каждый из них.

Принцип благоразумного использования отношений применяется также к двум отношениям-стереотипам в UML, встречающимся при моделировании сценариев использования, — отношениям «include» и «extend». Отношение «include» означает, что выполнение одного сценария использования охватывает (всегда) функциональные возможности включенного сценария использования. Например, Withdraw Money (выдача денег) включает Check Account Balance (проверка баланса счета). Withdraw Money называется **основным сценарием использования**, а Check Account Balance — **дополнительным сценарием использования**. Отношения «include» позволяют общие функциональные возможности разбить так, что к одному и тому же дополнительному сценарию использования можно обращаться из нескольких основных сценариев использования. Отношения «include» могут также использоваться для уменьшения размера сложного сценария использования, выделяя из него дополнительный элемент.

Отношение «extend» означает, что выполнение одного сценария использования может потребовать расширения (иногда) функциональными возможностями второго сценария использования. Например, Print Transaction Record (печать операционного отчета) расширяет Withdraw Money. Print Transaction Record называется **расширяющим сценарием использования**, а Withdraw Money — основным сценарием использования. Отношения «extend» могут использоваться для моделирования исключительного или необязательного поведения, но главной причиной является то, что основной сценарий использования в своих функциональных возможностях не зависит от расширения (следовательно, направление стрелки — от расширения к основному сценарию использования).

Отношения «include» и «extend» всегда **однонаправлены**, чтобы обозначить направление включения или расширения. Отношения «communicate» и ассоциации могут иметь, а могут и не иметь стрелки, означающей, что передвижение (**навигация**) возможно лишь в одном направлении. Отсутствие стрелки означает, что или перемещение возможно в обоих направлениях, или направление перемещения не определено (то есть, возможно, что это будет определено в будущем, но пока нас не волнует).

На рис. 2.11 представлена диаграмма сценариев использования для приложения MovieActor («Фильм-актер»). Диаграмма соответствует двум процессам (процессы 1 и 2) DFD-диаграммы с рис. 2.3. Она состоит из трех акторов и шести сценариев использования. Корпоративная БД (Corporate Database) представлена в виде актора, чтобы подчеркнуть, что она находится отдельно от смоделированного приложения. Отображение информации кинозрителю требует включения функций двух сценариев использования,

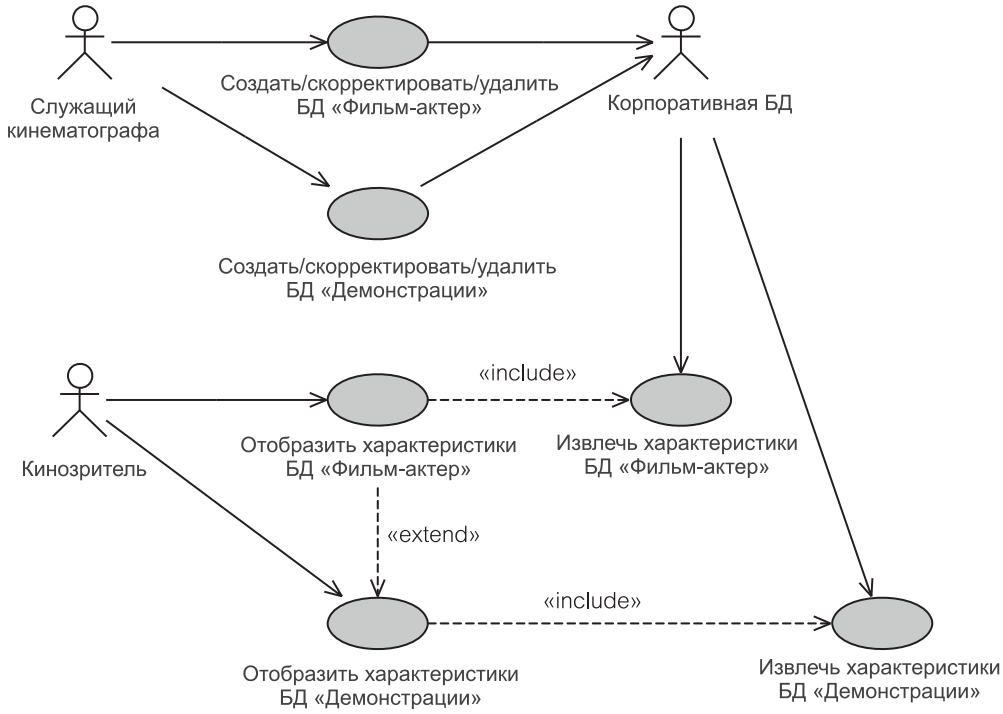


Рис. 2.11. Диаграмма сценариев использования

осуществляющих поиск в БД. Отображение информации о демонстрациях расширяется показом деталей из БД «Фильм-актер» всякий раз, когда кинозритель хочет получить больше информации о фильмах и актерах, сыгравших в данных фильмах.

2.2.3. Диаграммы взаимодействия

Диаграммы взаимодействия (interaction diagrams) в UML — основная технология моделирования поведения (на уровне проектирования). Они могут применяться для конкретных сценариев использования, групп сценариев использования, а также их частей. Диаграммы могут быть представлены на различных уровнях абстракции и с различной степенью детализации.

Последовательно создаваемые UML-стандарты внесли существенные изменения в диаграммы взаимодействия и в интерпретацию элементов моделирования, используемых в этих диаграммах. Подход, принятый в этой главе, состоит в том, чтобы различать два вида диаграмм взаимодействия: *диаграммы последовательности действий* и *диаграммы сотрудничества* (называемые *диаграммами связей* в разделе 11.3.2). Эти две диаграммы взаимозаменяемы, и существует много CASE-средств для их автоматического преобразования.

Диаграммы последовательности действий

Диаграмма последовательности действий (sequence diagram) — раздел 11.3.1 — графическая визуализация последовательности сообщений между объектами, то есть, последовательности вызовов объектами методов, которые завершаются выполнением некоторых задач. Акцент в диаграмме последовательности действий делается на последовательность сообщений. Размещение сообщений одно под другим показывает это. Необязательная нумерация сообщений также указывает на эту последовательность. Объект, получающий сообщение, активизирует соответствующий метод. Время, когда поток управления сосредоточен на объекте, называется **активизацией**. Активизации показываются в виде узких прямоугольников на *линиях жизни* объекта (вертикальные пунктиры).

Иерархическая нумерация сообщений показывает отношения активизаций к сообщениям. В частности, сообщение самому себе (то есть метод обращается к тому же самому объекту) в пределах одной активизации приводит к новой активизации. Это показывает и иерархическая нумерация, и графическое добавление нового узкого прямоугольника.

Как показано на рис. 2.12, диаграммы последовательности действий различают следующие сообщения:

- **синхронные сообщения** — представленные зачерненной стрелкой;
- **сообщения обратной связи** — представленные пунктирной линией с незачерненной стрелкой; такие сообщения называются **откликами на метод**, и их не следует путать с **возвращением метода** через вызов сообщения, кото-

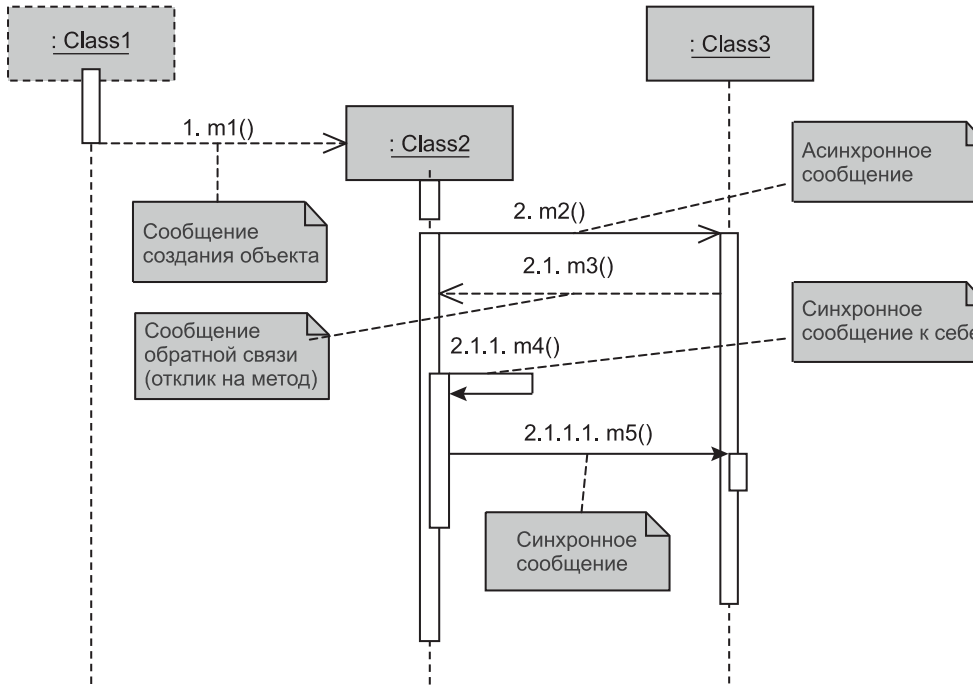


Рис. 2.12. Сообщения в диаграмме последовательности действий

рое неизвестно в конце активизации и обычно не показывается на диаграммах взаимодействия;

- **асинхронные сообщения** — представленные незачерненной стрелкой;
- **сообщения создания объекта** — представленные пунктирной линией с незачерненной стрелкой, направленной к прямоугольнику объекта.

Взаимодействия имеют дело с последовательностями сообщений, а не с данными, которые эти сообщения передают. Поэтому сигнатуры сообщений не нужно показывать, а возвращаемые типы обычно не отображаются. О возвращаемых типах часто можно судить из контекста.

Чтобы загрузить объекты сущностей в память, приложение MovieActor («Фильм-актер») должно запросить БД и создать объекты сущностей из набора результатов, возвращенных запросами. Для простоты приложение MovieActor состоит только из пяти классов: трех классов сущностей, класса `MovieSearcher` (поиск фильмов) и класса `Connection` (связь). Класс `MovieSearcher` начинает обработку и отображает результаты доступа к БД на экран. Класс `Connection` взаимодействует с БД.

Рисунок 2.13 представляет диаграмму последовательности действий для сценария использования `Retrieve MovieActor Details` (извлечение характеристик MovieActor). Этот сценарий использования требует загрузки в

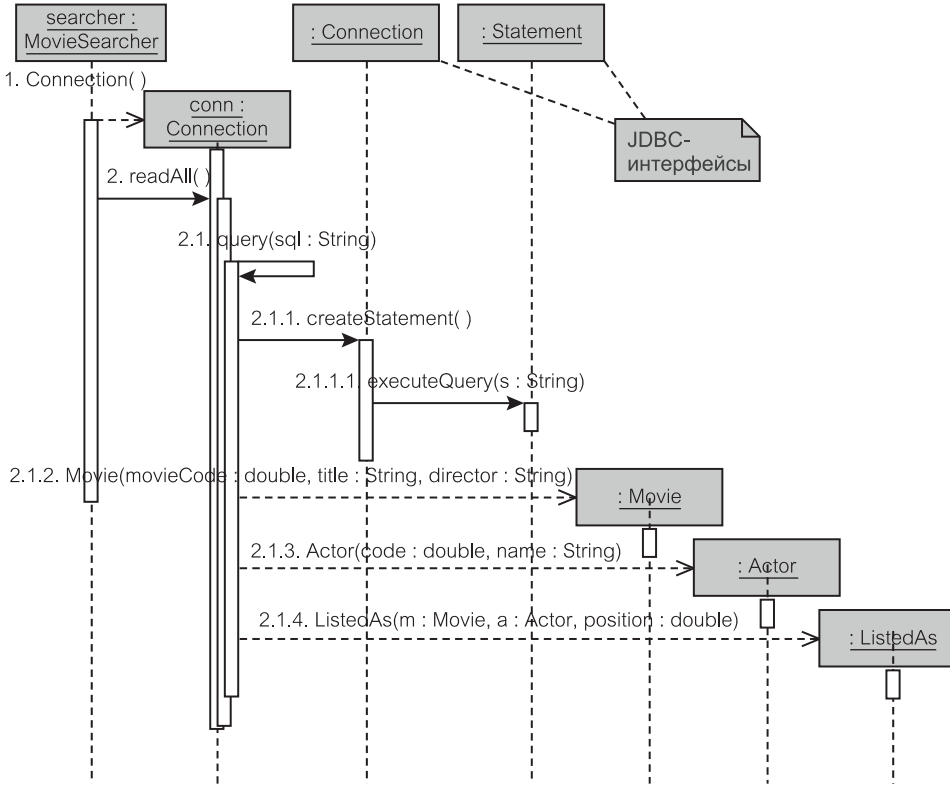


Рис. 2.13. Диаграмма последовательности действий

память объектов сущностей (то есть заполнение кэша сущностей после извлечения данных из БД).

Объект `searcher:MovieSearcher` (объект «поиск» класса MovieSearcher) инициализирует объект `conn:Connection` (объект «связь» класса Connection) и затем посылает ему сообщение `readAll()` (читать все). Метод `readAll()` формирует строку SQL-запроса и передает ее методу `query()` (запрос). Метод `query()` посылает сообщение `createStatement()` (создать оператор выполнения запроса) объекту `conn`. Это показано на рис. 2.13 как сообщение к JDBC-интерфейсу (см. раздел 12.1.3) `Connection` (связь). `createStatement()` возвращает объект `Statement` (оператор выполнения запроса). Сообщение `executeQuery()` (выполнить запрос) в объекте `Statement` формирует объект `ResultSet` (результирующий набор), который возвращается методу `readAll()`.

Диаграммы сотрудничества (связей)

Диаграмма сотрудничества (связей) (collaboration (communication) diagram) — раздел 11.3.2 — разновидность диаграммы последовательности действий. На рис. 2.14 изображена диаграмма сотрудничества, которая соответствует диаграмме на рис. 2.13. Сообщения в диаграмме сотрудничества показываются непрерывной линией со стрелкой и с именем сообщения. Непрерывная линия также указывает, как объект, посылающий сообщение, запросил «руку» объекта-получателя (это могло быть, например, связью ассоциации или просто знакомством).

Диаграмма сотрудничества не отображает линии жизни (однако сами объекты представляют линии жизни). Порядок активизаций подразумевается в

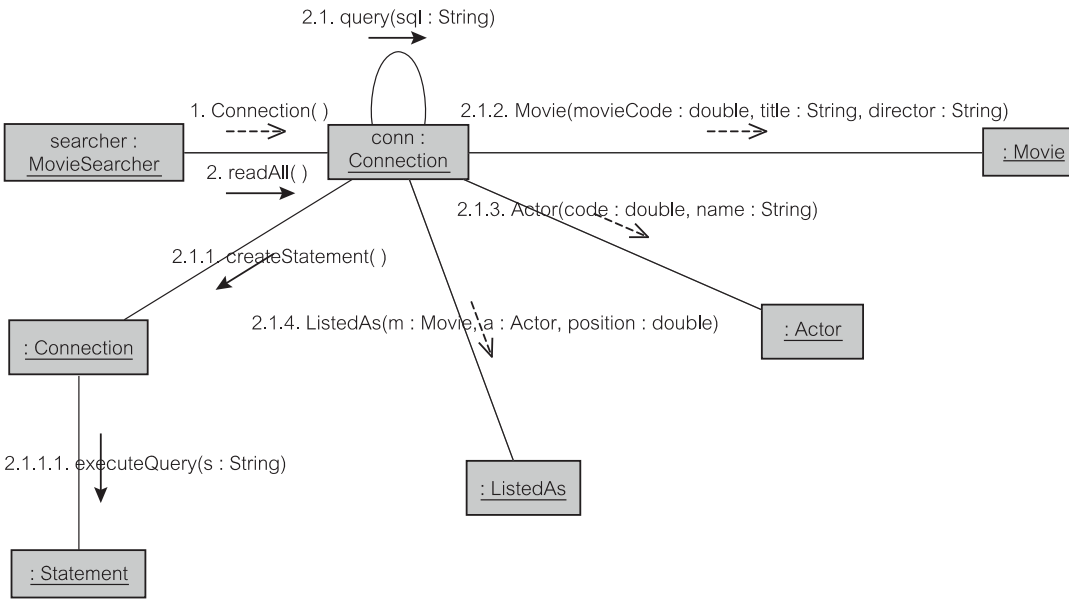


Рис. 2.14. Диаграмма сотрудничества (связи)

иерархической нумерации сообщений. Сообщение создания объекта также подразумевается (до некоторой степени) в нумерации — новый созданный объект продолжает нумерацию со следующей последовательной величины. Кроме этого, нет никаких визуальных различий между сообщениями создания объекта и обратной связи.

На практике разработчики предпочитают использовать диаграммы последовательности действий, а не диаграммы сотрудничества. Благодаря своей наглядности диаграммы последовательности действий имеют преимущество в представлении более сложных моделей, где явная визуализация последовательности сообщений является существенной — даже притом, что они могут потребовать печати на бумаге большого формата с использованием принтеров (графопостроителей), способных обеспечить такие форматы.

Диаграммы сотрудничества могут быть более полезны для анализа сообщений от данного объекта или к данному объекту. Они могут быть более приемлемы при изображении начального проекта взаимодействий и при выполнении «итеративного» моделирования методом проб и ошибок. Из-за этого они весьма удобны для метода «мозгового штурма».

2.2.4. Диаграммы состояний

В отличие от большинства других диаграмм UML, диаграммы состояний (statechart diagrams) применимы не только к объектно-ориентированному моделированию. Диаграммы состояний используют обозначения и идеи традиционных моделей **конечных автоматов** [96], применяемых в течение десятилетий для моделирования приложений, работающих в реальном масштабе времени. Эти виды приложений должны моделировать возможные состояния устройств и систем типа химических контейнеров, подъемников, электрических выключателей, электростанций, и т. д.

Диаграмма состояний охватывает состояния объекта и действия, которые приводят к переходам между состояниями этого объекта. Диаграмма состояний создается для каждого класса, который имеет интересные изменения состояний, достойные моделирования. **Состояние** объекта (экземпляра класса) меняется, когда изменяются значения ряда его атрибутов. Например, объект `Student` (студент), чей атрибут `enrollmentStatus` (посещаемость) имеет значение «`part—time`» (частичная), находится в состоянии `PartTime` (частичная). Тот же самый объект `Student` может иметь другой атрибут `paymentStatus` (состояние оплаты), которому задано значение «`paid`» (оплачено). Тогда `Student` находится в состояниях `PartTime` и `Paid`.

Закрепление диаграммы состояний за классом — не единственное ее использование. Диаграммы состояний могут быть построены для других элементов моделирования, которые имеют интересное динамическое поведение, типа сценариев использования, акторов, операций, взаимодействий.

Пример диаграммы состояний для класса `Movie` (фильм) изображен на рис. 2.15. Пример задуман так, чтобы проиллюстрировать разнообразие конструкций и вариантов моделирования. Состояния, представленные прямоугольниками с закругленными углами, характеризуют реакции объекта на входные события. **Переходы** показаны линиями со стрелками, берущими на-

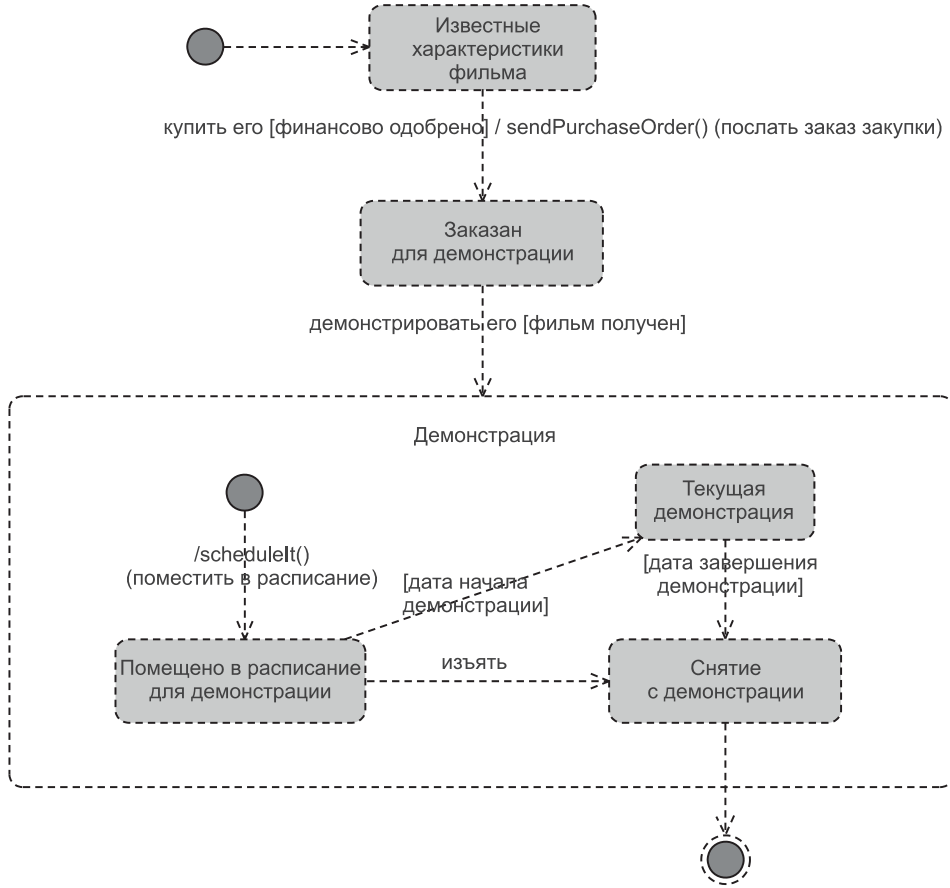


Рис. 2.15. Диаграмма состояний

чало из *исходного состояния* и завершающимися в *заданном состоянии*. *Начальное состояние* обозначено черным кругом. *Окончательное состояние* представлено черным кругом с окружностью, известным в жаргоне как «глаз быка». Диаграмма может иметь несколько начальных состояний и несколько конечных состояний.

«Состояние — это условие во время жизни объекта или взаимодействие, в течение которого оно (состояние) удовлетворяет некоторым условиям, либо выполняет некоторое действие, либо ожидает некоторое событие. *Составное* состояние — это состояние, которое в отличие от *простого* состояния, имеет графическую декомпозицию» [108]. «Демонстрация» на рис. 2.15 — составное состояние. Начальным простым состоянием внутри состояния «Демонстрация» является «Помещено в расписание для демонстрации».

Состояния обладают продолжительностью. Она соответствует интервалу времени между двумя переходами. Продолжительность — относительное понятие. Обычно состояния требуют больше времени на нашей относительной

шкале времени, чем то, которое определяется переходами. Ведь состояния могут представлять вычисления.

Переход может снабжаться примечанием согласно следующему формату:

сигнатура_события [условие блокировки] */действие-выражение*

Использование любого из этих трех элементов в формате необязательно.

События могут иметь различные типы. В частности, это могут быть события вызова подпрограммы, то есть сообщения. События могут также иметь параметры. **Условия блокировки** — логические выражения, определяющие при каких условиях переход «инициализируется» (осуществится). **Действия** выполняются один раз, когда переход «инициализируется». Они обычно представляют методы объекта-владельца.

2.2.5. Диаграммы деятельности

Диаграмма деятельности (activity diagram) — конечный автомат, который выполняет вычисления, то есть выполняет такие действия, от завершения которых зависят дальнейшие переходы. Как правило, диаграмма деятельности дополняет реализацию операций или сценарий использования. В текущей версии UML [108] диаграммы деятельности используются исключительно для отображения поведения, используя модель потока управления и данных, напоминающую *сети Петри* [34]. В предыдущих версиях UML диаграммы деятельности служили объединенной цели задания спецификации поведения и выполнения роли конечного автомата [61].

На рис. 2.16 представлена **диаграмма деятельности** UML, снабженная UML-комментариями, объясняющими назначение графических символов. Диаграмма изображает действия (состояния действий), вызванные классом `CAdmin` (администрирование), когда тот запрашивает сервис `get EMovie` (получить фильм) у класса `MDataMapper` (отображение данных). Первый блок ветвления используется потому, что `EMovie` (фильм) может найти фильм в кэше программы, но если его там нет, то класс `FReader` (чтение) извлекает его из БД. Даже если `EMovie` и находит информацию в кэше, но она изменена (то есть, изменена после того, как была извлечена из БД), процесс также переадресуется к `FReader`.

Состояние действия диаграммы деятельности несколько отличается от *обычного состояния* диаграммы состояний. В отличие от обычных состояний, которые могут иметь рекурсивные (внутренние) переходы к самому себе, состояния действия — это вычисления, которые не должны прерываться внешними событиями и не должны иметь никаких выходов, связанных с какими-либо явными событиями. Выходы из состояния действия — результат завершения работы этого состояния.

Объект диаграммы деятельности подобен объекту (линии жизни) диаграммы взаимодействия. **Потоки объектов** (пунктирные стрелки) между объектами и состояниями действий определяют, какой объект посылает сообщение, и какой объект содержит метод, чтобы обслужить это сообщение. Объект, который является результатом действия, обычно представляет вход к одному или нескольким последовательным действиям. **Потоки управления** (сплошные стрелки) используются, когда состояние действия формирует ре-

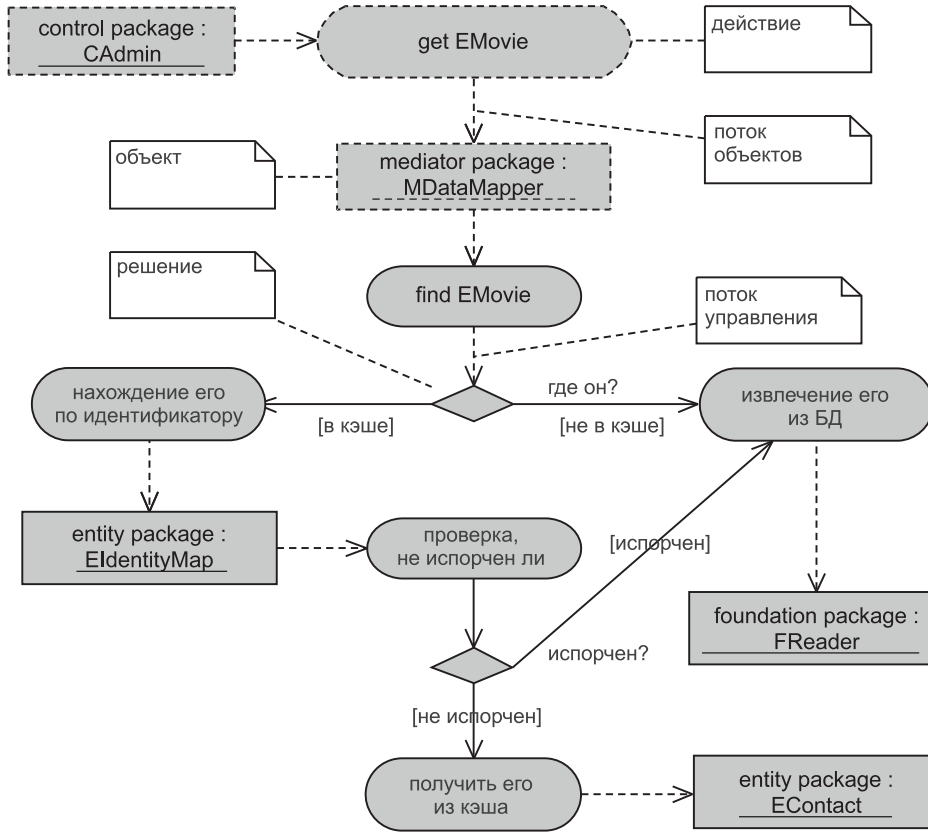


Рис. 2.16. Диаграмма деятельности

зультат (обычно принятое решение), который является входом в состояние последовательных действий.

2.2.6. Диаграммы выполнения

Диаграммы выполнения (implementation diagrams) — модели физической реализации системы. Они показывают компоненты системы, их структуру и зависимости, а также как они размещены в узлах компьютерной системы. «Диаграммы выполнения» — обобщающий термин для двух видов диаграмм: диаграммы компонентов и диаграммы размещения. Диаграммы компонентов показывают структуру компонентов, включая их интерфейс и зависимости реализации. Диаграммы размещения показывают распределение системы во время выполнения по отдельным узлам вычислительной системы.

Документация по UML [108] указывает, что диаграммы выполнения могут широко использоваться для бизнес-моделирования и организационного моделирования. При таком моделировании компоненты могут представлять бизнес-сущности, процедуры и другие продукты. Узлы размещения могут представлять организационные единицы и ресурсы (люди, машины, деньги и т. д.) бизнес-проекта.

Диаграммы компонентов

«**Диаграмма компонентов** (component diagram) показывает зависимости между компонентами ПО, включая классификаторы, которые определяют их (например, классы реализации), и продукты, которые реализуют их, типа файлов с исходным кодом, исполняемых файлов, скриптов» [108]. «**Компонент** представляет модульную, размещаемую и заменяемую часть системы, который инкапсулирует выполнение и предлагает ряд интерфейсов» [108].

Компонент изображается в виде прямоугольного блока, снабженного двумя меньшими прямоугольниками. Он не имеет своих особенностей типа атрибутов или операций и выступает как контейнер для других *спецификаторов*, которые имеют особенности. Компонент *неявно* предлагает ряд интерфейсов, которые представляют сервисы, обеспечиваемые элементами, размещенными в компоненте. *Явно* такие интерфейсы могут быть обозначены кругом, снабженным именем интерфейса. Связь, представленная прямой линией, соединяет компонент с интерфейсом для указания, что конкретный компонент предлагает услуги, определенные в интерфейсе.

Компоненты могут быть связаны с другими компонентами физическим включением (прямое вложение компонента в охватывающий его компонент). Физическое включение представляет композиционное отношение, называемое отношением «`reside`» — размещение (следовательно, альтернативным обозначением включения на диаграмме может служить связующая линия). Аналогично, физическое включение компонента в продукт может быть изображено отношением «`implement`» — выполнение.

Пунктирные стрелки используются, чтобы изобразить зависимости между компонентами. Такие же стрелки используются, чтобы показать зависимости между компонентом и интерфейсом. Смысл зависимости в том, что конкретный компонент зависит от сервисов, определенных в интерфейсе, нежели от сервисов, определенных в другом компоненте.

Пакеты могут использоваться в диаграмме компонентов для иллюстрации их группировки. Чтобы указать определяемые реализацией зависимости или характеристики компонентов, могут использоваться стереотипы.

Рис. 2.17 показывает диаграмму компонентов для небольшого приложения MovieActor («Фильм-актер»). Стереотип «`Entity`» (сущность) используется для указания, что компонент размещает объекты сущности. Пакет `Movie_Actor` (фильм-актер) содержит три компонента-сущности, а именно `Actor` (актер), `Movie` (фильм) и `ListedAs` (перечисление). Эти компоненты-сущности физически расположены в файле с именем `Movie_Actor.jar` (фильм-актер, jar — файл архива Java). Включение показано отношением «`implement`».

Компонент `SearchController` (контроллер поиска) ответственен за выполнение поиска в БД MovieActor на основе критериев поиска, заданных пользователями. Чтобы выполнить это, `SearchController` использует два интерфейса, называемые `searchMovie` (поиск фильма) и `searchActor` (поиск информации об актере). Эти интерфейсы реализованы в компонентах `MovieSearcher` (поиск фильма) и `ActorSearcher` (поиск информации об актере) соответственно. Всякий раз, когда `SearchController` нуждается в

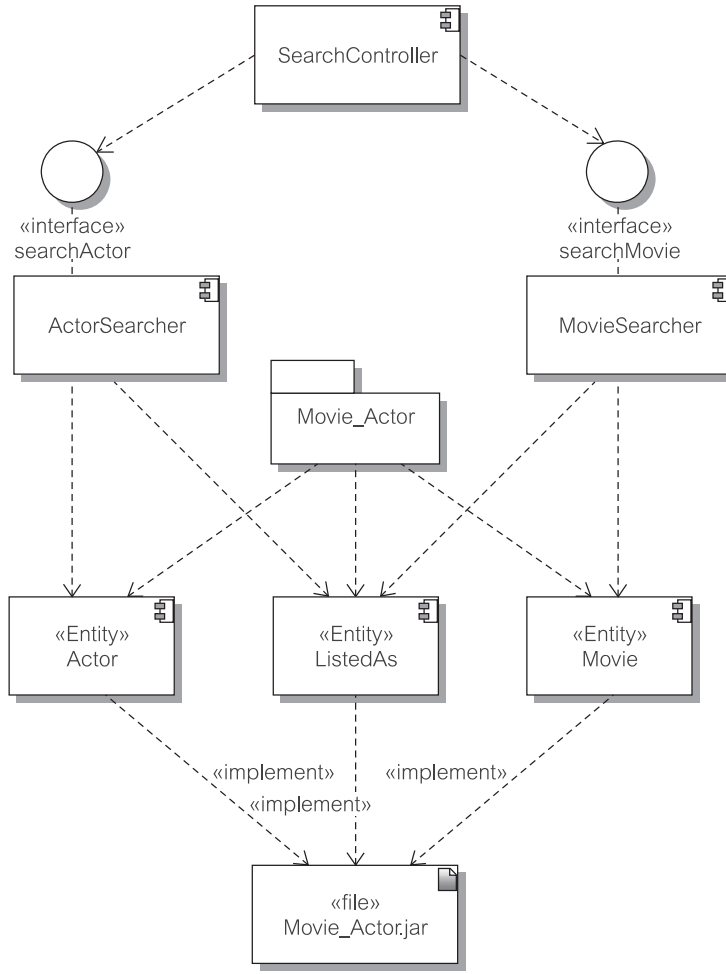


Рис. 2.17. Диаграмма компонентов

некоторой информации о фильме, он запрашивает `searchMovie`. Точно так же используется `searchActor` для поиска информации об актерах.

Диаграммы компонентов можно легко приспособить к изменениям, связанным с использованием различных платформ реализации. Например, если бы в качестве платформы реализации использовались Enterprise Java Beans (EJB), рис. 2.17 может быть модифицирован соответствующими стереотипами компонентов. В случае EJB `SearchController`, `ActorSearcher` и `MovieSearcher` имели бы стереотип «`EJB Session`» (EJB-сессия) для указания, что они — *Bean-компоненты сеанса EJB*. `Movie`, `ListedAs` и `Actor` имели бы стереотип «`EJB Entity`» (EJB-сущность), чтобы представить *Bean-компоненты сущностей EJB* (см. раздел 3.3.3 и гл. 22).

Диаграммы размещения

«**Диаграммы размещения** (deployment diagrams) показывают конфигурацию элементов процесса времени выполнения, а также компоненты ПО, процессы и объекты, которые там выполняются. Экземпляры компонентов ПО представляют реализацию (времени выполнения) модулей кода ПО» [108]. Эти экземпляры размещаются в узлах, где они запускаются или выполняются.

«**Узел** — физический объект, который представляет ресурс обработки, обычно имеющий, по крайней мере, память, а часто также и способность обработки. Узлы включают вычислительные устройства, а также человеческие или механические ресурсы обработки» [108].

Размещение компонентов, показанных на рис. 2.17, изображено на рис. 2.18. На этой диаграмме основное внимание уделяется размещению ком-

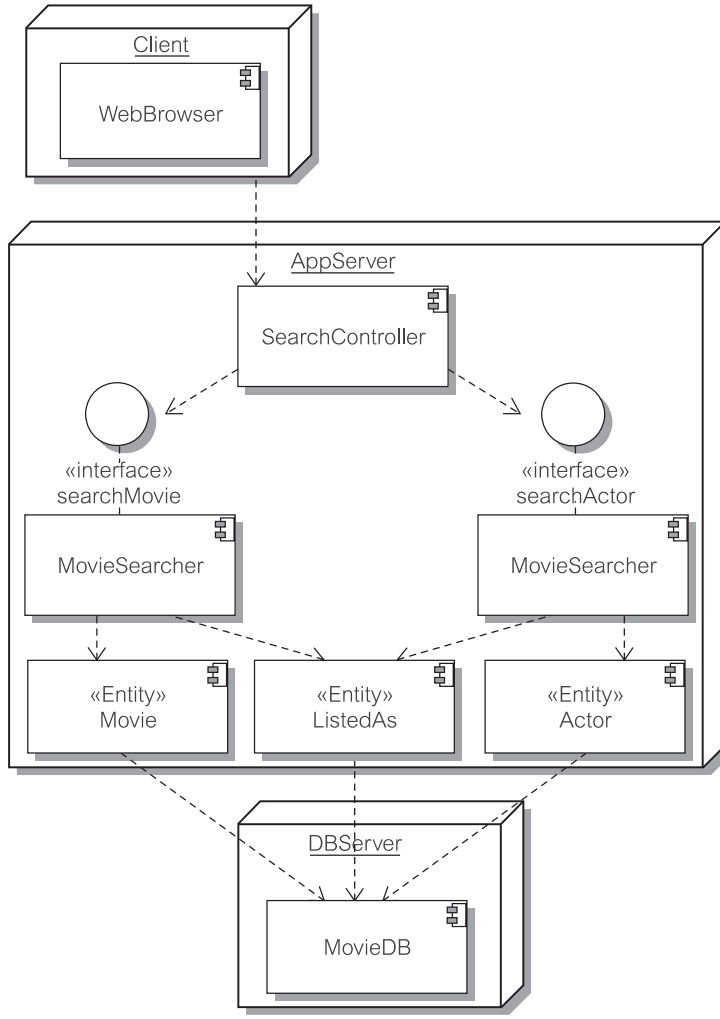


Рис. 2.18. Диаграмма размещения

понентов и взаимодействию ресурсов. Узлы обозначены параллелепипедами. Рис. 2.18 определяет механизм `Client` (клиент), который размещает компонент `WebBrowser` (Web-браузер). Большинство компонентов, определенных на рис. 2.17, помещено в сервер приложения `AppServer` (сервер приложения). Компоненты `Movie`, `ListedAs` и `Actor` организуют доступ к БД, представленной компонентом `MovieDB` (БД фильмов) и расположенной в узле `DBServer` (сервер БД).

Резюме

1. UML (*Unified Modeling Language* — унифицированный язык моделирования) — стандартный язык моделирования для современных *объектно-ориентированных* систем ПО. Существуют различные расширения UML, известные как *UML-профили*, чтобы адаптировать UML к определенной предметной области и технологии.
2. Язык *структурного моделирования* в поддержку разработок, соответствующих *функциональному подходу* в старом стиле (глава 1), включает целый диапазон технологий визуализации, наиболее популярными из которых были диаграммы потока данных (data flow diagrams — DFD), диаграммы «сущность-отношение» (entity-relationship diagrams — ERD) и диаграммы структуры.
3. Структурное моделирование обычно нацелено на функциональную декомпозицию, представленную в DFD. ERD и диаграммы структур используются как поддерживающие технологии. DFD имеют три основных элемента моделирования: процессы, потоки данных и информационные хранилища.
4. ERD — технология моделирования данных, которая широко используется за пределами структурного моделирования для представления структур БД. ERD строятся, используя два графических элемента: сущности и отношения. И сущности, и отношения могут содержать атрибуты.
5. UML предлагает широкий диапазон технологий моделирования для представления различных точек зрения на моделируемую систему. Диапазон диаграмм включает диаграммы классов, диаграммы сценариев использования, диаграммы взаимодействия, диаграммы состояний, диаграммы деятельности и диаграммы выполнения.
6. *Объектно-ориентированное UML-моделирование* имеет дело с диаграммами классов, но в первую очередь определяется диаграммами сценариев использования. Модель сценариев использования — ориентир для всех других моделей. Другие модели обращаются к модели сценариев использования, чтобы выяснить требования пользователя и/или проверить, соответствуют ли они требованиям пользователя. Диаграммы классов представляют и состояние, и поведение системы. В конечном счете, модели классов определяют главный подход к программированию.
7. *Диаграммы классов* визуализируют классы объектов, содержание их атрибутов и операций, а также отношения между классами. Имеются три вида отношений: ассоциации, агрегирования и обобщения.

8. *Диаграммы сценариев использования* обеспечивают простую визуализацию сценариев использования, их отношений и акторов, взаимодействующих со сценариями использования. Однако реальная мощь сценариев использования не в визуальных моделях, а в текстовых определениях требований пользователя, которые должны быть обеспечены этими диаграммами.
9. *Диаграммы взаимодействия* — основная технология моделирования поведения (уровня проектирования) в UML. Они представляют передачу сообщений в системе. Имеются два вида диаграмм взаимодействия: диаграммы последовательности действий и диаграммы сотрудничества (связи).
10. *Диаграммы состояний* охватывают состояния объектов и действия, которые приводят к переходам между состояниями объектов. Они предназначены для отдельных классов, хотя могут также использоваться и для моделирования изменений состояния в сложных элементах модели, таких как пакеты или даже целая система.
11. *Диаграмма деятельности* — конечный автомат, который представляет выполняемые в системе вычисления. Как правило, диаграмма деятельности дополняет реализацию операций или сценарий использования.
12. *Диаграммы выполнения* — модели для физической реализации системы. Они показывают компоненты системы, их структуру и зависимости, а также их размещение в узлах компьютерной системы. Имеются два вида диаграмм выполнения: диаграммы компонентов и диаграммы размещения.

Ключевые термины

Обзорные вопросы

1. Объясните место и роль хранилища данных в моделировании ПО. Что это? Что оно содержит?
2. Почему структурные анализ и проектирование называются «структурными»? Какими могли бы быть альтернативные термины? Обсудите.
3. Можно ли в DFD иметь поток данных между двумя информационными хранилищами или между внешней сущностью и информационным хранилищем? Объясните.
4. Объясните требование балансировки потока данных в DFD. Приведите пример, демонстрирующий возможные случаи, когда требование балансировки нарушается.
5. Объясните, как в ERD связаны понятия множественности и участия.
6. Объясните сходства и различия между отношениями агрегирования и ассоциации в моделировании класса.
7. Обратимся к рис. 2.8. Объясните, как используется интерфейс `Collection` (коллекция) в Java, чтобы представить отношения между классами в модели.

8. Отношение «extend» (расширение) в моделях сценариев использования представляется в виде стрелки от расширения к основному сценарию использования (например, «проверка багажа» расширяет сценарий использования «входная проверка перед рейсом»). Не лучше ли рисовать стрелку в противоположном направлении и изменить имя стереотипа в отношении на «is extended by» — расширено с помощью... (то есть, «входная проверка перед рейсом» расширяется «проверкой багажа»)? Объясните.
9. Обсудите ситуации, когда моделирование с помощью диаграммы последовательности действий имеет преимущества по сравнению с использованием диаграммы сотрудничества и наоборот.
10. Объясните и проиллюстрируйте различия между событиями, условиями блокировки и действиями в моделях состояния.
11. Что такое поток объектов и поток управления в диаграмме деятельности?
12. Каково назначение интерфейса в диаграмме компонентов?

Примеры задач

1. Обратимся к диаграмме контекста на рис. 2.2. Является ли эта диаграмма полной для описанной проблемы? Если нет, то как бы вы улучшили модель?
2. Рассмотрите сущность по имени Student (студент) с атрибутами вроде student_id (идентификатор студента), student_name (имя студента) и т. д. Пусть к тому же система должна хранить элементы адресов студентов, чтобы можно было группировать студентов по их почтовым индексам или даже по названиям улиц, на которых они живут. Дополнительно нужно хранить номера телефонов, которых у отдельного студента может быть несколько. Как такая информация может быть смоделирована в ERD? Предложите различные модели и объясните все их «за» и «против».
3. Рассмотрим диаграмму классов на рис. 2.7. Изобразите альтернативную диаграмму, представляющую то же моделирование. Объясните семантику обеих диаграмм и приведите какие-то их семантические различия.
4. Использование диалоговой системы резервирования покупки театральных билетов включает действия наподобие выбора места, оплаты с помощью кредитной карты, получения скидки на основе истории приобретения клиентом билетов и принятия решения относительно того, должен ли билет быть отправлен по почте в адрес клиента или он будет получен на месте. Изобразите диаграмму сценариев использования для вышеупомянутого сценария, заполненную отношениями «include» (включение) и «extend» (расширение).
5. Почему на рис. 2.17 не показано, что ListedAs (перечисление) фактически является картой связи между сущностями Actor (актер) и Movie (фильм), то есть, ListedAs — объект отношения, который устанавливает отношение актеров к фильмам и наоборот? Должна ли диаграмма вместо этого показать что-то подобное изображенному на рис. 2.19? Обратите внимание, что рис. 2.19 показывает только эти три рассматриваемых

компонента, и предположите, что другие компоненты — те же, что и на рис. 2.17.

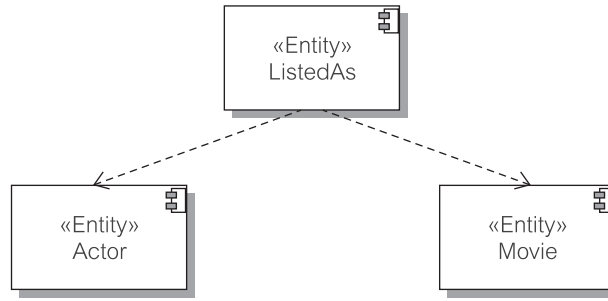


Рис. 2.19. Зависимости сущности ListedAs для сущностей Actor и Movie

Глава

3 Инструментальные средства программной инженерии

Имеются два важных подхода к инструментальным средствам программной инженерии: индивидуальный и организационный. Инструментальное средство может использоваться для индивидуальной и/или кооперативной разработки (в составе рабочей группы, в совместной работе). В обоих случаях оно требует определенного уровня знаний, навыков и управления. Для определения этих требований используется термин «компетентность» (глава 5). Без адекватного уровня компетентности использование инструментального средства может больше повредить, чем помочь.

Имеется важное различие между индивидуальной компетентностью и организационной компетентностью в процессе совместной работы. В первом случае инструментальное средство находится под управлением единственного человека. Во втором — нет. Вывод очевиден. Если компетентность всего коллектива разработчиков недостаточна, использование инструментальных средств может препятствовать совместной производительности. Инструментальное средство задает уровень строгости в выполнение работы. Если организация работ находится в беспорядке, использование инструментальных средств в процессе разработки произведет лишь еще большее количество беспорядка.

Понятно, что использование инструментальных средств программной инженерии для повышения производительности **коллективной работы** требует больших инвестиций в людей и технологии. Инструментальные средства в этом случае принадлежат к категории программных продуктов, известной как категория, обеспечивающая **совместную автоматизированную разработку** (computer-supported collaborative work — CSCW) или **групповую вычислительную обработку** (workgroup computing). CSCW-средства — сложные системы; фактически они принадлежат к группе наиболее сложных программных продуктов, которые можно купить. Этому есть несколько причин.

Во-первых, совместная работа требует, чтобы технология БД поддерживала хранение, восстановление и распределение продуктов разработки. Во-вторых, многие продукты разработки — графические модели. Хранение, восстановление и распределение таких мультимедиа-объектов создают реальную проблему для обычной (реляционной) технологии БД. В-третьих, творческая работа занимает много времени и может привести к результатам, которые находятся в противоречии с другими решениями, она должна учитывать выявленные ошибки, результаты эксперимента и т. д.

Короче говоря, совместная работа требует такой технологии разработки БД, которая может управлять разными версиями различных продуктов и ком-

бинировать их в различные окончательные конфигурации. И снова, обычная технология разработки БД не предусматривала эти виды проблем. Продавцы инструментальных средств программной инженерии часто не имеют никакого другого выбора, кроме как развивать технологию «переднего края» и помещать ее в свои изделия.

Базовый термин для инструментальных средств программной инженерии — **автоматизированная программная инженерия** (computer assisted software engineering — CASE). Однако практически смысл CASE-средств обычно ограничивается возможностями визуального моделирования для анализа и проектирования систем. Хотя большинство таких инструментальных средств имеет возможность прямого проектирования (формирование кода) и обратного проектирования (от существующего кода к визуальным моделям), они обычно не составляют законченную среду программирования. Инструментальное средство, которое всесторонне обеспечивает программиста, называется **интегрированной средой разработки** (integrated development environment — IDE).

CASE и IDE не охватывают все инструментальные средства, используемые в процессах создания ПО. Жизненный цикл ПО требует таких инструментальных средств *управления проектом*, которые планировали бы и управляли ресурсами проекта. Он также требует средств *управления изменениями*, чтобы те учитывали недостатки, обеспечивали корректировку и заботились о полном сопровождении ПО. Будучи по своей сути деятельностью коллектива людей, программная инженерия нуждается в технологии *управления конфигурацией* ПО. В соответствии с этим ниже представлены следующие группы инструментальных средств программной инженерии:

- инструментальные средства управления проектом;
- инструментальные средства моделирования систем (CASE);
- среды формирования ПО (IDE);
- инструментальные средства управления изменениями и конфигурацией.

Инструментальные средства программной инженерии — крупный и достаточно конкурентоспособный бизнес. В то время, когда читатели получат эту книгу, используемые в промышленности инструментальные средства, упомянутые в этой главе, могут быть, а могут и не быть в представленной здесь форме. Читатель должен обратиться к Web-сайту книги для ознакомления с обновлениями или попытаться выполнить с помощью Интернета соответствующие исследования самой последней информации (как предложено в разделе «Решение задач» в конце этой главы).

3.1. Инструментальные средства управления проектом

Управление проектом объединяет множество инструментальных средств, методов и технологий, связанных с планированием проекта и управлением интегрированным процессом. Методы и технологии управления проектом и процессами рассматриваются в следующих двух главах. Инструментальные

средства, их основные свойства и возможности обсуждаются в этой главе. Обсуждение инструментальных средств управления внесением изменений и конфигурацией, в ряде случаев рассматриваемых как часть управления проектом, отложено до раздела 3.4.

Управление проектом — нечто вроде распределения и управления бюджетом, временем и людьми. Главные вопросы здесь: Сколько будет стоить разработка системы? Сколько потребуется времени на разработку системы? Какие люди нам нужны для разработки системы?

Старый принцип гласит, что если вы не сможете спланировать что-то, вы не сможете это и сделать. Но планирование не может быть «высосано из пальца». Оно требует некоторых начальных знаний того, что требуется для разработки системы в этой организации, с этими людьми, с этими ресурсами, в этой организационной культуре и т. д. Чтобы иметь это, нужно знать прошлое организации. Организация должна оценить, что происходило с предыдущими проектами. Она должна определить из предыдущих проектов *метрики* (систему показателей). Это приводит к следующему принципу: «если вы не знаете ваше прошлое, вы не можете должным образом планировать и ваше будущее».

Управление проектом требует использования инструментальных средств для эффективного планирования и регулирования показателей проекта, для оценки проектных затрат, для сбора метрик и т. д. Современные инструментальные средства часто включаются в интегрированные средства поддержки управления. Эти средства объединяют обычное управление проектом со стратегическим планированием, бизнес-моделированием, портфельным управлением, управлением документацией, технологическим процессом и т. д. Такие интегрированные среды представляют единое универсальное инструментальное средство для управления проектами в более широком контексте выявленных стратегических инициатив, обеспечения тактического управления, выполнения ежедневных задач и управления людьми. Интегрированные среды управления могут касаться не только предприятия, но и поставщиков, клиентов и других деловых партнеров.

Лучшие представители управления большими проектами используют Web-технологии, которые позволяют осуществлять динамическое, интерактивное и синхронное управление большими коллективами и процессами. Многие традиционные инструментальные средства управления проектами имеют Web-версии [79].

3.1.1. Планирование и управление проектом

Инструментальные средства для **планирования и управления** показателями проекта помогают в подготовке графиков операций. **Сетевые графики** операций задают события, которые должны произойти до того, как начинается операция, определяют продолжительность каждой операции и ее сроки, распределяют людские и другие ресурсы на все операции и т. д. Два традиционных вида графиков операций — **метод критического пути** (critical path method — CPM) и **диаграммы Ганта** (Gantt charts). Известное расширение CPM — это **система планирования и руководства разработками** (Program Evaluation and Review Technique — PERT).

Инструментальные средства управления проектами могут формировать график операций, как только будет обеспечена необходимая проектная информация. Требуемая проектная информация включает определение операций (задач), их продолжительность и ресурсы, предшествующие события и события, которые являются следствиями операций, и т. д. Проектную информацию можно разместить в тексте или ввести непосредственно в график.

Инструментальные средства управления проектами могут также автоматически преобразовывать графики (например, PERT в диаграммы Ганта и наоборот) и представлять графики, подчеркивающие различные точки зрения и с различными деталями. Например, отдельные графики могут быть сделаны для всего проекта и для отдельных сотрудников, работающих над проектом. Графики могут отображаться, используя различные временные шкалы, показывая (или не показывая) определенные человеческие и другие ресурсы, продолжительности операций и/или даты их начала и окончания и т. д.

Инструментальные средства управления проектами регулируют и перестраивают планы всякий раз, когда происходят изменения в операциях или ресурсах. Планы корректируются, одновременно заново пересчитываются даты начала и конца операций с информацией о последних изменениях. Они могут быть повторно рассчитаны, чтобы учесть влияние добавления ресурсов в проект. Это позволяет менеджерам понять, может ли дополнительный ресурс дать лучшие результаты (иногда организаторские накладные расходы, связанные с добавлением нового ресурса, могут увеличить время разработки).

Рисунок 3.1 — простой пример CPM-сети, представленной в MS Project® от фирмы Microsoft [70] — одном из лучших инструментальных средств

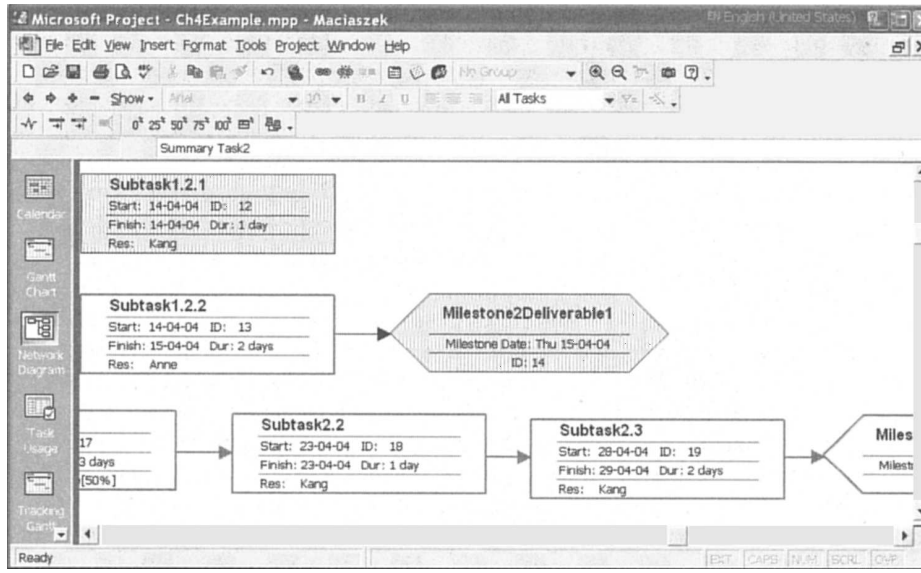


Рис. 3.1. MS Project®: CPM-диаграмма

Источник: образ экрана в Microsoft® Office Project (2003), перепечатанный по разрешению Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

управления проектами. Диаграмма показывает ряд проектных задач (операций) и контрольные точки (еще называемые вехами — milestones). Для каждой задачи определяются даты начала и конца, продолжительность и назначенный людской ресурс. Пример на рис. 3.1 обсуждается более подробно в главе 4.

3.1.2. Управление проектированием и реализацией с учетом основных показателей

Организации функционируют в конкурентной и динамичной среде. Чтобы выживать и успешно конкурировать, организации должны постоянно приспосабливать свои показатели, цели, планы, а также проекты разработок к изменениям в потребностях клиентов, к новым технологиям, изменениям внешних факторов и т. д. В этих условиях управление проектом должно отвечать требованиям совершенствования результативности, эффективности и качества.

Неудивительно, что ряд инструментальных средств управления проектами включен в более общие программные средства, чтобы обеспечить широкое разнообразие управленческих операций. В частности, они обеспечивают управление проектом в контексте совершенствования **управления реализацией** и стратегического планирования. Они помогают в управлении людьми и коллективами в вопросах планирования, отслеживания результатов и достижения бизнес-целей. Акцент перемещен от проектов к людям. С людьми нельзя обращаться как с простыми «ресурсами», распределенными по операциям или задачам. Людьми управляют, задавая им бизнес-цели и прослеживая результаты.

Рис. 3.2 представляет краткий обзор ManagePro™ от Performance Solutions Technology [65]. Это — инструментальное средство для целевого управления проектами на основе используемого коллектива людей. ManagePro™ разработано так, чтобы быть основным средством управления. Оно может управлять целями, проектами, планами разработки, встречами, электронной почтой, документами и анализом выполнения работ, включая годовые обзоры.

3.1.3. Унификация управления проектом с организацией совместной работы и информационного обеспечения на основе Web-технологии

Многие проекты ПО являются распределенными — они охватывают большое количество людей и выходят за обычные организационные границы. Поэтому связь между участниками создания таких проектов должна использовать Web-технологии. Управление проектом становится «компонентом» сотрудничества *специалистов в сфере информационных технологий* на основе Интернета. Специалисты принадлежат различным распределенным коллективам. Чтобы обеспечить выполнение проекта, они должны планировать, общаться, сотрудничать, принимать решения, распределять обязанности и т. д. по Интернету.

Управление совместной работой включает поддержку передачи спорных вопросов заинтересованным сторонам проекта, определения и задания соот-

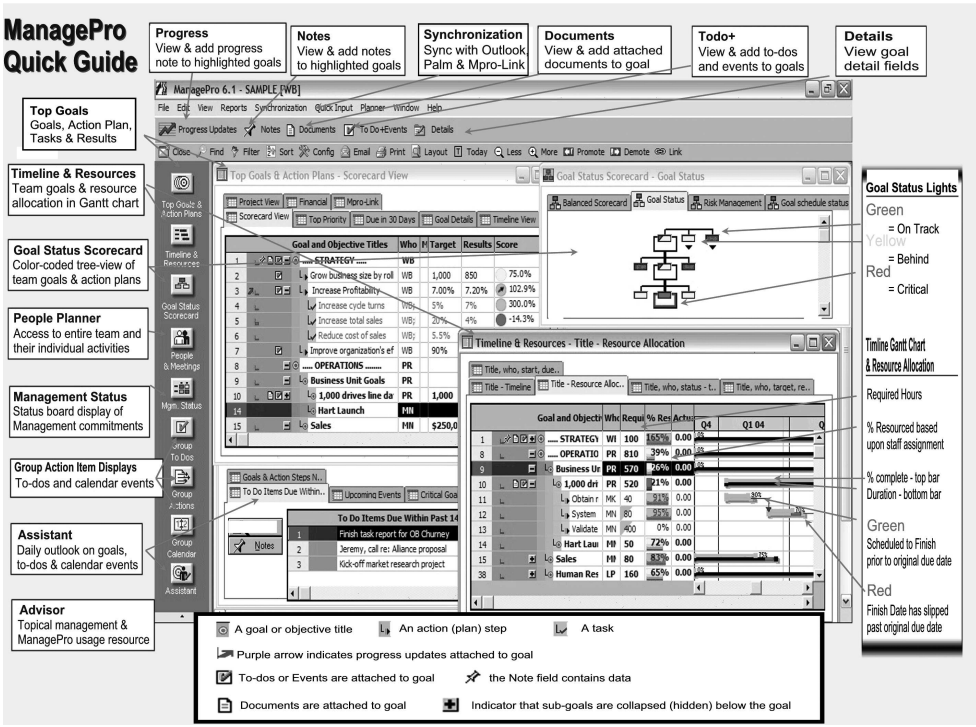


Рис. 3.2. ManagePro™: общий вид

Источник: образ экрана ManagePro™ 6.1 из www.managepro.net, март 2004, перепечатано с любезного согласия Performance Solutions Technology, LLC

ветствующих мероприятий, формирования документации. Чтобы напоминать о событиях и выполняемых операциях, могут быть установлены правила совместной работы, использующие сообщения электронной почты. Таблицы учета рабочего времени для участников создания проекта становятся неотъемлемой частью управления проектом, и весь коллектив автоматически информируется относительно продвижения в выполнении проекта.

Наряду с обеспечением сотрудничества участников проекта управление проектом на основе Web-технологии обеспечивает легкий доступ и передачу документации и других информационных материалов типа Web-страниц и сообщений по электронной почте. Инструментальные средства такой категории используют общедоступные библиотеки документации и другие источники информации, то есть, выполняют то, что сегодня известно как **информационное обеспечение**. Чтобы информировать участников создания проекта об его изменениях, широко используется механизм подписки на рассылку информации.

Среда eRoom, разработанная фирмой Documentum [29], является примером инструментального средства на основе Web-технологии, которое объединяет управление проектом с организацией совместной работы и информационным обеспечением. Рис. 3.3 иллюстрирует диаграмму Ганта в корпоративной среде eRoom с использованием Web-технологии.

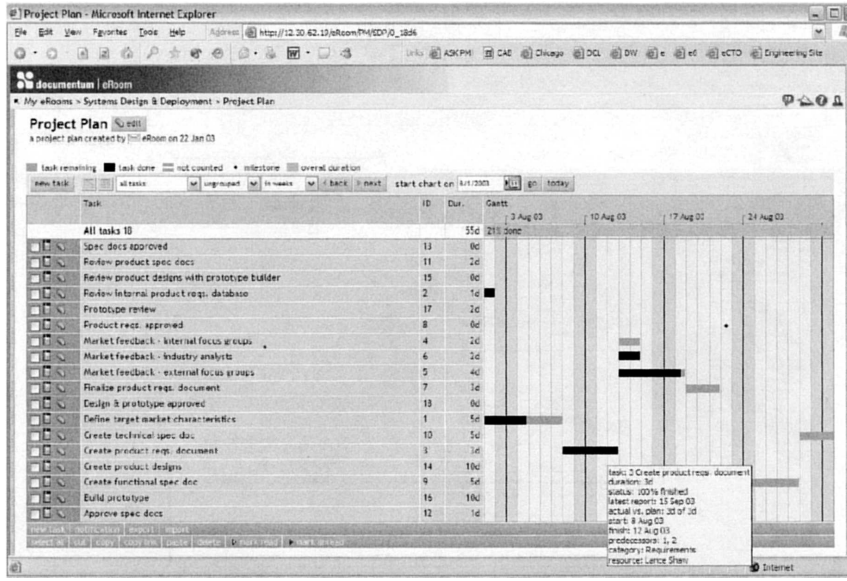


Рис. 3.3. Инструментальное средство eRoom: диаграмма Ганта
Источник: образ экрана eRoom из сайта www.eroom.net/eRoomNet/, июль 2003 г., перепечатано с любезного согласия Documentum, Inc

3.1.4. Унификация управления проектом на основе портфельной Web-технологии

Объединение управления проектом с организацией сотрудничества и информационным обеспечением, все это с использованием Интернета, является важным шагом в планировании и составлении графиков. Этот шаг является *многопроектным* и *с ограниченными ресурсами*. Инструментальные средства, поддерживающие такое планирование и составление графиков, концентрируются на совместных ресурсах и их распределении, а также на оценке общего времени. Этот процесс предоставления предприятию возможности распределить ресурсы и инвестиции в соответствии с основными показателями называется управлением портфелем проектов предприятия или просто **портфельным управлением**.

Инструментальные средства, которые объединяют управление проектом с портфельным управлением предприятия, обычно являются комбинацией целого ряда связанных средств типа управления коллективами и ресурсами, связями и совместной работой, документооборотом и информационным обеспечением. Инструментальные средства этой категории типа eProject Enterprise фирмы eProject [28] или Primavera Enterprise® Product Suite фирмы Primavera Enterprise [82] обеспечивают среду разработки на основе Web-технологий для:

- обсуждения проектов, назначения работ и их отслеживания;

- составления графиков работы коллективов и таблиц учета рабочего времени;
- проведения электронных конференций для достижения согласия и выработки консультативных решений;
- распределения проектных документов и поддержания их версий;
- автоматизации технологического процесса и отслеживания решений;
- обычного планирования и составления графиков с использованием диаграмм Ганта, CPM и PERT;
- формирования шаблонов документов и образцов для выбора лучших методов;
- установления соответствия проектов стратегическим планам и бизнес-моделям;
- управления индивидуальными и распределенными ресурсами;
- распределения возможностей человеческих и других ресурсов по проектам;
- формирования взаимных проектных резюме и индикаторов состояния.

Рис. 3.4 иллюстрирует итоговый отчет о проекте, сформированный в eProject Enterprise [28]. Здесь показаны секторная диаграмма потраченного бюджета по отношению к оставшемуся, график, изображающий затраты на запланированные задачи по сравнению с фактической стоимостью, и иерархия задач, где отображено, как они влияют на общее состояние проекта. Панель инструментов в верхней части окна показывает, как инструментальное средство можно использовать для персонального управления и для совместной работы в пределах всего предприятия.

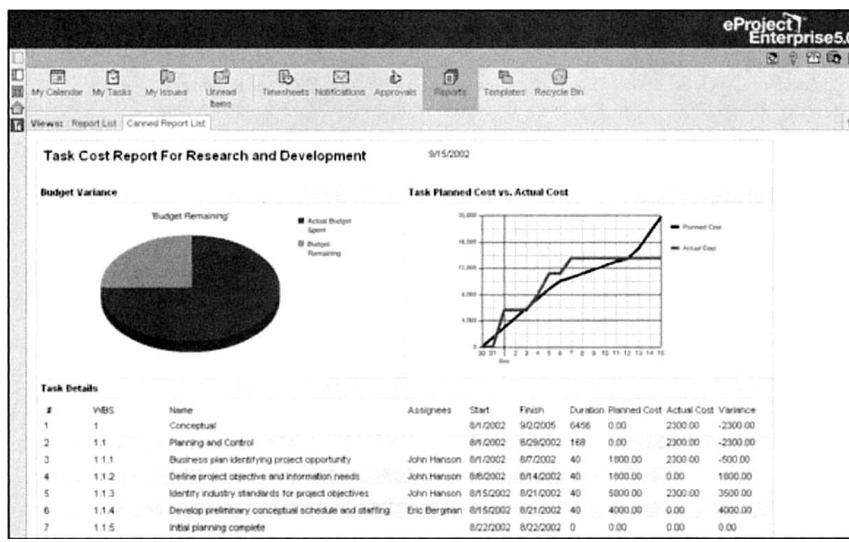


Рис. 3.4. eProject Enterprise: общий отчет о проекте
Источник: образ экрана eProject Enterprise, июль 2003, перепечатано с любезного согласия eProject, Inc

3.1.5. Интеграция управления проектом с метриками

Метрики в программной инженерии не имеют никакого отношения к Европейской метрической системе. **Метрики** — это порядок количественной оценки процессов разработки ПО и созданных продуктов с тем, чтобы собранная информация могла помочь в будущем планировании проектов.

Несмотря на их важность, в большинстве проектов метриками пренебрегают. Ими жертвуют под ежедневным давлением необходимости выполнить текущий проект. Метрики требуют ресурсов и их перемещения от непосредственной разработки к сбору системы показателей — жестокое требование! Эта работа может также привести к требованию остановки в некоторых областях, хотя сам проект будет и на правильном пути.

Поскольку сбор метрик — дорогое удовольствие, иногда отвергаемое разработчиками и менеджерами, использование удобных и информативных инструментальных средств получения системы показателей приобретает особую важность. Инструментальные средства получения метрик должны обеспечить легкий доступ к информации. Всякий раз, когда это возможно, информация должна автоматически получаться из моделей разработки и программного кода.

Анализ кода структурного проекта — важный пример того, как управление проектом может быть дополнено критически важными метриками. Структурное проектирование (глава 9) отвечает за *возможность сопровождения* (supportability) системы, то есть ее понятность, удобство сопровождения и масштабируемость. Поэтому важно оценить возможность сопровождения системы как на ранних стадиях жизненного цикла ПО, так и на поздних, когда должно быть оценено влияние изменений существующего кода. Метрики нацелены на определение мест минимизации зависимостей компонент с тем, чтобы улучшить общую стабильность и качество разрабатываемой системы.

Рис. 3.5 показывает пример анализа «*что, если...*», выполненного на коде с 7 пакетами (6 из них видимы), представленными квадратами, и 62 классами, изображенными в виде кругов и имеющими 201 отношение. Анализ проводился на классе `FWriter` (запись) — крайний левый круг. Линии показывают потенциальные зависимости от `FWriter` так, что воздействие изменения `FWriter` на другие классы может быть понятно. Список справа — объекты, на которые потенциально возможно воздействие — включает внутренние классы Java. Внутренние классы Java представлены также для большого количества кругов в правом верхнем пакете. Рис. 3.5 был создан с помощью Small Worlds [95]. Это инструментальное средство было приобретено в июле 2003 фирмой IBM (его функциональные возможности должны быть включены в портфель продуктов IBM, известный как IBM Rational Quality by Design).

Рис. 3.6 — другой образ экрана от Small Worlds. Граф представляет внутренние зависимости классов и интерфейсов, которые существуют в классе по имени `CAdmin` (класс «администрирование» пакета `control` — см. разделы 9.1.6, 9.2.2). Пунктирные стрелки обозначают зависимости «*использует*». Сплошные стрелки показывают зависимости «*содержит*».

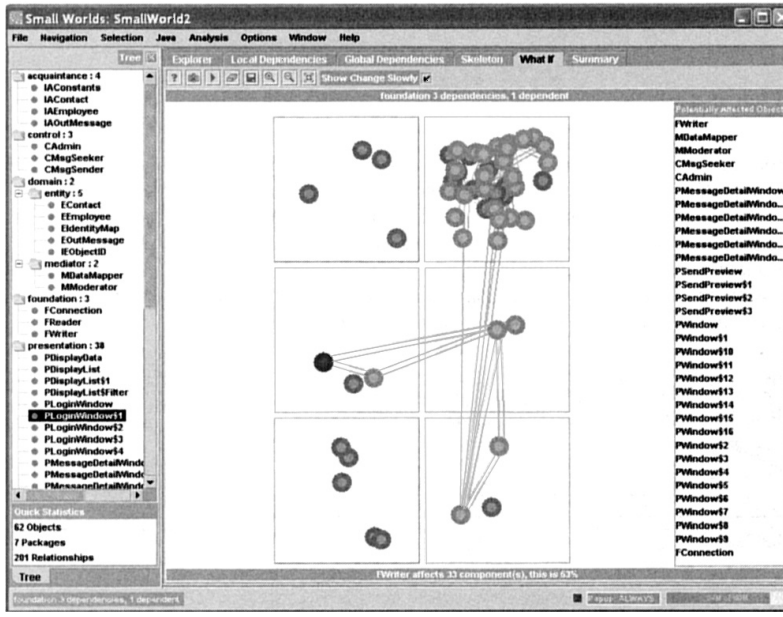


Рис. 3.5. Small Worlds: анализ «что, если...»
Источник: образ экрана *Small Worlds*, перепечатано по разрешению с www.thesmallworlds.com/, © Copyright 2003 by International Business Machines Corporation. All Rights Reserved

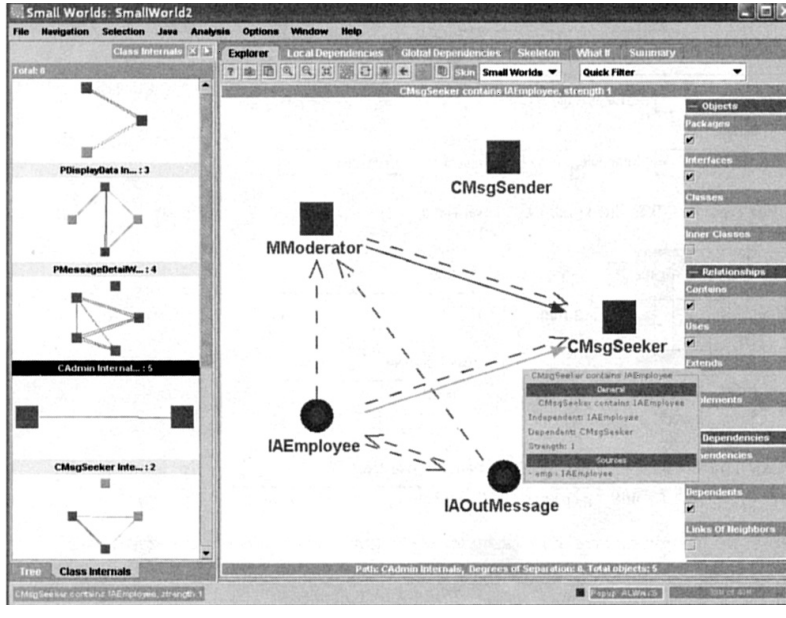


Рис. 3.6. Small Worlds: анализ внутренних зависимостей классов
Источник: образ экрана *Small Worlds*, перепечатано по разрешению с www.thesmallworlds.com/, © Copyright 2003 by International Business Machines Corporation. All Rights Reserved

3.1.6. Интеграция управления проектом с управлением рисками

От управления проектом и метрик есть только небольшой шаг к анализу рисков проектов. Гецци и др. [34] определяют **риски** как «потенциально неблагоприятные обстоятельства, которые могут повредить процессу разработки и качеству продуктов». Риски — каждодневные события, которые сопровождают любую человеческую деятельность. Поэтому неудивительно, что управление рисками — отдельная отрасль теории управления.

Управление рисками — операция принятия решения, которая оценивает воздействие рисков (неопределенностей) на решения. Она оценивает распределения возможных проектных результатов по отношению к вероятностям достижения этих результатов. Приняв приемлемый уровень допущения риска, она оценивает вероятность, с которой произойдет желаемый результат.

Управление рисками проектов использует то же самое мышление, которое применяется в исследованиях рисков финансовых портфелей. Простой метод финансового планирования состоит в том, чтобы взять начальный портфель и оценить будущий доход, применяя средние нормы процентных ставок на возвращение активов. Использование средних возвращаемых величин может дать в лучшем случае средние результаты. Лучшие результаты могут быть получены путем использования теории вероятностей, принимая во внимание различные шансы. Шансы определяются различными функциями распределения вероятности, а диапазон возможных результатов вычисляется с помощью моделирования. Управление рисками проекта использует подобные подходы.

Полагая, что желаемые проектные результаты определены, типичные шаги анализа риска следующие:

1. Идентифицировать риски и представить их, используя диапазоны возможных величин. Это делается заменой значений риска выбранной функцией распределения вероятности типа распределений Пуассона, равномерного, хи-квадрат, треугольного и т. д.
2. Использовать моделирование типа метода Монте-Карло или Латинского гиперкуба, чтобы получить возможные результаты и вероятности этих результатов.
3. Принять решение на основе графических результатов (гистограммы, интегральные кривые и т. д.), использования персональных суждений и приемлемого уровня допущения риска.

Инструментальные средства, которые объединяют управление проектом с управлением рисками, часто реализуются как дополнения к обычным инструментальным средствам управления проектами (типа Microsoft Project), дополнения к электронным таблицам (типа Microsoft Excel) или как приложения БД (то есть, например, для Microsoft Access). Например, ряд программных продуктов, называемых @Risk от Palisade [75], состоит из дополнений к Microsoft Project и Excel. С другой стороны, Risk Radar™ — продукты от ICE (Integrated Computer Engineering — Интегрированная компьютерная разработка) [40] являются приложениями к Access.

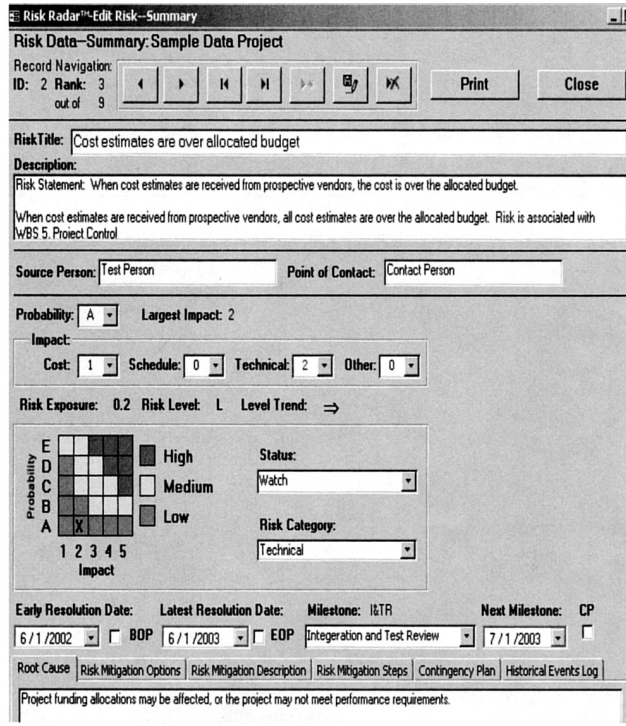


Рис. 3.7. Risk Radar™: Входные данные рисков
Источник: Integrated Computer Engineering [40]

Рис. 3.7 иллюстрирует окно входных данных Risk Radar™ для определения риска [40]. Каждый риск определяется таким образом и помещается в БД Microsoft Access перед моделированием анализа риска.

Рис. 3.8 показывает выходные результаты моделирования методом Монте-Карло, использующего @Risk [75]. Моделирование проводилось путем замены сомнительных значений из электронной таблицы переменных риска функциями распределения вероятности, представляющими диапазоны возможных величин. Окончательные графики показывают распределения возможных результатов и вероятности получения этих результатов. Графики дополнены итоговой статистикой. Для дальнейшего динамического анализа имеются панели вероятности и скользящие разделители для ответа на конкретные вопросы, типа «Каковы шансы, что число ошибок в коде будет превышать некоторую величину?»

3.2. Инструментальные средства моделирования систем

Как было отмечено в разделе 1.1.5, программная инженерия сродни моделированию. Следовательно, диапазон *инструментальных средств моделирования систем* охватывает все средства, которые поддерживают инженеров ПО

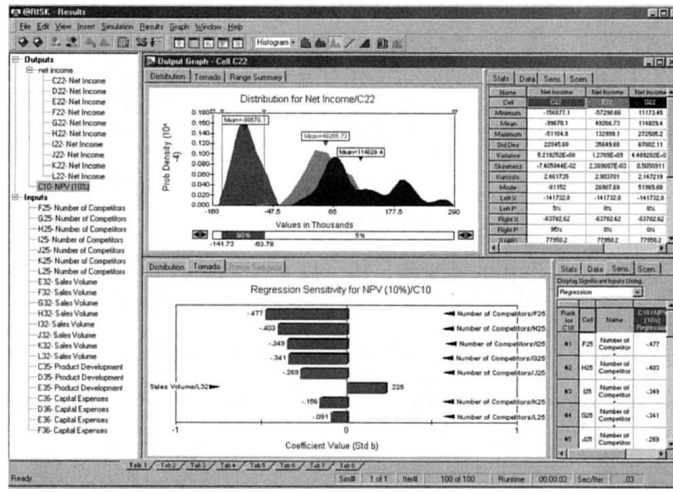


Рис. 3.8. @Risk: результаты моделирования
Источник: образ экрана из www.palisade-europe.com, перепечатано с любезного согласия Palisade Corporation

в задачах разработки — от анализа через проектирование к реализации. Они в значительной степени обеспечивают задачи **визуального моделирования** на языке UML. Средства визуального моделирования обычно называются CASE-средствами. Хотя формально понятие моделирования ПО включает само программирование, интегрированные среды программирования рассмотрены отдельно в разделе 3.3.

Современные инструментальные средства моделирования систем — средства рабочей группы, которые обеспечивают многих разработчиков доступом к распределенному хранилищу продуктов разработки, импортом продуктов в локальные рабочие станции, работой на этих продуктах и возвращением их назад в хранилище. Аспекты взаимодействия инструментальных средств моделирования систем рассмотрены в разделе 3.4.

Инструментальные средства моделирования систем сосредоточены на анализе требований и стадиях проектирования системы. Они обычно включают возможность как формирования исходного кода (прямое проектирование) на основании моделей, так и обратное проектирование моделей, исходя из кода. Одной из основных особенностей инструментальных средств моделирования является урегулирование зависимостей между требованиями пользователей (выраженных в виде текстовых предложений) и представление концепций визуального моделирования, отображающих эти требования.

Визуальное моделирование использует унифицированный язык моделирования (Unified Modeling Language — UML) — глава 2. Он вполне достаточен для всех задач моделирования, связанных с бизнес-требованиями, состоянием системы, ее поведением и динамическими изменениями состояния. Другие проблемы моделирования, типа моделирования БД или проекта, использующего Web-технологии, вероятно, потребуют специализированного языка моделирования или специально разработанного варианта UML (UML-профиля).

3.2.1. Управление требованиями

Требования — это текстовые предложения в пределах технического задания. Обычно имеется одно техническое задание на один сценарий использования — use case (раздел 2.2.2). Требования определяются на различных уровнях абстракции, и между ними имеется иерархическая согласованность. Описательное имя сценария использования также является требованием.

Инструментальное средство моделирования системы должно облегчить сотрудничество коллектива, обеспечить способы написания технических заданий, идентификацию различных категорий требований в документах, управление их изменениями и формирование требований, доступных всем разработчикам. Универсальный доступ к требованиям разработчиков подчеркивает важность этих требований в жизненном цикле разработки. Они используются аналитиками, чтобы выполнить анализ модели, проектировщиками, чтобы определить структуру системы и детальную спецификацию проекта, испытателями, чтобы разработать контрольные примеры, и менеджерами проекта, чтобы разработать планы и бюджет.

Рис. 3.9 содержит фрагмент технического задания, созданного с помощью IBM RequisitePro [86]. Это инструментальное средство расширяет Microsoft Word операциями, необходимыми для создания и управления различными

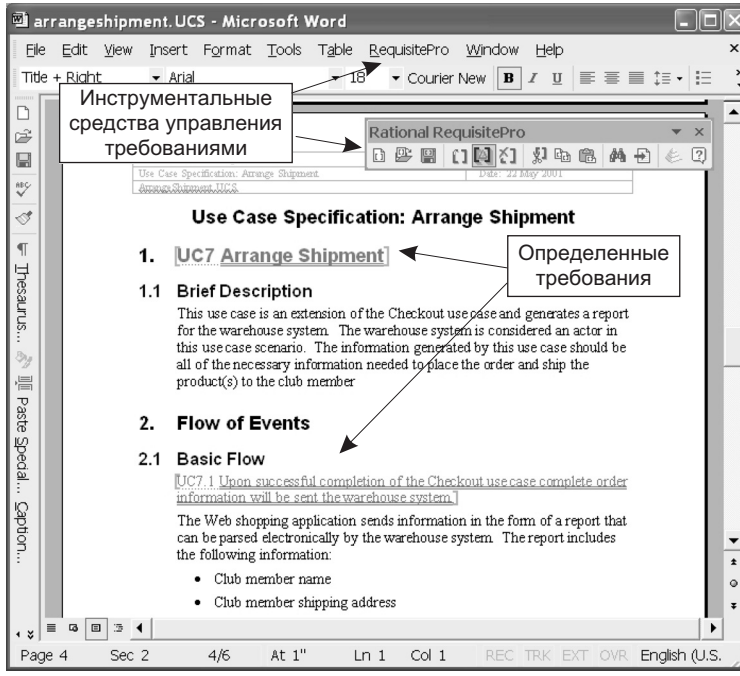


Рис. 3.9. IBM Rational RequisitePro: документ требований

Источник: образ экрана *IBM Rational Requisite Pro*, перепечатано с разрешения *Rational Suite Tutorial, Version 2002.05.00,* © Copyright 2002 by International Business Machines Corporation. All Rights Reserved; образ экрана Microsoft® Word, перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

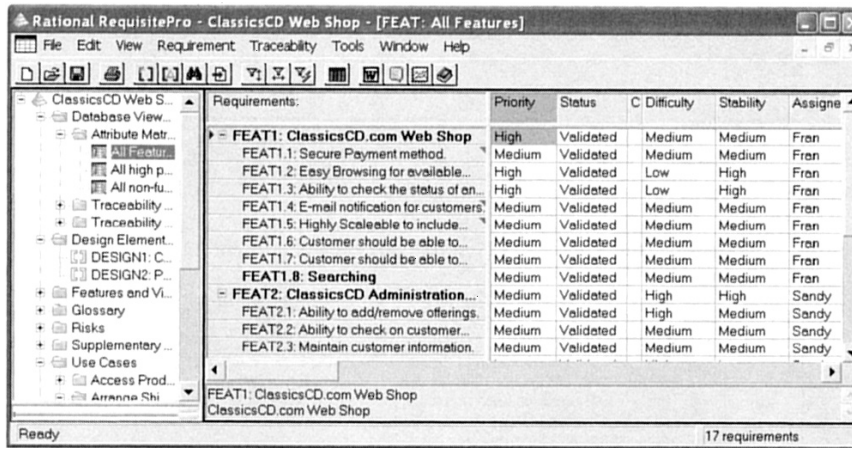


Рис. 3.10. IBM Rational RequisitePro: свойства требований
Источник: образ экрана *IBM Rational Suite*, перепечатано с разрешения *Rational Suite Tutorial, Version 2002.05.00,* © Copyright 2002 by International Business Machines Corporation. All Rights Reserved

уровнями требований в пределах текстового документа. Специальные операции RequisitePro доступны из меню и панели инструментов. Документ на рисунке определяет два требования — UC7 и UC7.1.

Требования, определенные в техническом задании, размещаются в *хранилище* инструментального средства моделирования. Они могут быть отражены на экране и модифицированы разработчиками. Изменения названий требований автоматически отражаются в техническом задании и наоборот. Инструментальное средство должно поддержать навигацию между связанными требованиями и между требованиями и другими продуктами моделирования.

Типичный способ показа требований для манипуляций — формат электронной таблицы. Этот знакомый всем формат позволяет легко задавать значения различных свойств (атрибутов). Свойства требования определяют такие характеристики, как статус требования, трудность его реализации, стабильность (то есть, может ли оно изменяться), кто владеет этим требованием, кем оно назначено, когда оно было создано и/или обновлено. Рис. 3.10 показывает список требований высокого уровня, называемых особенностями, и их свойства.

Желательно представлять различные характеристики требований, используя цвет. Рис. 3.11 показывает представление требований в виде электронной таблицы в пакете DOORS фирмы Telelogic [104]. Открытое выпадающее меню показывает, какой уровень риска (High — высокий, Medium — средний или Low — низкий) может быть назначен требованию. Каждый уровень риска закодирован цветом, и названия/определения требований показаны в цветах своего риска.

Интеграция управления требованиями с визуальным моделированием UML должна поддерживать двунаправленные ассоциации между требованиями (на различных уровнях) и визуальными элементами (продуктами) модели. Ассоциации между требованиями и сценариями использования устанавлива-

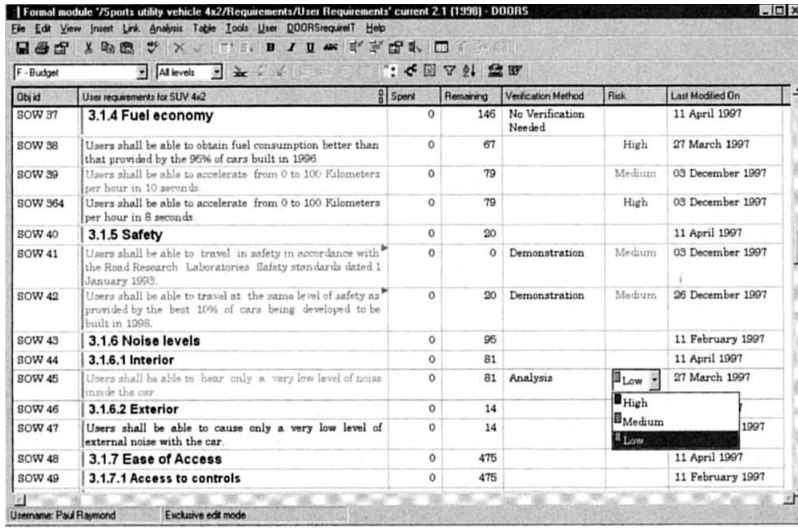


Рис. 3.11. DOORS: электронная таблица управления требованиями
Источник: образ экрана DOORS® из www.telelogic.com, перепечатано с любезного согласия Telelogic UK Ltd., Copyright © 2004 Telelogic AB

ются на ранних стадиях разработки. На рис. 3.12 показано диалоговое окно инструментального средства IBM Rational Rose, которое связывает спецификацию сценария использования с техническим заданием и требованием UC7, представленным сценарием использования. Подобное диалоговое окно можно было бы показать и с противоположной целью, то есть если разработчик, работающий с RequisitePro, хотел бы связать техническое задание со спецификацией сценария использования в Rational Rose.

Управление требованиями покрывает остальную часть процессов жизненного цикла. Оно включает возможность связать требования (различных уровней иерархии) с элементами модели, которые реализуют эти требования.

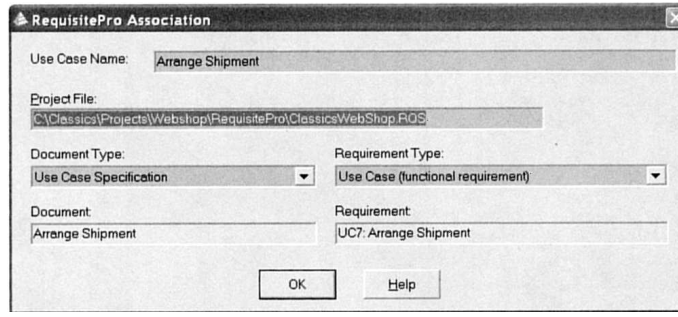


Рис. 3.12. IBM Rational Rose: управление ассоциациями между требованиями и сценариями использования
Источник: образ экрана *IBM Rational Suite*, перепечатано по разрешению из *Rational Suite Tutorial, Version 2002.05.00,* © Copyright 2002 by International Business Machines Corporation. All Rights Reserved

Интеграция требований в модель и ее различные элементы (сценарии использования, классы, компоненты, методы и т. д.) усиливают ценность проекта. Этот тип инструментального средства — «*все в одном*» — старается включить управление требованиями во все основные процессы жизненного цикла ПО от моделирования требований до формирования ПО (прямое проектирование).

Интеграция требований и других элементов моделирования требует умелого представления моделей на различных уровнях детализации и четкой навигации между элементами. Имея возможность создания в проекте большого количества диаграмм и элементов модели, легко потеряться в «пространстве проекта». Те, кто обращал внимание на изменения в программе «Проводник», мог бы заметить особенность *представления пиктограмм*, которая позволяет пользователю помещать картинки на папки как напоминание об их содержании. Представление пиктограмм также формируется автоматически для папок, которые содержат изображения, показывая уменьшенные рисунки на изображениях папок. Пользователи не должны просматривать папку, чтобы увидеть ее содержание, так как вид пиктограммы уже дает хорошее представление об этом. Подобный подход может быть включен в инструментальные средства моделирования, чтобы избежать ненужного просмотра диаграмм и просто напомнить проектировщику относительно содержания диаграммы.

Кроме создания пиктограмм или использования подобных технологий существенно может улучшить управляемость системы группировка связанных требований и элементов модели в UML-пакеты (возможно, в пакеты различных уровней). На рис. 3.13 показано, как сценарии использования могут быть сгруппированы в пакеты и показаны графически, при этом одновременно они связаны с требованиями, размещенными в дереве окна браузера. Пример взят из Sparx Systems' Enterprise Architect [97].

3.2.2. Визуальное UML-моделирование

Инструментальные средства для визуального **UML-моделирования** имеют диапазон от простых (только графических) средств до средств на основе хранилищ, поддерживающих коллективную работу и позволяющих настраивать графические представления. Наиболее сложные средства имеют хранилище на основе коммерческой БД, обеспечивают полную интеграцию моделей, интерфейс к другим инструментальным средствам программной инженерии, поддерживают различные версии продуктов и моделей проекта, формируют код для различных языков программирования и БД и т. д. Все эти проблемы подробно рассмотрены в соответствующих главах этой книги.

Инструментальные средства **визуального UML-моделирования** весьма похожи на подход для реализации GUI (Graphical User Interface — графический пользовательский интерфейс). Они состоят из области графического моделирования и соответствующей панели инструментов, содержащей элементы моделирования. Окно навигационного браузера (типа проводника) помогает в навигации между различными диаграммами и концепциями моделирования в пределах проекта. Есть также окна документации, позволяющие вводить текстовые описания для элементов моделирования. Могут

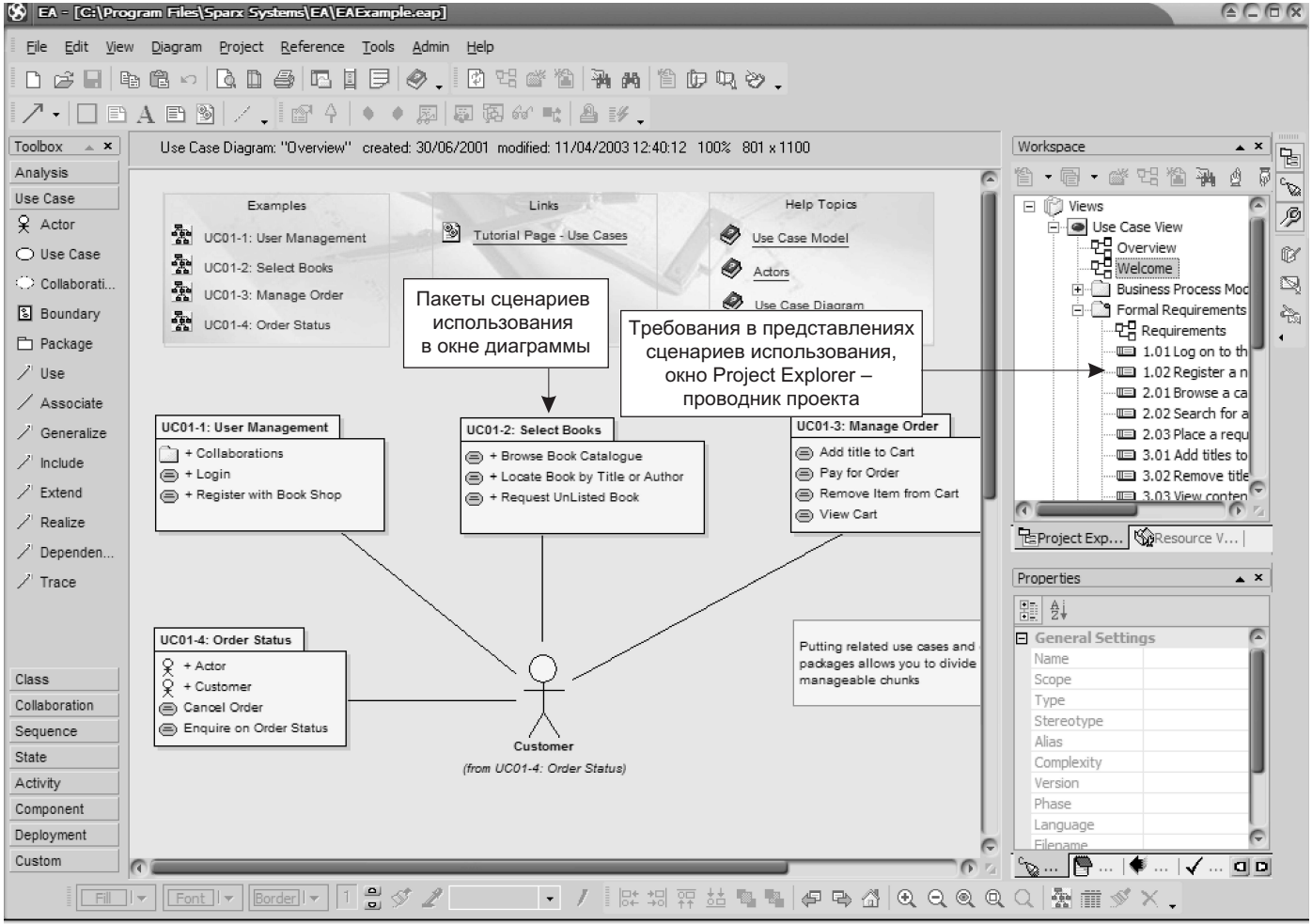


Рис. 3.13. Enterprise Architect: интегрированные требования и элементы модели

Источник: образ экрана Enterprise Architect из www.sparxsystems.com.au, август 2003 г., перепечатано с любезного согласия Sparx Systems Pty Ltd

быть запущены чувствительные к контексту окна спецификаций, чтобы войти в детальные спецификации (включая текстовые описания) для элементов моделирования. Все диаграммы и спецификации размещены в единственном хранилище для того, чтобы разработчики могли трудиться в многопользовательском режиме.

Рис. 3.14 — пример рабочего пространства UML-моделирования в IBM Rational Rose [85]. Окно отображает модули одного из небольших проектов этой книги — систему использования временного протокола — TLS (Time Logging System). Хотя это из рисунка и не очевидно, представленная диаграмма классов была получена обратным проектированием Java-программы в Rose-модель. В окне Class Specification (спецификация классов) отображена информация интерфейса `IFReader` (интерфейс «чтение» пакета `foundation` — см. разделы 9.1.6, 9.2.2), видимого в верхней правой части диаграммы. В окне Operation Specification (спецификация операций) находится информация для операции `readEmployee()` (чтение информации о служащем) интерфейса `IFReader`.

Рис. 3.15 подтверждает, что GUI-аспект разных инструментальных средств моделирования UML очень похож. Все инструментальные средства имеют навигационный браузер, перемещающий область модели, и окна спецификации хранилища. Пример на рис. 3.15 представляет рабочее пространство визуального моделирования инструментального средства Poseidon фирмы Gentleware [33]. Poseidon основан на ArgoUML [4]. Интересная особенность Poseidon заключается в том, что оно обслуживает документацию проекта с помощью модели, находящейся в блокноте с закладками (в нижней части рабочего пространства). Эта технология позволяет использовать информацию хранилища одновременно с созданием графических элементов, без необходимости открывать отдельные окна хранилища, как это делается в Rational Rose.

В среде Poseidon видимость, понятность и возможность манипулирования диаграммой облегчается использованием разных цветов и окна обзора, чтобы оценить размер всей диаграммы. Окно обзора графиков видно в левом нижнем углу рис. 3.15. Оно имеет две закладки: для «вида с высоты птичьего полета» и для выбора по приоритетам элементов моделирования. Окно обзора — характерная помощь при работе с большими моделями.

3.2.3. Формирование отчетов

Визуальное моделирование доминирует над другими видами операций по разработке ПО, оно стоит на одном уровне только с программированием и тестированием. Эти виды операций также в большинстве случаев ориентированы на коллективную работу. Поэтому необходимо, чтобы визуальные модели и связанная с ними документация были легко доступны и распределялись среди всех заинтересованных сторон проекта, а не только среди разработчиков. Это та область, где удобно использовать **формирование отчетов**.

Отчеты могут быть сформированы на основе различных шаблонов и с использованием различных форматов файлов. Хорошие инструментальные средства имеют целый диапазон предопределенных шаблонов документов и допускают также создание шаблонов, определяемых пользователем. Шаблоны

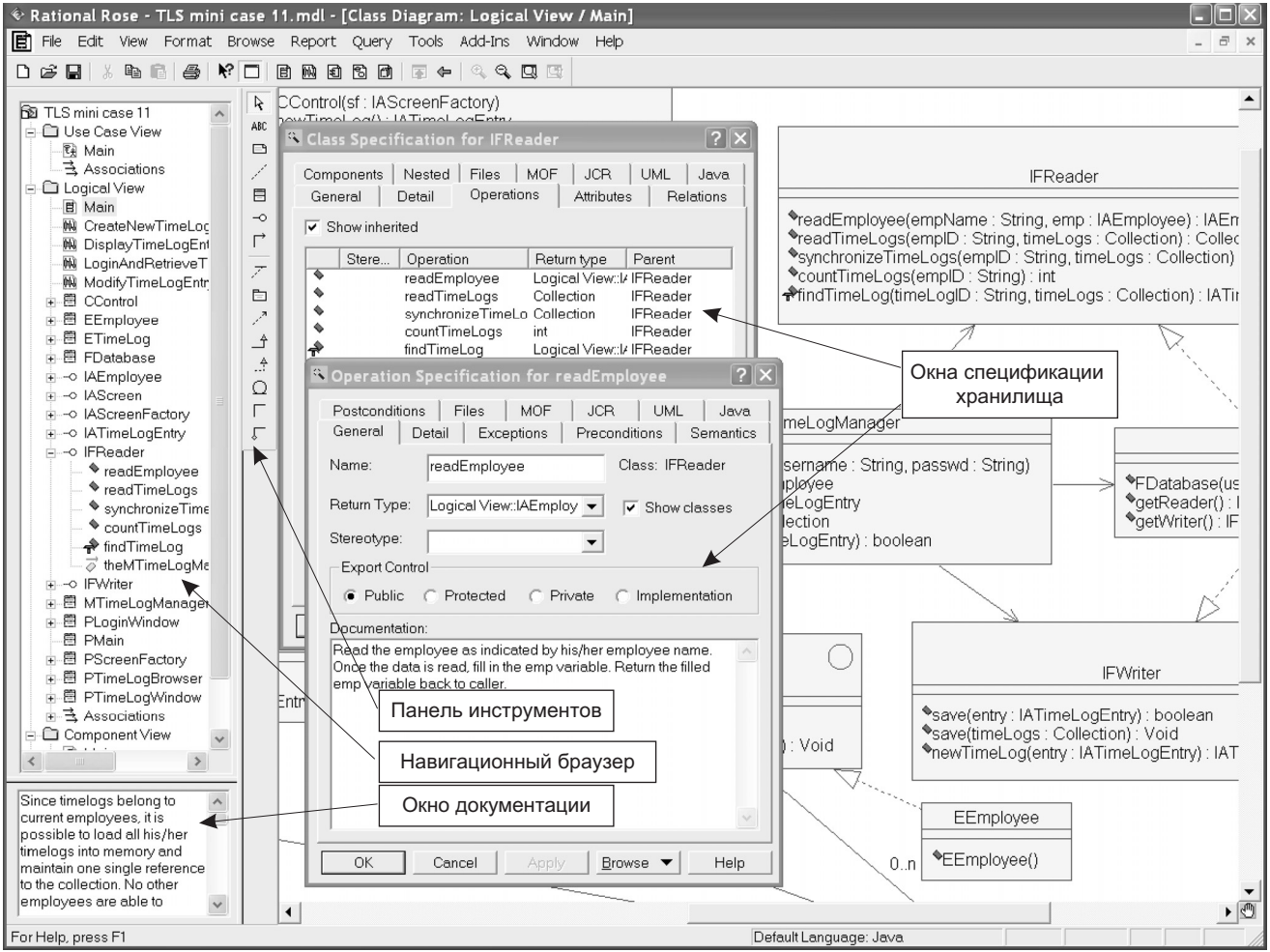


Рис. 3.14. IBM Rational Rose: рабочее пространство визуального моделирования

Источник: образ экрана *IBM Rational Suite*, перепечатано по разрешению с *Rational Suite Tutorial*, версия *2002.05.00,* © Copyright 2002 by International Business Machines Corporation. All Rights Reserved

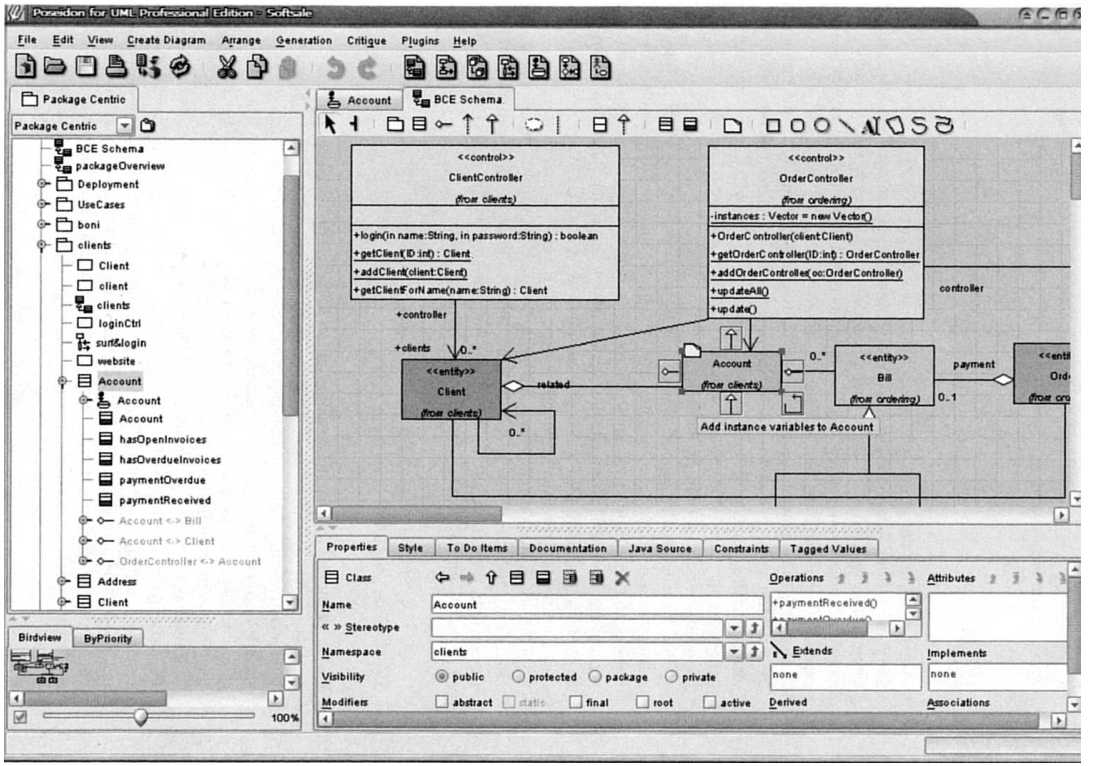


Рис. 3.15. Poseidon фирмы Gentleware: рабочее пространство визуального моделирования Источник: образ экрана Gentleware's Poseidon Sales Application model, август 2003, перепечатано с любезного согласия Gentieware AG

определяют структуру документов. Содержание может также быть определено пользователями, которые могут задать, какие модели и элементы моделирования должны быть включены в (или исключены из) процесс формирования отчета.

Форматы файлов, обычно поддерживаемые инструментальными средствами моделирования, включают HTML, Microsoft Word и/или pdf-файлы программы Acrobat. Поскольку существуют внешние инструментальные средства, легко конвертирующие эти три формата друг в друга, поддержка всех этих форматов в инструментальном средстве не существенна. Более важная проблема возникает, если нужно создать отчет для отображения/просмотра в Интернете или для печати. Инструментальное средство, формирующее HTML-файлы, должно иметь возможность при необходимости формировать HTML-отчеты, занимающие один экран, позволяя таким образом выполнять удобную печать.

Рис. 3.16 демонстрирует два отчета, созданные с помощью разных инструментальных средств моделирования: MagicDraw фирмы No Magic [73] и IBM Rational SODA [86]. Первый использует HTML-формат файла для просмотра пользователем. Второй использует формат Microsoft Word, удобный для печати.

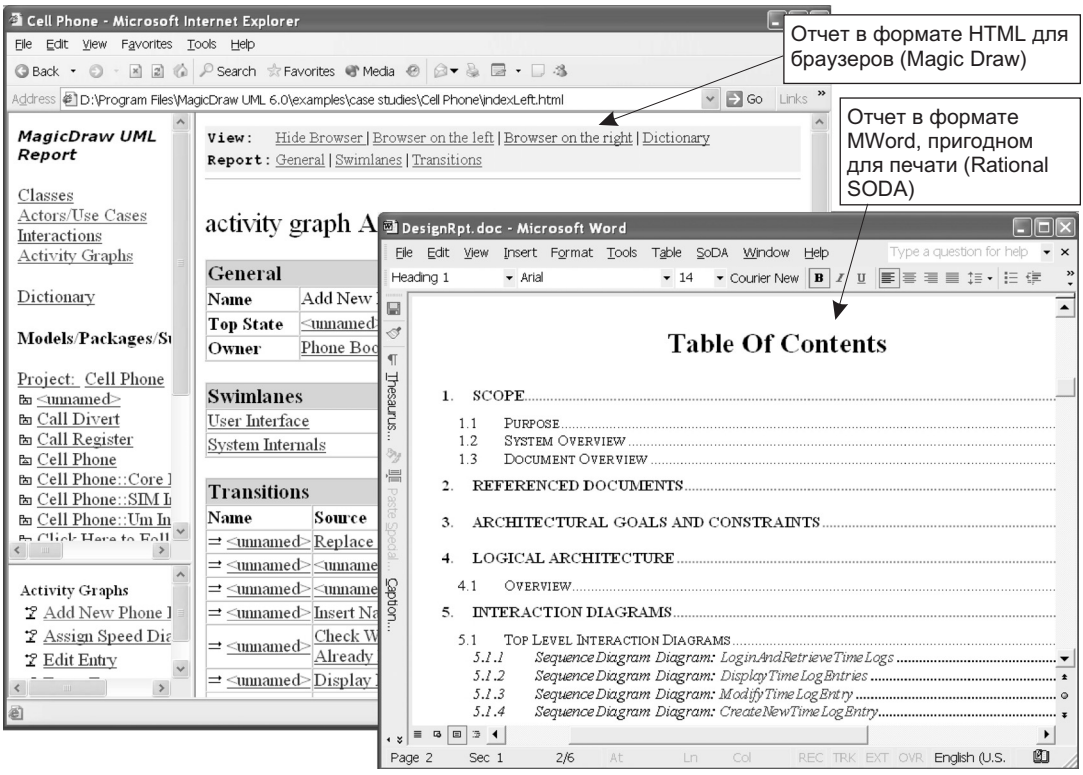


Рис. 3.16. No Magic's MagicDraw™ и IBM Rational Suite: отчеты моделирования
Источник: учебный пример MagicDraw [73], образ экрана No Magic's MagicDraw, © Copyright No Magic, Inc., перепечатано по разрешению. All Rights Reserved; образ экрана *IBM Rational Suite*, перепечатано по разрешению с *Rational Suite Tutorial, версия 2002.05.00,* © Copyright 2002 by International Business Machines Corporation. All Rights Reserved

3.2.4. Моделирование БД

Моделирование БД требует уникального набора возможностей. Хотя модели БД высокого уровня могут быть разработаны с использованием UML-диаграмм классов, требуются специальные логические/физические модели для отношений таблиц, а не классов. Специфические концепции БД, не известные в UML, включают также ссылочную целостность, индексы, хранимые процедуры, триггеры. Кроме того, имеются отличия в разных СУБД, которые должны быть рассмотрены, когда формируется код.

На рис. 3.17 показано рабочее пространство визуального моделирования инструментального средства моделирования БД PowerDesigner от Sybase [103]. Так же как и в инструментальных средствах UML-моделирования, рабочее пространство содержит окно навигационного браузера и окно рисования с плавающей палитрой используемых элементов моделирования. Диаграмма показывает таблицы реляционной БД и отношения ссылочной целост-

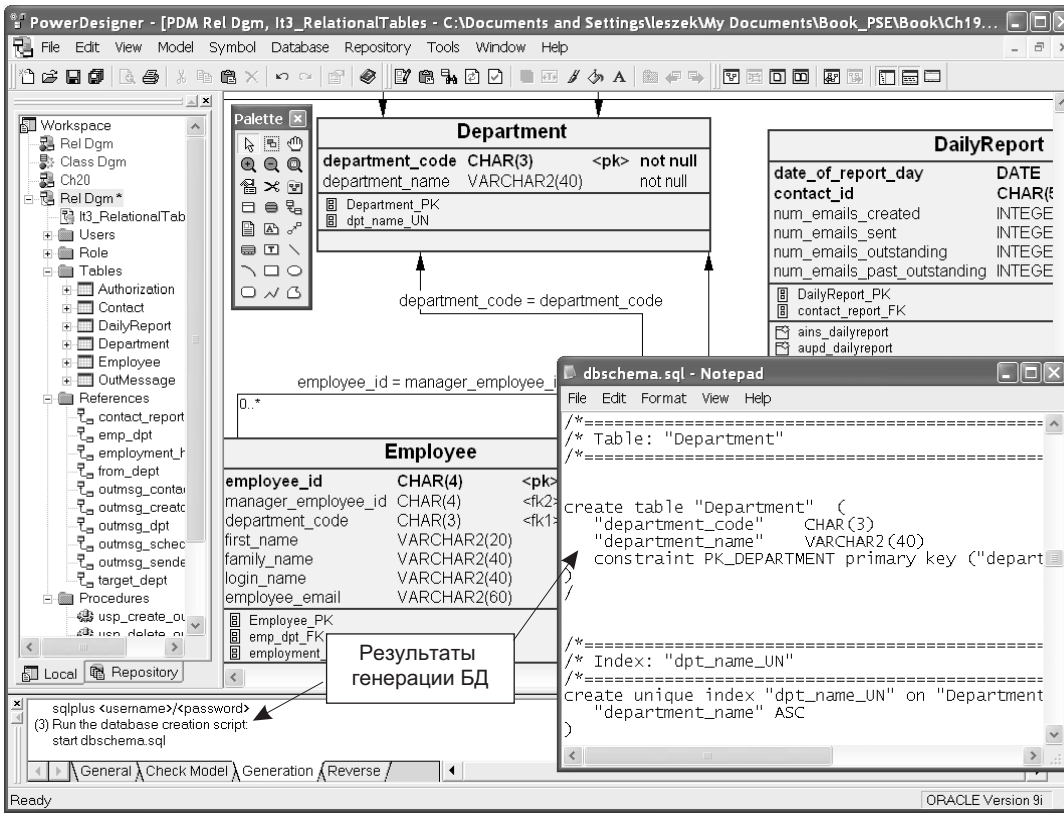


Рис. 3.17. Sybase PowerDesigner: рабочее пространство визуального моделирования Источник: образ экрана Sybase® PowerDesigner®, версия 9.5, из www.sybase.com, июль 2003, © Copyright 2002, Sybase, Inc

ности между ними. Инструментальные средства моделирования БД объединяются с различными СУБД. Можно создать код схемы точно так же, как код триггеров и хранимых процедур. Некоторые инструментальные средства поддерживают также формирование тестовых данных. Обратное проектирование от БД к графическим моделям также возможно.

Пример на рис. 3.17 относится к учебному примеру данной книги — Email Management (EM) — управление электронной почтой. Представленная диаграмма — часть модели, используемой в итерации 3 из учебного примера EM (часть 4 книги). Хотя здесь и не показано, PowerDesigner имеет также возможность UML-моделирования, которая позволяет формировать UML-диаграммы классов из логической/физической модели БД. Обратное отображение также возможно. Можно сформировать модель БД из UML-диаграммы классов (хотя в этом случае модель должна быть далее доработана, чтобы учесть специфические для БД свойства).

3.3. Интегрированные среды разработки

Среды программирования обычно называются **интегрированными средами разработки** (integrated development environments — IDE). IDE автоматизируют многие утомительные процессы программирования и позволяют разработчикам сосредоточиться на более важных проблемах. Стандартные возможности IDE включают завершение кодирования, поддержку стандартных приемов программирования (ревизия кода), интеграцию с инструментальными средствами конструирования типа Ant or make и т. д. Этот раздел, подобно остальной части книги, обсуждает среды программирования на основе Java. Однако многие выводы применимы также и к другим средам.

Хотя разные IDE и схожи в стандартных особенностях, все-таки имеются различия между ними. Некоторые различия касаются интеграции IDE с другими инструментальными средствами программной инженерии, в особенности, с инструментальными средствами моделирования (раздел 3.2) и с инструментальными средствами, поддерживающими коллективную работу (инструментальные средства управления изменениями и конфигурацией) (раздел 3.4). Сказывается также, что различные IDE-продавцы нацелены на различные рынки разработчиков. Разные IDE могут различаться следующими особенностями:

- интеграция с моделированием ПО;
- поддержка возможностей и характера прикладной разработки;
- интеграция с распределенными средами разработки и с инструментальными средствами управления изменениями и конфигурацией.

Так как UML становится стандартным языком моделирования, его поддержка и пригодность для IDE становится важной. UML-проекты обеспечивают программистов абстракциями, необходимыми, чтобы рассматривать ПО в процессе разработки с различных и более понятных ракурсов. Это важно, поскольку программисты нередко попадают в ловушку деталей отдельных функциональных возможностей и теряют полную перспективу.

Технологии Java выпускаются платформами. Понятие платформы используется, чтобы показать возможности и характер предметной области разработки. Главные виды **технологий Java** [45]:

- Java 2 Platform, Standard Edition (J2SE) — стандартное издание;
- Java 2 Platform, Enterprise Edition (J2EE) — издание уровня предприятия;
- Java 2 Platform, Micro Edition (J2ME) — микро-издание.

Типы IDE J2SE обеспечивают быструю прикладную разработку и интеграцию в цикле «план-разработка-внедрение». IDE в этой категории обеспечивают необходимый набор инструментальных средств для редактирования и компилирования программ, а также API (Application Programming Interface — программный интерфейс приложения) для написания, внедрения и запуска Java-апплетов и приложений.

Типы IDE J2EE обеспечивают широкий диапазон инструментальных средств, чтобы облегчить разработку ПО уровня предприятия. IDE в этой категории обычно объединяются с существующими серверами приложений и способны создавать конфигурации, необходимые для подключения приложе-

ния к серверу. Это основная технология для Web-сервисов. Она предназначена для реализации взаимодействующих бизнес-приложений.

Большинство IDE для обеспечения разработки встраиваемых приложений появляются на рынке в более современном варианте — J2ME. Разработка J2ME-приложения требует уникального набора инструментальных средств, чтобы обеспечить ограниченные в ресурсах и высоко оптимизированные среды реального времени в устройствах и изделиях потребителя типа мобильных телефонов, персональных цифровых помощников или автомобильных навигационных систем.

Открытые программные средства и их компоненты требуют использования распределенной среды программирования. Разработчики часто находятся в различных местах. Взаимодействие разработчиков должно централизованно управляться с помощью инструментальных средств, чтобы обеспечить синхронизацию работы и быструю разработку ПО. Чтобы помочь в этом, IDE должна быть объединена с инструментальными средствами управления изменениями и конфигурацией.

Границы между рыночными сегментами продукта исчезают, поскольку IDE обеспечивают более богатые комбинации особенностей. Уже неудивительно видеть среду программирования предприятия, которая имеет UML-поддержку наряду с возможностью разрабатывать встраиваемое ПО, или управляемую БД IDE, которая объединена с циклическим проектированием на основе Java.

3.3.1. Задачи стандартного программирования

Универсальная цель каждой IDE состоит в том, чтобы помочь разработчикам писать программы быстро и без ошибок. С этой целью IDE обеспечивает дружественное графическое рабочее пространство для всех типичных задач программирования. Данный раздел и его подразделы демонстрируют, как задачи стандартного программирования выполняются в среде Sun ONE Studio [102], ранее известной как Forte for Java IDE. Sun ONE Studio разработана на основе открытых программных средств, называемых NetBeans. Другие IDE имеют сходные возможности.

Рис. 3.18 показывает типичное IDE-рабочее пространство разработки, в данном случае это Sun ONE Studio. Чтобы начать программировать, в папку «source» (куда будут помещены все исходные файлы) и в папку «target» (куда будут помещены скомпилированные файлы) должен быть загружен ряд файлов проекта. До загрузки должны быть сконфигурированы некоторые настройки, чтобы сообщить IDE, как расположить/разместить файлы.

Написание программы

Современные IDE оборудованы мастерами, чтобы провести разработчика через различные задачи, включая создание Java-файлов. Рис. 3.19 показывает окно мастера для создания нового исходного Java-файла. Имя класса — `HelloWorld`. Класс будет помещен в пакет `pkg`. Объявление класса, которое основано на выборе опции «Access Level» (уровень доступа), указывает, что это класс `public` (общедоступный).

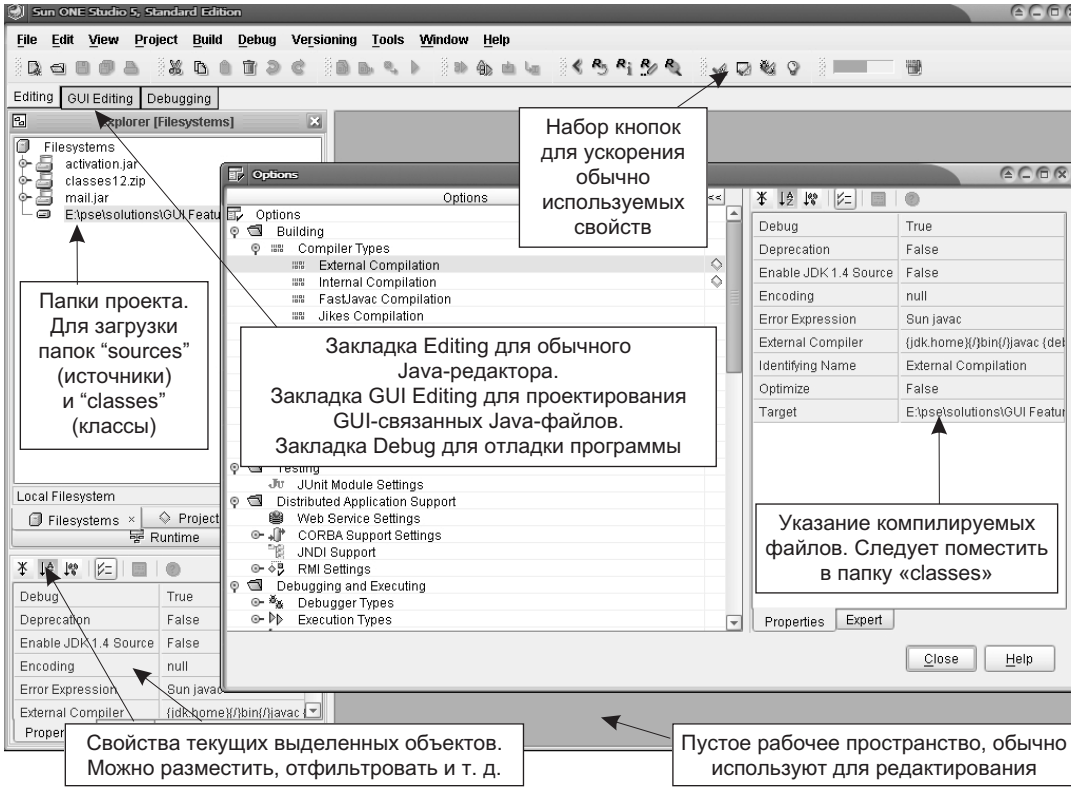


Рис. 3.18. Sun ONE Studio: рабочая среда разработки
Источник: Sun Microsystems, Inc

Рис. 3.20 представляет результирующий Java-файл с незаполненными методами, готовыми для редактирования программистом. Стандартный текстовый редактор имеет следующие особенности:

- Нумерация строк, которая помогает программистам понять, где они находятся, и в случае ошибки быстро перейти к нужному номеру строки.
- Подсвечивание синтаксиса, которое помогает программисту определить, какое слово является ключевым, переменной, комментарием, строкой, пробелом и т. д. (обычно настраиваемое использование цвета).
- Индикация, был ли конкретный файл изменен или нет. Это устраняет необходимость переключиться через ряд файлов, чтобы выяснить, было ли что-то изменено в них.
- Быстрая навигационная панель, помогающая переключаться между методами и переменными.
- Индикация текущего положения курсора (номер строки и номер колонки) и другие индикации типа режима «`ins`» (режим вставки) по сравнению с режимом «`ovr`» (переписать).

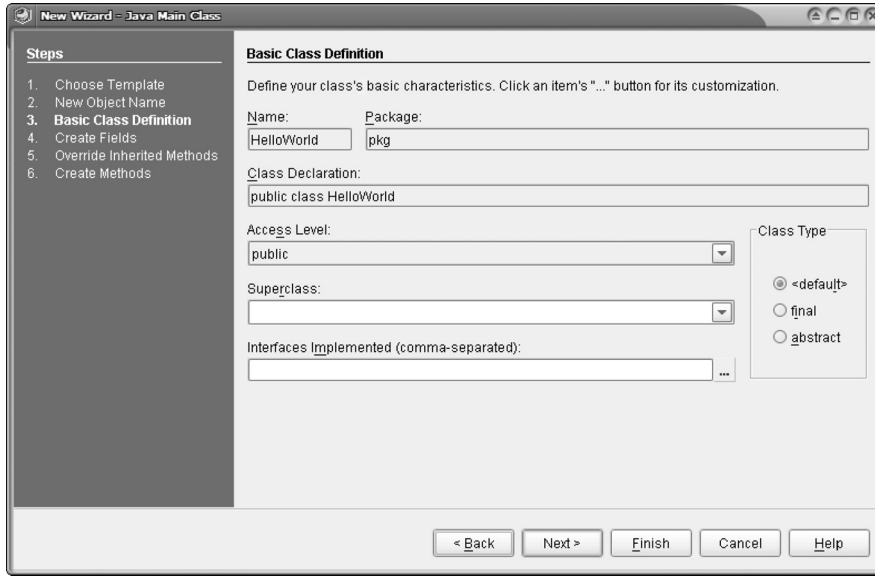


Рис. 3.19. Sun ONE Studio: мастер создания файла
Источник: Sun Microsystems, Inc

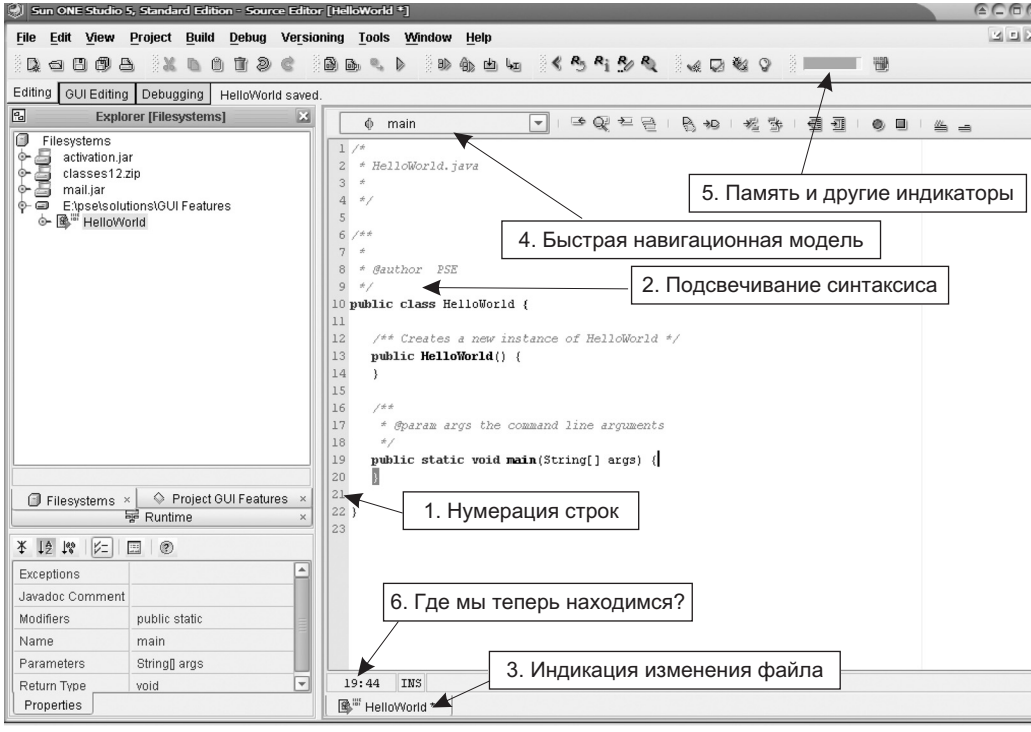


Рис. 3.20. Sun ONE Studio: Java-файл с незаполненными методами
Источник: Sun Microsystems, Inc

Редактирование исходного Java-файла легко осуществляется при наличии ярлыков всплывающих инструментальных средств, помогающих программистам в изменении путей работы (рис. 3.21). Ярлык инструментального средства может сообщить, какие методы в текущем запущенном объекте являются доступными (1). Рис. 3.21 показывает, что программист находится в `System.out` и ему доступно множество методов `println()` — распечатать строку (здесь имеется ряд перегружаемых версий `println()` с различными параметрами). Контекстное окно может показать описание текущего метода (2). Во время редактирования порой полезно привлечь внимание программиста к тому, что кое-что отсутствует, типа факта, что вышеупомянутый метод не имеет признака «`javadoc`» (документ Java) и поэтому он не будет иметь надлежащей документации, когда выполняется `javadoc` (чтобы сформировать документацию для проекта).

Как ожидается, другие средства редактирования также будут доступными. Инструментальное средство должно иметь возможность делать следующее:

- Автоматическое заполнение соответствующих аргументов метода, как только программист решит, какой метод выбрать. Например, если программист выбирает из списка `println(boolean)`, то инструментальное средство пробует определить, имеется ли готовая доступная переменная типа `boolean` (логический), которая может использоваться как параметр.
- Заполнение имени метода, как только программист выберет метод. Это освобождает программиста от печатания имени метода.

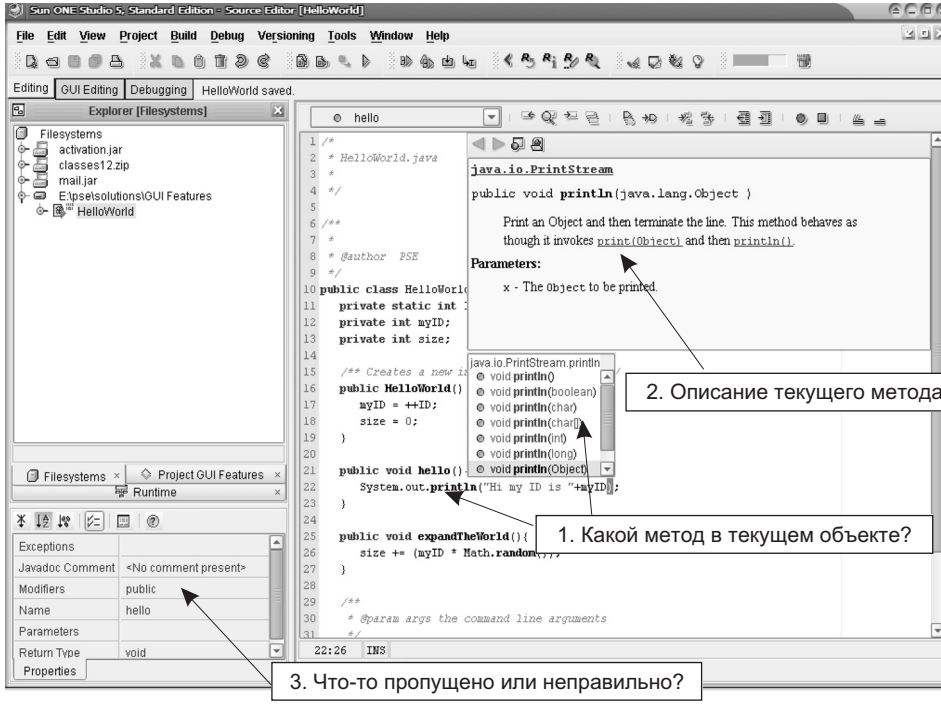


Рис. 3.21. Sun ONE Studio: облегчение редактирования
Источник: Sun Microsystems, Inc

- Автоматическое устранение ненужных названий методов из выпадающего списка. Например, если программист напечатает `System.out.pri`, тогда нет никакой необходимости показывать `flush()`, `clone()`, `toString()` и другие методы, коль скоро они не начинаются с последовательности «`pri`».
- Автоматическое перемещение курсора в ближайшее пустое место, которое требует внимания программиста. Например, курсор должен быть перемещен в позицию пустого параметра, который должен быть напечатан программистом. То есть, если программист выбирает `println(object)` из выпадающего списка, то курсор должен быть перемещен в `System.out.println(|)` (`|` указывает новое положение курсора).
- Формирование выпадающего контекстного окна, чтобы предложить список методов или переменных, доступных в текущем рабочем пространстве (текущие файлы или любые разрабатываемые файлы). Это весьма важно, так как порой программист должен обратиться к методам, созданным в других классах, и неудобно переключаться туда и обратно, чтобы найти имена методов или их сигнатуры и т. д.
- Обеспечение подсвечивания скобок, чтобы указать программисту, согласованы ли они. Подсвечивание должно работать для скобок функций (то есть, «`functionName()`») и любых других скобок, типа «`[]`».
- Указание синтаксических ошибок в написанном программистом тексте.

Выполнение программы

Как только программа будет успешно скомпилирована и связана с библиотеками, ее можно выполнить в рамках IDE (рис. 3.22). IDE должна поддерживать выполнение программы, представляя выход программы, обеспечивая (если необходимо) возможность пользователю задать вход программе, и указывать, работает ли программа или уже завершена. Ничто так не раздражает, как ожидание в течение долгого времени завершения неработающей программы, если нет никакого признака, работает ли она вообще.

Отладка программы

Отладка должна быть доступной в IDE. Недостаток функций отладки означает кошмар для разработчиков, так как они должны будут положиться на набор операторов `println()`, чтобы получить приблизительный результат (не говоря уж о времени, потраченном впустую, чтобы напечатать операторы `println()` и удалить их, когда они не будут больше нужны).

IDE-редактор обычно имеет возможность интеграции с отладчиком, что позволяет установить точки останова (отладчик будет делать паузу на точке останова во время выполнения программы) и выражения переменных (значение переменной или выражения показывается во время выполнения программы) — рис. 3.23. Простое двойное нажатие мышью номера строки обычно устанавливает точку останова для данной строки. Нажатие элемента меню или кнопки на панели инструментов — другой способ установить точки останова или наблюдаемые переменные.

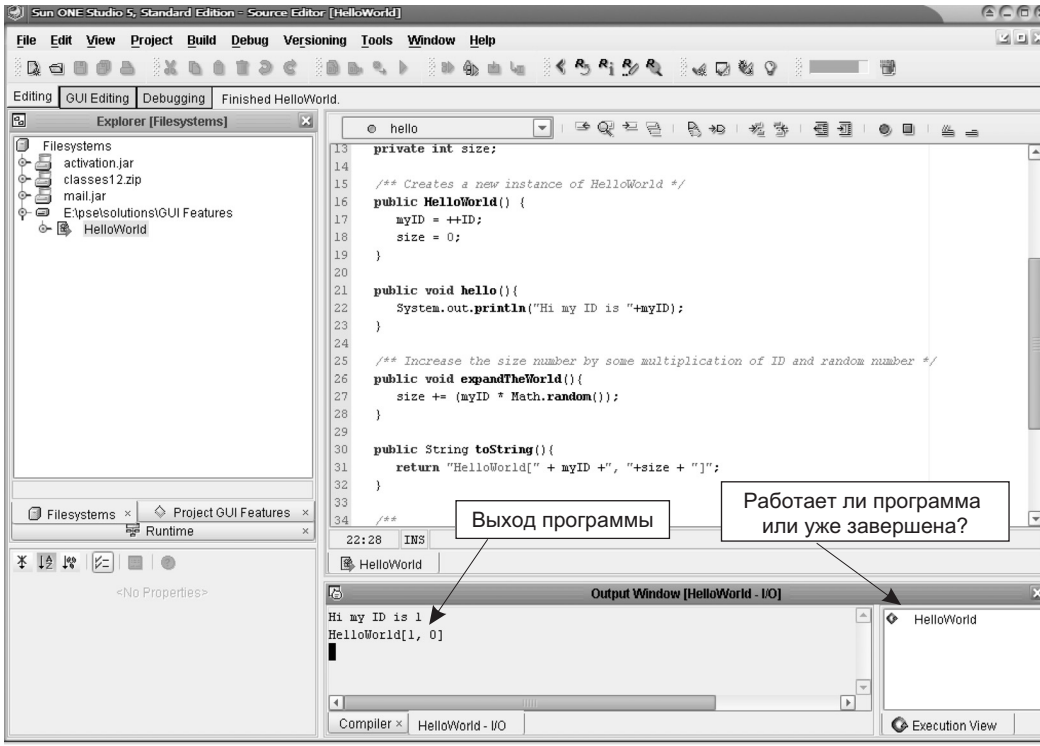


Рис. 3.22. Sun ONE Studio: выполнение программы
Источник: Sun Microsystems, Inc

Рис. 3.23 включает диалоговое окно с переменной «`size`» (размер), установленной для наблюдения во время выполнения программы. Если потребуется, то вместо переменной может использоваться сложное выражение. Например, `getsizeMult2()` — подходящее выражение для отладки, чтобы указать отладчику, что программист хочет получать информацию о результате выполнения метода `getSizeMult2()` в любое время, когда работает отладчик.

Другие особенности отладки включают:

- Наблюдение HTTP-запросов (Hypertext Transfer Protocol — протокол передачи гипертекстовых файлов) «`post`» и «`get`» от клиента, если отладчик запущен на Web-сервере.
- Проверка окна трассировки стека — графа иерархического вызова объектов.
- «Run to cursor» (запуск до курсора) — выполнение программы в отладчике и остановка, когда выполнение достигнет текущей позиции курсора.
- «Step Into» (шаг в...) — начать выполнение программы с заходом в тело текущего метода.
- «Step Out» (шаг до выхода) — быстрое выполнение до конца текущего выполняемого метода и переход к следующему вызову метода.

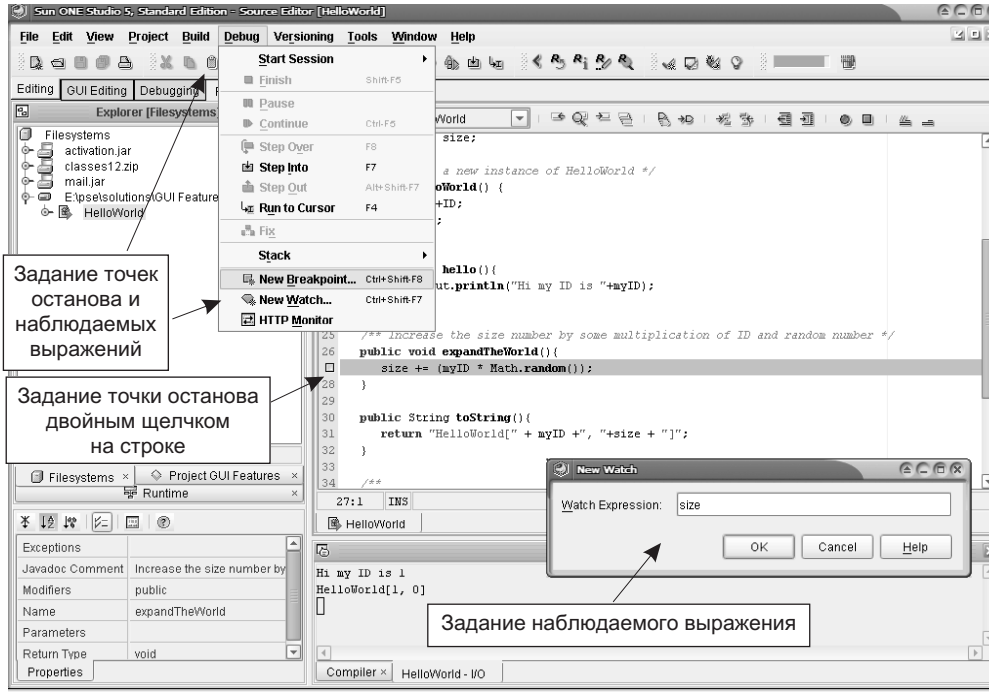


Рис. 3.23. Sun ONE Studio: установка программы для отладки
Источник: Sun Microsystems, Inc

- «Step Over» (шаг поверх...) — начать выполнение программы и «скачок поверх» выполняемого в настоящее время метода. Это ускоряет выполнение, если разработчик уверен относительно правильности выполняемого в настоящее время метода и не хочет его проверять.
- «Pause» (пауза) — сделать паузу в текущей сессии отладки и «Continue» (продолжение) — продолжить остановленную сессию. Это могло бы быть, например, полезно, если программа работает на основе потоков и в настоящее время имеется несколько потоков, конкурирующих друг с другом, и необходимо «заморозить» их на некоторое время.
- «Finish» (конец) или «Stop» (остановка) — выбросить программу из сессии отладки в целом.
- Расцвечивание строк важно, чтобы указать различные точки останова, текущую вызываемую строку и т. д.

Рис. 3.24 показывает результат отладки «Step into». Отладчик входит в активный метод (которым является `HelloWorld()`). Затем он останавливается на строке определения метода и ожидает следующую инструкцию от пользователя. Из этого рисунка можно сделать следующие выводы:

- Ясная индикация следующей выполняемой строки. Отладчик переходит на эту строку из оператора `new HelloWorld()`, не видимого на текущем экране.

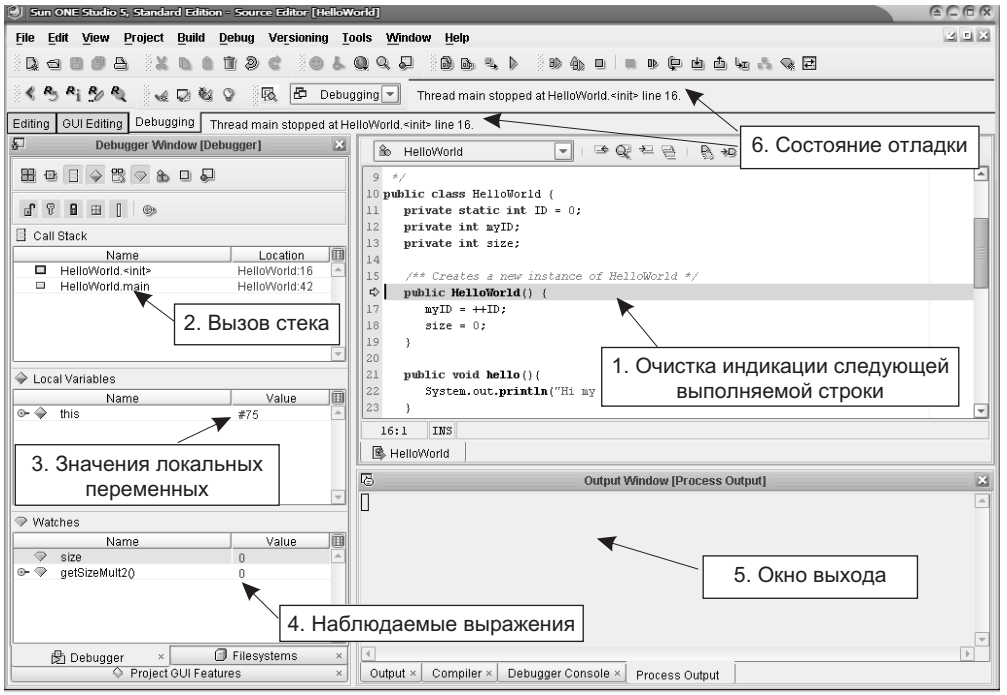


Рис. 3.24. Sun ONE Studio: отладка «шаг в...»
Источник: Sun Microsystems, Inc

- Ясная индикация стека вызовов (то есть иерархического характера вызываемых объектов) и другой информации (номер строки и т. д.).
- Некоторая информация относительно выполняющегося в настоящее время метода, каковы значения его параметров и т. д.
- Каковы значения для всех «наблюдаемых параметров»? Отображение того, имеют ли они значения или недоступны в текущем контексте.
- Обычно отображается выходное окно, чтобы видеть, что происходит с результатами программы.
- Некоторая другая информация, сообщающая разработчику о состоянии отладчика.

3.3.2. Интеграция с моделированием ПО

Объединение инструментальных средств программной инженерии особенно очевидно при включении возможностей **UML-моделирования** в IDE. Та же тенденция видна также и в другом направлении, но менее очевидна. Результат интеграции — среда коллективной разработки, облегчающая связь и сотрудничество, использующая синхронизированные графические модели и код. Разработчик может работать на модели и наблюдать автоматические изменения в коде или наоборот.

Популярное инструментальное средство, которое объединяет моделирование и кодирование, — Together ControlCenter фирмы Borland [14] (рис. 3.25). Это — полностью обеспеченное инструментальное средство, использующее диаграммы UML, которое поддерживает автоматическое прямое и обратное проектирование (циклическое проектирование) между кодом и моделью. Разработанная модель может быть проанализирована для ее соответствия образцу, например, с помощью моделей Gang of Four (GoF) [32]. Рис. 3.25 показывает UML-модель с редактором кода, где действия, выполненные на модели, немедленно отражаются в коде.

3.3.3. Разработка приложения предприятия

Среда формирования ПО квалифицируется как IDE предприятия, когда она определенно позволяет проектировать, разрабатывать и внедрять J2EE-приложения. J2EE-приложения находятся в диапазоне от разработки простых Enterprise JavaBeans (EJB) с Bean-компонентами Bean-Managed Entity Persistent (BMP) или Bean-компонентами Container-Managed Entity Persistent (CMP) до разработок JSP, Servlet и Web-сервисов.[1]

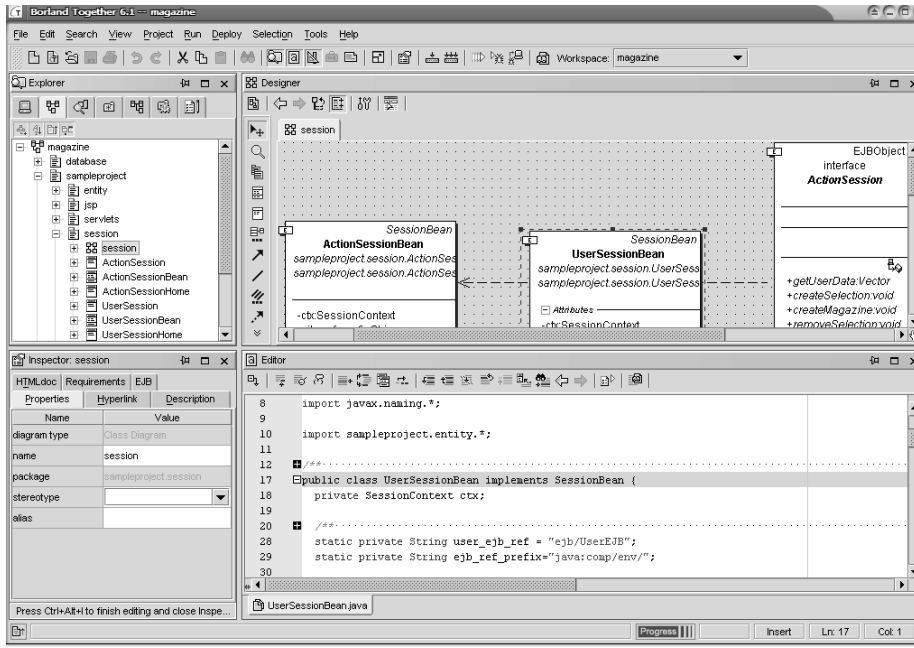


Рис. 3.25. Borland Together ControlCenter: интеграция UML-моделирования и программирования
Источник: образ экрана *Borland® Together® ControlCenter®*, пример (2003), www.borland.com, перепечатано по разрешению Borland Software Corporation

[1] Bean-компоненты, как мы будем называть их в данном переводе, или «бины», «бобы», как они порой называются на жаргоне программистов, представляют собой программные компоненты различных пакетов фирмы Sun, которые можно использовать в программах как некоторые целостные структуры, выполняющие определенные функции. (*Прим. перев.*)

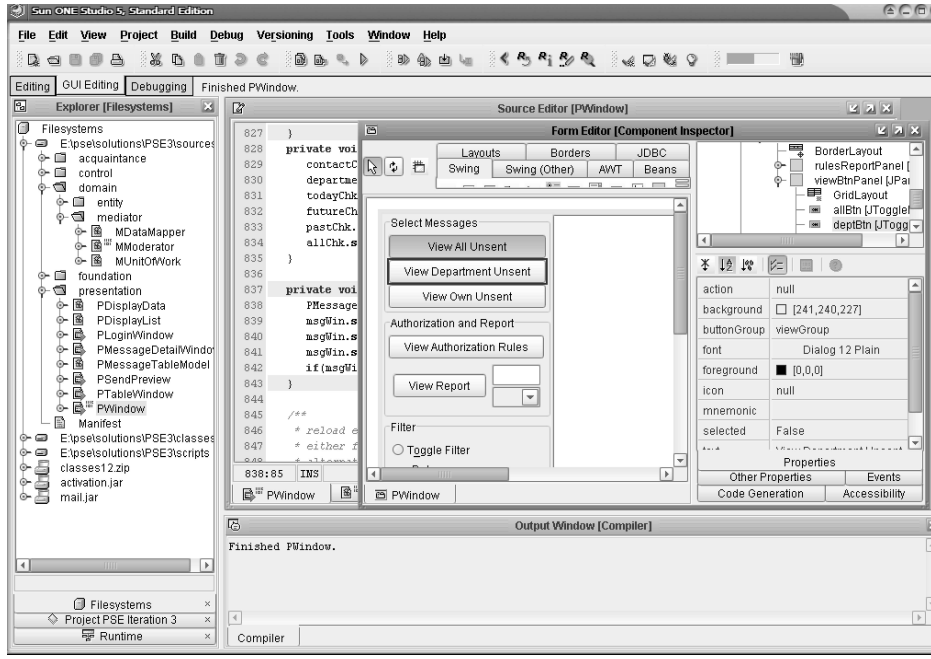


Рис. 3.26. Sun ONE Studio для J2EE-разработки
Источник: Sun Microsystems, Inc

Интеграция с сервером приложения (где будут помещены разработанные Bean-компоненты) — основная особенность этого типа IDE. Знания, которыми должны обладать разработчики, чтобы успешно создавать и внедрять J2EE-приложения, значительны. Как правило, разработка J2EE-приложения будет иметь следующую последовательность:

1. Создать компоненты проекта, необходимые для приложения (обеспечивают потребности приложений JSP, HTML, Servlet, BMP/CMP и т. д.).
2. Разработать «скелеты» компонентов с помощью «щелчков мышью» в мастерах.
3. Заполнить логику компонента.
4. Протестировать компонент.
5. Сформировать конфигурацию внедрения приложения.

Тестирование компонентов требует, чтобы IDE обеспечила или внутреннюю, или внешнюю поддержку тестирования. Для внутреннего тестирования IDE должна иметь сервер для тестирования приложения, оптимизированный по отношению к стандартным возможностям тестирования типа контроля переменных, отслеживания потоков и т. д. Внешнее тестирование предполагает у IDE наличия отдаленного отладчика и тестера, чтобы обеспечить выполнение приложения на существующем сервере приложения.

Учебный пример управления электронной почтой (Email Management — EM), обсуждаемый в этой книге, был разработан как J2EE-приложение уровня предприятия. Одним из основных инструментальных средств, используе-

мых на стадии программирования, было Sun ONE Studio. На рис. 3.26 показано рабочее пространство Sun ONE Studio, используемое во время GUI-редактирования класса `PWindow` (класс «окно» пакета `presentation`) учебного примера EM (часть 4 книги).

Ожидается, что IDE уровня предприятия поддержит различные шаблоны для быстрого выполнения работы, включая шаблоны для JSP, EJB, Java-файлов, Web-сервисов и т. д. Должны поддерживаться различные типы GUI. Изменения в GUI должны быть немедленно отражены в коде, обеспечивающем быструю прикладную разработку (Rapid Application Development — RAD) визуальных приложений. Возможность подключения БД должна быть JDBC-совместима и позволять обратное проектирование уже существующих таблиц БД в Bean-компоненты сущностей, которые можно использовать на сервере приложения.

3.3.4. Интеграция с бизнес-компонентами

Разработка J2EE-приложения может быть далее дополнена соответствующей поддержкой конструкций бизнес-компонентов. **Бизнес-компоненты** составляют средний ряд структуры многоуровневой системы. Это — ряд между GUI-приложением и данными приложения, находящимися в БД. Бизнес-компоненты лучше разместить на отдельном сервере — сервере приложения.

Одно такое J2EE-средство разработки, планирующее бизнес-компоненты — Oracle JDeveloper [74]. Оно поддерживает стандарт разработки Java, а также разработку приложений предприятий, которые будут развернуты не обязательно в Oracle-инфраструктуре. Оно прочно объединено с программным продуктом фирмы Oracle Business Components for Java (BC4J), который позволяет разрабатывать серверные приложения с БД (глава 22).

BC4J обеспечивает создание блоков бизнес-уровня, чтобы иметь возможность интеграции с любой JDBC-совместимой БД. Обратное проектирование таблиц БД — всего-навсего вопрос «щелчка мышью». Связь таблиц приложения (так называемые представления — Views) с таблицами БД сделано в XML-конфигурации файлов. Для связи с БД могут использоваться и JDBC-, и SQLJ-технологии. Во время разработки BC4J-компонентов JDeveloper использует свой внутренний Oracle Container for Java (OC4J) — подмножество Oracle Application Server, который управляет задачами связи EJB и БД.

Разработка серверного приложения с БД обычно выполняется в следующей последовательности:

1. Перепроектировать существующую структуру/таблицы БД (это включено в мастера для BC4J).
2. Сформировать ряд представлений, которые отображают таблицы БД.
3. Настроить представления, используя бизнес-логику.

Простой браузер приложения с BC4J-реализацией показан в главе 22 (рис. 22.2). Рис. 3.27 показывает шаг формирования BC4J из существующей схемы БД (в данном случае `PSE3`). После успешной связи с БД мастер будет использовать определения ее элементов: таблицы, представления, синонимы (псевдонимы) и снимки для формирования GUI-представлений (окон приложения).

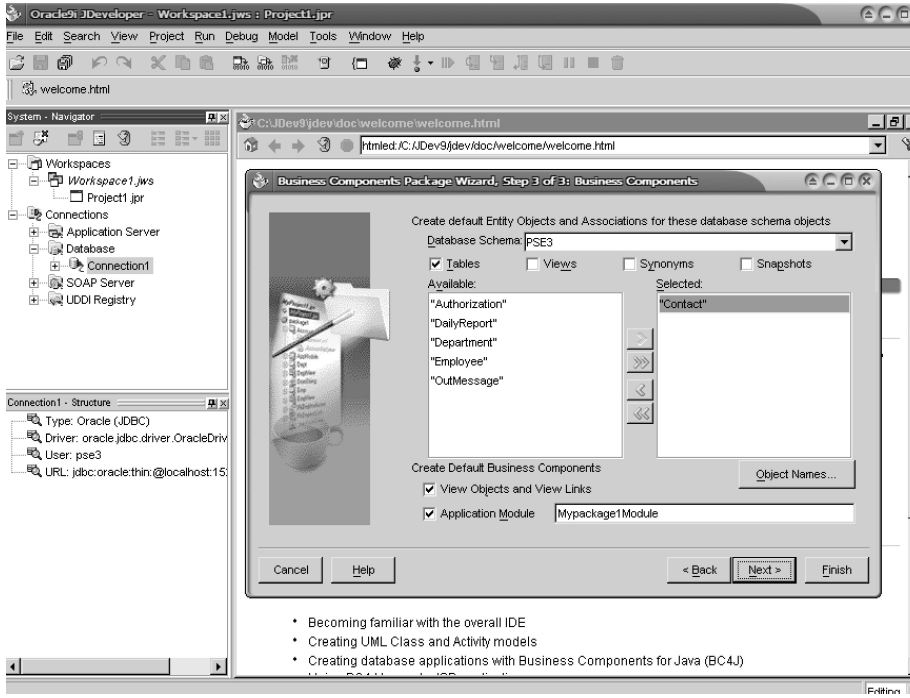


Рис. 3.27. JDeveloper в разработке BC4J-приложения
Источник: образ экрана Oracle JDeveloper из www.otn.oracle.com, перепечатано по разрешению Oracle Corporation

3.3.5. Интеграция с управлением изменениями и конфигурацией

Следующий раздел, раздел 3.4, обсуждает инструментальные средства управления изменениями и конфигурацией, необходимые для совместной разработки и сопровождения системы. IDE промышленного уровня должны поддерживать деятельность многих разработчиков, для чего они имеют встроенные средства коллективной работы. Способности **коллективной работы** часто достигаются связью с независимыми инструментальными средствами управления изменениями и конфигурацией. Как правило, интеграция настолько прозрачна, что IDE выглядит и работает подобно средству управления изменениями и конфигурацией. В некоторых случаях способности управления изменениями и конфигурацией — свойственная особенность IDE.

В кругах программистов инструментальное средство управления конфигурацией часто считается синонимом системы управления версиями (Version Control System — VCS). Несомненно, **управление версиями** — доминирующий компонент управления конфигурацией, но полноценное инструментальное средство управления конфигурацией охватывает продукты, моделируемые на всех стадиях жизненного цикла. Многие же VCS связаны только с программированием кода.

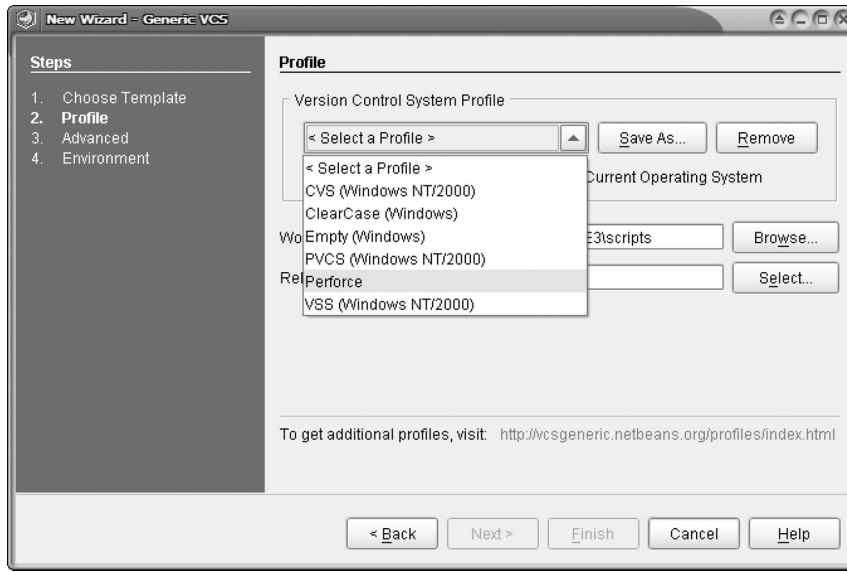


Рис. 3.28. Sun ONE Studio: интеграция с управлением версиями
Источник: Sun Microsystems, Inc

Одна из наиболее полных IDE, опирающаяся на открытые программные средства Eclipse [26], — WebSphere Studio Application Developer (WSAD) фирмы IBM [39] объединяет коллективное программирование, управление конфигурацией и управление версиями. Она интегрируется с Concurrent Version System (CVS) и объединяется с IBM Rational ClearCase LT (CCLT).

Как видно на рис. 3.28, Sun ONE Studio обеспечивает интеграцию с рядом VCS, одна из которых — Perforce [77]. Perforce и любая другая VCS работают на уровне проекта. Склад (или хранилище) определяет местоположение документов, управляемых с помощью Perforce. Пользователи могут соединяться с Perforce, чтобы добавлять, корректировать или удалять документы (программы). Система поддерживает состояние документов, импортируя их в рабочее пространство пользователей. Такие документы обычно размещаются в складе. Когда пользователь закончит работать с документом, он/она может пожелать синхронизировать (или экспортировать) документ, а следовательно, скорректировать хранилище.

Системы управления версиями обеспечивают графические средства «`diff`» (различия), чтобы выполнить сравнение между версией кода клиента и текущей *основной редакцией*, которая — в терминологии Perforce — является самой последней версией кода, доступной на сервере (в складе). Версия в рабочем пространстве клиента могла быть создана из текущей основной редакции или из более ранней основной редакции в зависимости от того, когда код клиента был импортирован из склада.

Рис. 3.29 показывает команду Perforce, выполняющую операцию «`diff`» между результатом пересмотра клиентом класса `MUnitOfWork` (единица работы) и версии, размещенной в хранилище. Результат «`diff`» показан в раз-

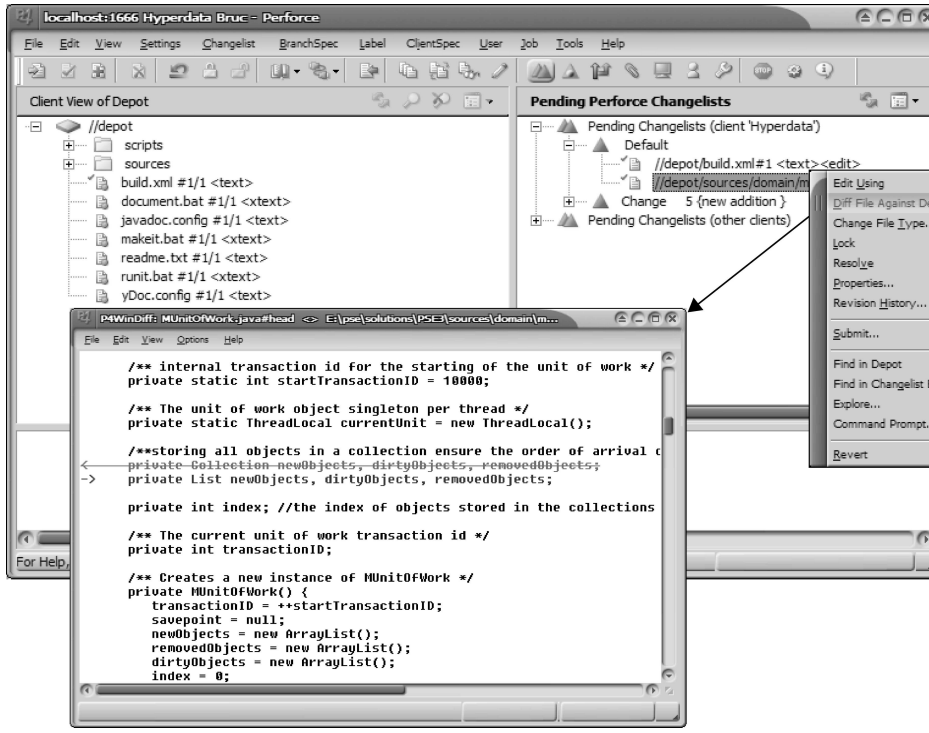


Рис. 3.29. Perforce: сравнение версий

Источник: образ экрана из www.perforce.com, перепечатано с любезного согласия Perforce Software, Inc

мещенном поверх других окне. Красная перечеркивающая линия, которая пересекает одну строку кода, указывает, что документ версии клиента имеет одну удаленную строку. Следующая строка (синяя и с указателем в виде боковой стрелки) — новый оператор в коде.

3.4. Инструментальные средства управления изменениями и конфигурацией

Хотя технически управление изменениями и конфигурацией различаются, на практике они всегда объединяются. **Инструментальное средство управления изменениями** записывает изменения, которые произошли в течение жизненного цикла ПО. Управление изменениями особенно заметно в цикле тестирования реализации, когда обнаруженные дефекты требуют изменений в коде. Однако запись только изменений без фиксации истории изменений практически не жизнеспособна. Требуется хранить следы различных версий продуктов разработки, включая версии одного и того же продукта, созданные различными разработчиками, с тем, чтобы старые версии можно было снова использовать, когда будет необходимо. **Управление версиями** — составная часть **инструментальных средств управления конфигурацией**.

Унифицированные операции управления изменениями и конфигурацией можно объяснить следующей типичной последовательностью шагов [85]:

1. Разработчик присоединяется к проекту, регистрируясь для использования инструментальных средств и хранилища проекта. Это создает **личное рабочее пространство** на автоматизированном рабочем месте разработчика, из которого разработчик получает доступ к **общему рабочему пространству** (БД) продуктов, доступному всему коллективу.
2. Личное рабочее пространство разработчика состоит из представлений разработки и потока разработки. *Представление* состоит из одной версии каждого продукта в рабочем пространстве разработчика. *Поток* (или конфигурация) — текущее множество версий, для которых конструкторские операции проанализированы инструментальным средством управления конфигурацией.
3. *Операции*, которые будут выполнены разработчиком, помещаются в хранилище с помощью инструментального средства управления изменениями. Они могут быть восстановлены разработчиком, чтобы создать список операций.
4. Разработчик *импортирует* продукты в личное рабочее пространство. Выполнив операции на импортированных объектах, инструментальное средство управления конфигурацией корректирует *массив изменений* (список измененных продуктов).
5. Разработчик *экспортирует* (измененные) продукты в общее рабочее пространство. Изменения в продуктах станут видимы коллективу после объявления их *поставленными* (delivered). Поставка изменений относится ко всему массиву изменений. Более сложное инструментальное средство позволило бы организовать сотрудничество разработчиков для обсуждения предложенных изменений с тем, чтобы можно было разрешить любые конфликты между версиями, поставленными различными разработчиками.
6. По решению руководства из массивов изменений, поставленных коллективом разработчиков, могут быть созданы новые *исходные данные* (или *конструкция*) — непротиворечивое множество изделий и продуктов процесса, которые определяют новую общую точку приложения для коллектива проекта. Новые исходные данные могут формироваться часто, даже ежедневно.
7. Исходные данные подвергаются контролю качества и проверке на обеспечение гарантий и могут быть *переведены* в ранг *рекомендуемых исходных данных*. Рекомендуемые исходные данные используются, чтобы *пересмотреть* набор задач разработчиков так, чтобы те могли работать в соответствии с текущими рекомендуемыми исходными данными.

3.4.1. Поддержка изменений

Одна из наиболее важных истин программной инженерии — это то, что изменениями проекта необходимо управлять, иначе изменения будут «управлять» проектом. Изменения вездесущи. Управление проектом в целом — это

тоже форма **управления изменениями**. Изменения в проекте могут быть как из-за внешних, так и внутренних проектных условий (бюджет, ресурсы, конкуренция и т. д.). Они могут возникнуть из-за изменений требований пользователей. Они возникают также из-за дефектов в ПО и необходимости совершенствования ПО (будущие возможности). Причины кишат повсюду.

Инструментальные средства управления изменениями обеспечивают обслуживание процесса изменений. Сам процесс изменений должен быть определен в технологическом процессе, который реализует инструментальное средство управления изменениями. Практически **технологический процесс** может быть смоделирован в виде диаграммы состояний (раздел 2.2.4). Технологический процесс определяет последовательности операций и *переходных* состояний для всех определенных *начальных* состояний, чтобы достигнуть одно или несколько *конечных* состояний. Может быть несколько технологических процессов, созданных для различных начальных состояний.

Управление технологическим процессом заслуживает самостоятельного изучения. Инструментальное средство управления изменениями может быть объединено с инструментальным средством управления технологическим процессом или оно может иметь собственные пути определения возможных технологических процессов. Эти собственные пути могут или не могут позволить формировать и печатать документы технологических процессов. Технологические процессы могут быть реализованы, разрешая/запрещая переходы состояний, которые разработчик может выполнить в зависимости от текущего состояния продукта, в котором произошло изменение. Переходы разрешаются или запрещаются в зависимости от текущего состояния продукта (неподтвержденный, назначенный, повторно открытый и т.д.) и возможного состояния решения, если оно есть (фиксированное, неработоспособное, резервное) [16].

На рис. 3.30 показан образ экрана инструментального средства управления изменениями IBM Rational ClearQuest [86]. Образ экрана касается управления **дефектами**. Он перечисляет все дефекты, относящиеся к системе, и позволяет просматривать и управлять характеристиками выбранного дефекта. Дефект, показанный на рисунке, находится в состоянии «`Resolved`» (пересмотренный). Находясь в этом состоянии, описание технологического процесса изменений разрешает действия (переходы состояний), как показано в выпадающем списке, находящемся в правом нижнем углу изображения.

Кроме состояния каждый дефект характеризуется дополнительными свойствами, типа приоритета, серьезности, владельца и т. д. Инструментальное средство дает возможность поиска дефектов с некоторыми свойствами и формирования различных сообщений и диаграмм. Браузер рабочего пространства содержит системные и определенные пользователем запросы, диаграммы и сообщения. Разработчик может задать собственные запросы типа «`ToDo List`» (список операций) на рис. 3.30.

Рис. 3.31 демонстрирует предопределенную диаграмму тенденций для текущих дефектов проекта. Диаграмма показывает подсчет дефектов по их состояниям: предъявленное, пересмотренное, открытое, назначенное и закрытое. Подсчеты выполнялись ежедневно в течение двух недель.

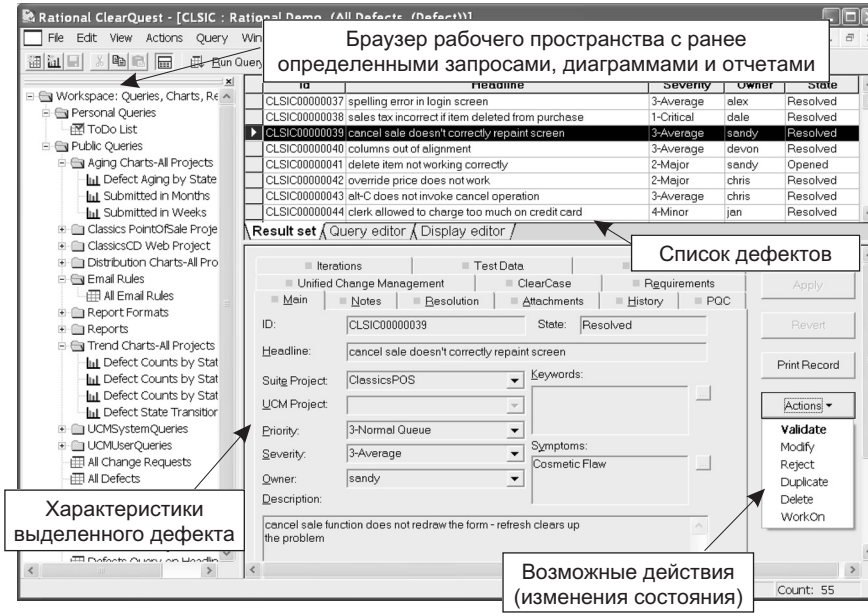


Рис. 3.30. IBM Rational ClearQuest: управление дефектами
Источник: образ экрана *IBM Rational Suite*, перепечатано по разрешению из *IBM Rational Suite Tutorial*, версия *2002.05.00,* © Copyright 2002, International Business Machines Corporation. All Rights Reserved

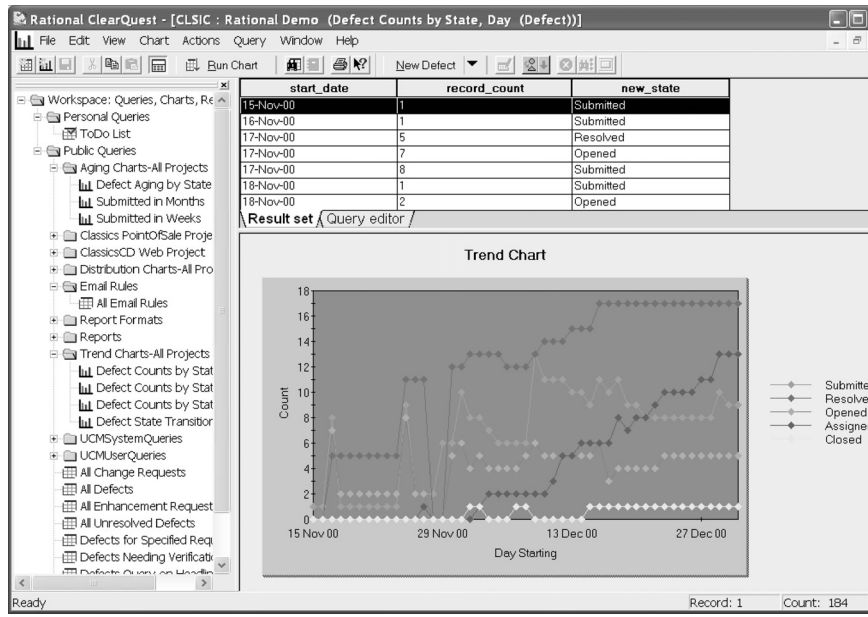


Рис. 3.31. IBM Rational ClearQuest: диаграмма тенденций дефектов
Источник: образ экрана *IBM Rational Suite*, перепечатано по разрешению из *Rational Suite Tutorial*, версия *2002.05.00,* © Copyright 2002, International Business Machines Corporation. All Rights Reserved

3.4.2. Поддержка версий

Управление конфигурацией тесно связано с управлением изменениями. Главная задача управления конфигурацией состоит в том, чтобы обеспечить историю изменений. История состоит из версий продуктов и изделий процесса. Система управления конфигурацией имеет внутренние средства для формирования различных **графов происхождения версий**. Графы позволяют прослеживать изменения продуктов, а также компоновать версии различных объектов, чтобы создавать *конфигурации* версий высокого уровня.

Разработчики должны знать, какие изменения были сделаны в продукте, над которым они в настоящее время трудятся. Как пример, интересный вопрос, который возникает на протяжении всего жизненного цикла разработки: «Когда мы неосторожно сделали эту ошибку?» Ответ на такой вопрос требует отслеживания множества изменений, сделанных в ПО, чтобы точно определить версию, где возникла ошибка. Определение причины проблемы дает надежду на распутывание ситуации.

Другая ситуация, когда важно «знание изменений», возникает, если разработчик готов экспортировать свои результаты в общее рабочее пространство (хранилище БД). В некоторый момент — до, в течение или после экспорта — может потребоваться сравнить текущую версию продукта и версию, зарегистрированную для потребителя. Такие сравнения легко делаются с помощью возможностей *сравнения и сопоставления*, имеющихся у инструментальных средств управления конфигурацией.

В некоторых ситуациях действие сравнения и сопоставления выполняется в фоновом режиме, но инструментальные средства управления конфигурацией обеспечивают также и GUI-средства для графических сравнений, завершающихся подсвечиванием различий. Рис. 3.32 показывает средство сопоставления Visual SourceSafe фирмы Microsoft [70]. Оно использует цвет для выделения различий в строках кода двух сравниваемых версий. Дополнительные строки кода представляются зеленым цветом, в то время как измененные строки представляются красными. Сравниваемые тексты обычно помещаются рядом и могут включать ссылки на номера строк (не показанные на рисунке).

3.4.3. Поддержка формирования системы

«**Формирование системы** — процесс компиляции и компоновки компонентов ПО в программу, которая выполняется в конкретной целевой конфигурации» [96]. Окончательная цель создания системы состоит в том, чтобы создать **товарную продукцию** — выполняемую версию ПО, которую можно предложить пользователям. Главная проблема при формировании — строго придерживаться всех зависимостей между компонентами ПО и их различных версий, а также обеспечить все необходимые ссылки на файлы данных. Процесс создания определяет также инструментальные средства, которые нужно использовать для успешного формирования (создания) программного продукта.

Средства формирования систем всегда объединяются в инструментальные средства управления конфигурацией и поэтому, естественно, расширяют IDE (раздел 3.3.5, рис. 3.29). Имеется много доступных инструментальных средств формирования систем типа `make` или `gnumake`, но фаворитом среди

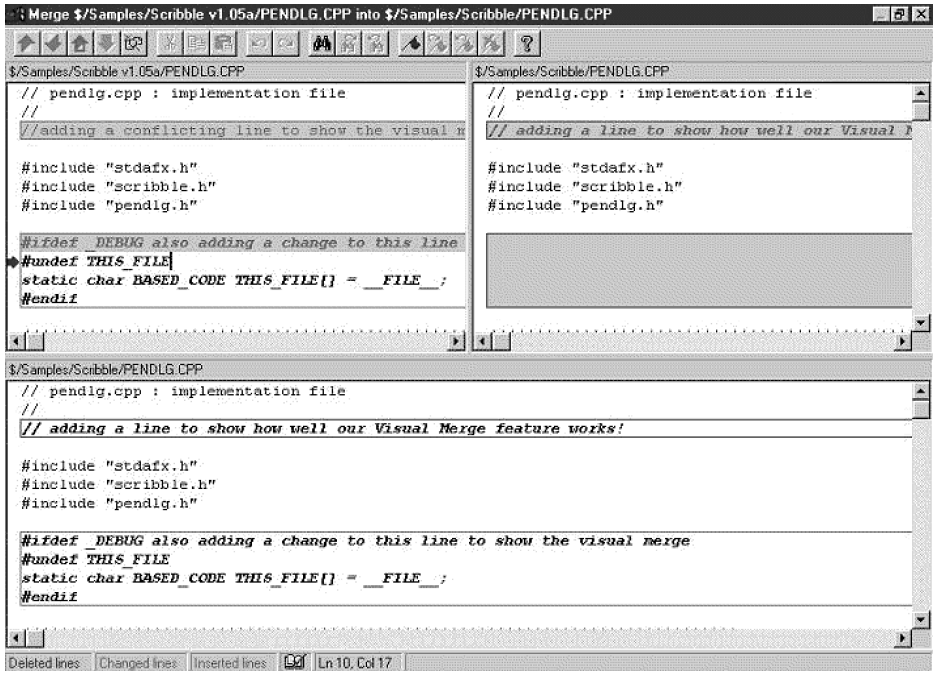


Рис. 3.32. Microsoft SourceSafe: выявление различий между версиями
Источник: образ экрана Microsoft® Visual SourceSafe из http://msdn.microsoft.com/ssafe/default.asp (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

Java-средств формирования систем следует считать Apache Ant (Another Neat Tool) [3]. Apache Ant особенно полезно для создания систем, которые работают на основе нескольких платформ.

Файлы конфигурации Ant (называемые также **файлами формирования**) написаны в XML и могут быть расширены путем использования классов Java. Файлы формирования вызывают целевое дерево, содержащее задачи формирования. Цели могут зависеть друг от друга типа того, что для создания товарной продукции нужно сначала скомпилировать код.

Рис. 3.33 является примером использования Apache Ant в Sun ONE Studio. Левая панель представляет `build.xml` как текущий файл Ant, видимый в основном окне. Внутри `build.xml` имеется ряд зафиксированных задач Ant (init, compile, jar, all, clean, javadoc и т. д.). Каждая из этих задач может быть выполнена непосредственно из IDE. Пример демонстрирует, что Ant-файл является документом формата XML. Документ состоит из XML-элементов (тегов), опознаваемых процессором Ant (см. гл. 17).

Тег «project» (проект) отмечает самый верхний элемент в Ant. На рис. 3.33 это — «`EMS Iteration 3`» с основной папкой «.» (текущая папка). Пока этот проект не будет обработан процессором Ant, а имя задачи до обработки не определено, процессор будет выполнять задачу, помеченную свойством по умолчанию (в данном случае это «`all`» — все).

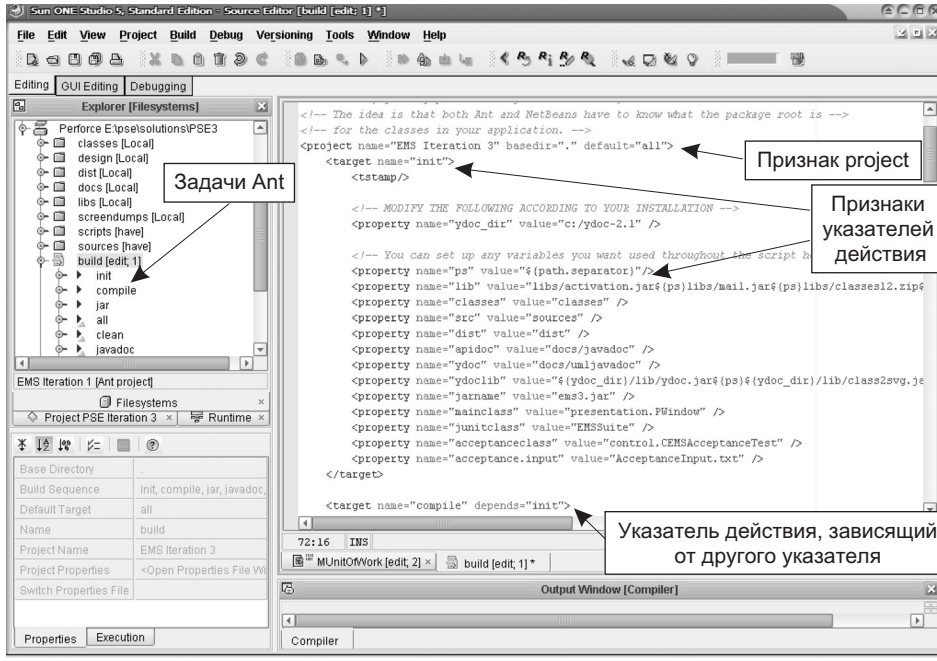


Рис. 3.33. Sun ONE Studio и Apache Ant: формирование системы
Источник: Sun Microsystems, Inc

Каждая задача Ant начинается с тега, называемого «target» (указатель действия) типа «`init`» (инициализировать) или «`compile`» (компилировать). Указатель действия «`init`» используется как основа для других указателей, задавая начальные свойства проекта. Указатель действия может ссылаться на другие указатели, чтобы выполнить свою работу. На рис. 3.33 указатель действия «`compile`» зависит от «`init`». Эта зависимость заставляет Ant выполнять указатель «`init`» перед указателем «`compile`».

3.4.4. Поддержка реинжиниринга

Самой объемной задачей программной инженерии является автоматическое преобразование проектов, представленных в существующих унаследованных системах, чтобы повторно реализовать их в новой форме. **Унаследованная система** — все еще используемое программное решение, обычно большая информационная бизнес-система, которая была реализована в давние времена в традиционной технологии и приобрела широкие дополнительные функциональные возможности в процессе **эволюционного сопровождения**.

Унаследованная система — история организации, которая остается с более раннего времени, но не может быть выброшена или заменена. Сама организация становится зависимой от этой системы, обеспечивающей ежедневные бизнес-операции. Эволюционное сопровождение системы, созданной много лет назад, нарушает ее первоначальный структурный проект. Добавление новых изменений необходимо из-за повышения требований пользовате-

лей и конкурентного пресса. Таким образом, эволюционное сопровождение должно продолжаться даже при условии, что изменения в системе могут привести к полностью непредсказуемым результатам. В организации уже может не остаться никого, кто понимает в деталях, как система формирует свои результаты.

Организация окажется в порочном круге. Понятно, что нужна новая качественная система, чтобы заменить унаследованную систему. Задача, однако, непреодолимая. Унаследованная система — это не только некая информационная система в организации, это — сам бизнес организации. Ее нельзя отставить в сторону, пока будет создана новая система на пустом месте. Еще более важно то, что замена системы означает необходимость в первую очередь понять функциональные возможности системы. В большинстве случаев для системы не существует никакой документации или документация уже не соответствует тому, что система на самом деле выполняет. Сложность системы не допускает ручные средства восстановления текущего проекта системы. Необходимы автоматические инструментальные средства.

Программный реинжиниринг (реорганизация) — Software reengineering — область знаний, связанная с методами, технологиями и инструментальными средствами для исследования и изменения унаследованных систем, чтобы восстановить их проекты и заново реализовать в новой форме. Реинжиниринг состоит из ряда технологий:

- обратное проектирование;
- формирование новой документации из исходного кода;
- реструктурирование логики программы;
- новое определение системы на современной платформе;
- перевод исходного кода на другой язык;
- реинжиниринг данных (в противоположность реинжинирингу процесса);
- прямое проектирование.

Текущая реальность такова, что унаследованные системы в основном запрограммированы на КОБОЛе и осуществляют доступ к пред-реляционным БД (то есть сетевым и иерархическим). Однако история имеет тенденцию повторяться. Имеется большое количество подтверждений того, что и сейчас разработанные и развернутые системы, как мы говорим, станут унаследованными в тот же день, когда они развернуты в организациях. Это справедливо для систем, созданных на Java и C#, и особенно для разработок на C++, где часто используется переключение на С-программирование низкого уровня и процедурный (не-объектно-ориентированный) стиль кодирования.

Первая операция реинжиниринга — обратное проектирование. В объектно-ориентированном мире **обратное проектирование** означает получение UML-модели из объектно-ориентированной программы. Имеются два главных *источника* обратного проектирования: исходный код программы или скомпилированный код программы. Главная *цель* здесь — получение модели классов.

Большинство инструментальных средств моделирования обеспечивает то или иное обратное проектирование. Фактически, как сказано в разделе 3.3.2, многие IDE позволяют выполнять синхронизированную разработку графи-

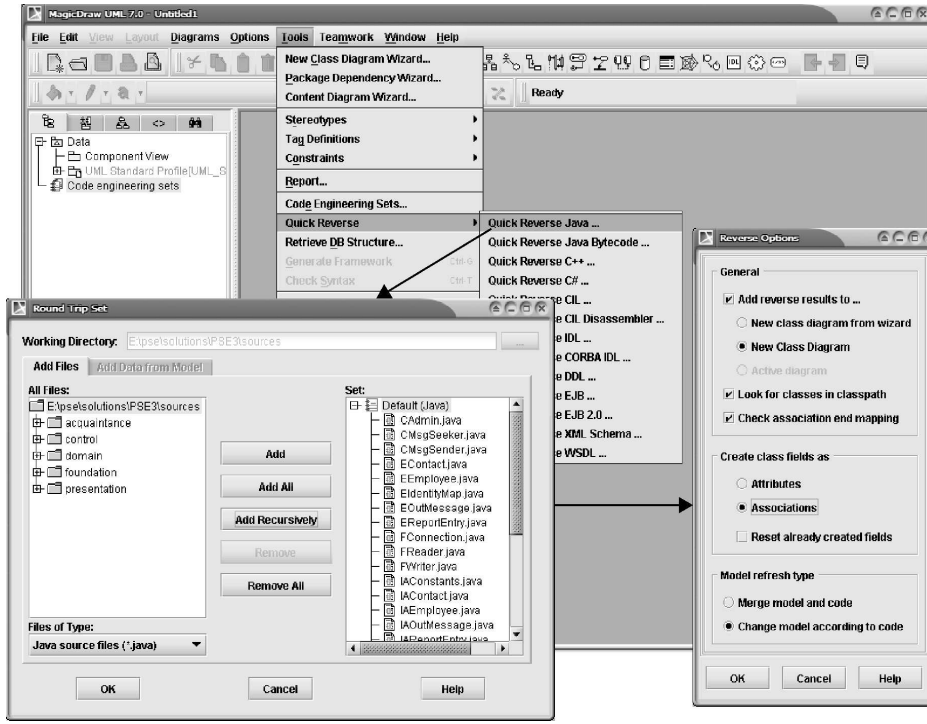


Рис. 3.34. No Magic's MagicDraw™ до обратного проектирования
Источник: образ экрана No Magic's MagicDraw (до Reverse Engineering — обратного проектирования). © Copyright No Magic, Inc., перепечатано по разрешению. All Rights Reserved

ческих моделей и кода. Это означает, что такие инструментальные средства имеют встроенные возможности циклического проектирования. Более полное обратное проектирование позволяет работать со многими источниками и формировать более завершенные UML-проекты.

Рис. 3.34 показывает ряд окон, которые начинают процесс обратного проектирования в MagicDraw [73]. *Источник* — Java-код (одна из программ учебного примера этой книги). Цель — новая диаграмма классов.

Результат обратного проектирования, определенного на рис. 3.34, показан на рис. 3.35. Преобразованные Java-пакеты и классы генерируются в окне браузера проекта. Модель класса показана в окне «вида с высоты птичьего полета», которое может масштабироваться, чтобы представлять в главном окне части диаграммы в удобной форме. Как видно, схема диаграммы требует человеческого вмешательства для устранения строк, попадающих в несколько классов, и т. д. (это — универсальная проблема, поскольку не существует никакого «совершенного» автоматического алгоритма определения расположения строк).

Текущее состояние в искусстве обратного проектирования таково, что ограничения конфигурации тревожат меньше всего. На первый взгляд кажется, что инструментальные средства работают хорошо, но они все еще оставляют же-

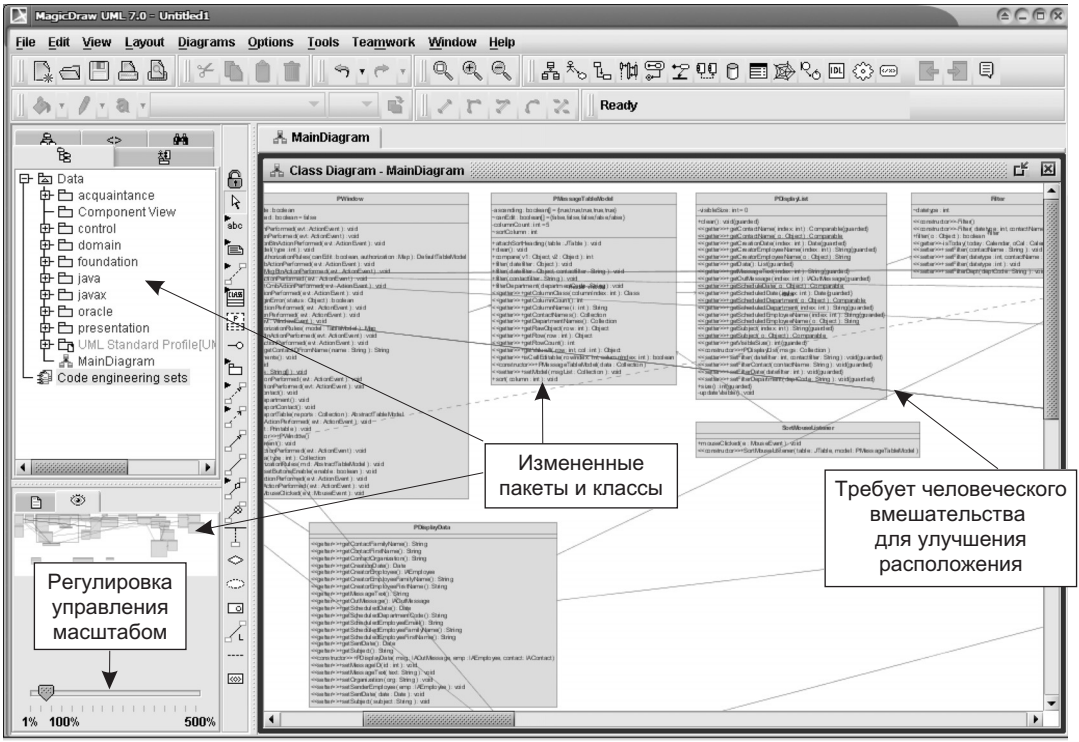


Рис. 3.35. No Magic's MagicDraw™ после обратного проектирования
Источник: образ экрана No Magic's MagicDraw (после Reverse Engineering). © Copyright No Magic, Inc., перепечатано по разрешению. All Rights Reserved

лать лучшего, когда выполняют обратное проектирование унаследованных систем. Унаследованные системы, разрабатываемые годами многими программистами с использованием различных версий компиляторов, имеют фрагменты кода, обладающие необычными особенностями программирования и уникальными опциями компилятора. Эти проблемы добавляются к проблемам современного состояния искусства обратного проектирования, которое неспособно должным образом преобразовать некоторые особенности кода. Например, очень немного инструментальных средств может удовлетворительно выполнить работу по обратному проектированию внутренних классов или отношений, основанных на коллекциях.

Резюме

1. Инструментальные средства программной инженерии принадлежат категории программных продуктов, известных как совместная автоматизированная разработка (*computer-supported collaborative work* — CSCW) или групповая вычислительная обработка (*workgroup computing*).
2. Инструментальные средства программной инженерии могут быть расклассифицированы на четыре группы: 1) инструментальные средства

управления проектами; 2) инструментальные средства моделирования систем (обычно известные как CASE-средства); 3) интегрированные среды разработки (IDE) и 4) инструментальные средства управления изменениями и конфигурацией.

3. *Инструментальные средства управления* проектами в основном связаны с планированием и управлением проектом. Однако современные средства дополнительно реализуют ряд связанных с этим функций или обеспечивают тесную интеграцию с используемыми инструментальными средствами. Эти функции — из области управления реализацией, сотрудничества на основе Web-технологий, информационного обеспечения, портфельного управления, обеспечения метрик и управления рисками.
4. *Инструментальные средства моделирования систем* охватывают все инструментальные средства, которые помогают инженерам ПО в задачах разработки — от анализа через проектирование к реализации. Все современные инструментальные средства поддерживают язык UML. Хотя они и способны сформировать код на основе моделей, они обычно все-таки не составляют полную среду программирования. Инструментальные средства моделирования систем сосредоточены на управлении требованиями и спецификациями, проектировании систем (включая моделирование БД), а также формировании отчетов и документов проекта.
5. *IDE* — сложные рабочие места программирования, которые обеспечивают дружественный интерфейс, чтобы помочь коллективам программистов в решении всех типичных задач программирования. Они могут существенно улучшить производительность программистов в цикле написания, реализации и отладки программ. Современные IDE обеспечивают интеграцию и связь с визуальным моделированием, БД и источниками бизнес-компонентов, стремясь помочь в разработке промышленных приложений. Они также объединяются с инструментальными средствами управления изменениями и конфигурацией, что обеспечивает тесное сотрудничество всего коллектива программистов.
6. *Инструментальные средства управления изменениями и конфигурацией* на самом деле являются двумя сторонами одной и той же медали. Инструментальное средство управления изменениями обеспечивает запись изменений, которые произошли в течение жизненного цикла ПО. Управление изменениями проявляется в цикле реализации-тестирования, чтобы обеспечить управление дефектами, выявленными при тестировании. Программирование, являясь творческой операцией, завершается формированием различных версий компонентов ПО. Необходимо отслеживать версии, созданные различными разработчиками, а также хранить предшествующие версии для их последующего пересмотра в случае необходимости. *Управление версиями* — область *инструментальных средств управления конфигурацией*. Инструментальные средства управления изменениями и конфигурацией имеют возможность *формирования* систем, работающих на определенных целевых платформах. Они могут также объединяться с инструментальными средствами реинжиниринга, чтобы помочь в обратном проектировании унаследованных систем.

Ключевые термины

Обзорные вопросы

1. Насколько важно инструментальным средствам управления проектами поддерживать Web-технологии? Обсудите все «за» и «против».
2. Как инструментальные средства управления проектами поддерживают формирование графов сетевых операций?
3. Как включение средств управления реализацией в инструментальное средство управления проектами изменяет его акцент? Объясните.
4. Что такое информационное обеспечение? Как оно используется в контексте управления проектом?
5. Что такое портфельное управление? Как оно связано с управлением проектом?

6. Объясните отношение метрик к управлению проектом.
7. Объясните отношение между управлением рисками и управлением проектом.
8. Какие главные возможности ожидаются от инструментального средства управления требованиями?
9. Какова роль хранилища в визуальном моделировании?
10. Действительно ли UML достаточен для моделирования БД? Следует ли его расширить, чтобы обеспечить такое моделирование БД, когда можно формировать коды БД?
11. Объясните различие между «Step Into» и «Step Out» в процессе отладки.
12. Объясните понятия «изменение», «версия» и «конфигурация» и как эти концепции связаны.
13. Что такое технологический процесс в управлении изменениями? Как он определяется и каким образом выражается?
14. Какие существуют технологии реинжиниринга ПО? Дайте краткое объяснение.

Примеры задач

1. Адреса Web-сайтов инструментальных средств управления проектами, упомянутых в этой главе, помещены в расположенную ниже таблицу 3.1. Посетите Web-сайты, чтобы изучить самые последние изменения в этих инструментальных средствах. Если связь с Web-сайтами невозможна, попытайтесь установить причину происшедшего (с инструментальным средством и/или продавцом). Поищите в Интернете информацию о самых последних тенденциях в инструментальных средствах управления проектами и в инструментальных средствах, формирующих заголовки. Создайте список важнейших инструментальных средств с текущими адресами Web-сайтов и кратким перечнем основных особенностей.

Таблица 3.1. Инструментальные средства управления проектами, на которые имеются ссылки в главе 3

Название инструментального средства	Адрес Web-сайта
MS Project [70]	http://www.microsoft.com/office/project/default.asp
Manage Pro [65]	http://www.managepro.net/
eRoom [29]	http://www.eroom.net/eRoomNet/
eProject [28]	http://www.eproject.com/
Primavera Enterprise [82]	http://www.primavera.com
Small Worlds [95]	http://www.thesmallworlds.com/
©Risk [75]	http://www.palisade-europe.com/
Risk Radar [40]	http://www.iceincusa.com/products_tools.htm

2. Адреса Web-сайтов инструментальных средств моделирования систем, упомянутых в этой главе, помещены в расположенную ниже таблицу 3.2. Посетите Web-сайты, чтобы изучить самые последние изменения в этих инструментальных средствах. Если связь с Web-сайтами невозможна, попытайтесь установить причину происшедшего (с инструментальным средством и/или продавцом). Поищите в Интернете информацию о самых последних тенденциях в инструментальных средствах моделирования систем и в инструментальных средствах, формирующих заголовки. Создайте список важнейших инструментальных средств с текущими адресами Web-сайтов и кратким перечнем основных особенностей.

Таблица 3.2. Инструментальные средства моделирования систем, на которые имеются ссылки в главе 3

Название инструментального средства	Адрес Web-сайта
RequisitePro [86]	http://www.rational.com
DOORS [104]	http://www.telelogic.com/
Enterprise Architect [97]	http://www.sparxsystems.com.au/
Rational Rose [86]	http://www.rationai.com
Poseidon [33]	http://www.gentleware.com/
ArgoUML [4]	http://argouml.tigris.org/
MagicDraw [73]	http://www.magicdraw.com/
Rational SODA [86]	http://www.rational.com
PowerDesigner [103]	http://www.sybase.com/products/enterprisemodeling

3. Адреса Web-сайтов интегральных сред разработки (IDE), упомянутых в этой главе, помещены в расположенную ниже таблицу 3.3. Посетите Web-сайты, чтобы изучить самые последние изменения в этих инструментальных средствах. Если связь с Web-сайтами невозможна, попытайтесь установить причину происшедшего (с инструментальным средством и/или продавцом). Поищите в Интернете информацию о самых последних тенденциях в IDE и в инструментальных средствах, формирующих заголовки. Создайте список важнейших инструментальных средств с текущими адресами Web-сайтов и кратким перечнем основных особенностей.

Таблица 3.3. IDE, на которые имеются ссылки в главе 3

Название инструментального средства	Адрес Web-сайта
Sun ONE Studio [102]	http://wwws.sun.com/software/sundev/jde/
Together ControlCenter [14]	http://www.borland.com/together/
Oracle JDeveloper [74]	http://otn.oracle.com/products/jdev/content.html
WSAD [39]	http://www-3.ibm.com/software/awdtools/studioappdev/

4. Адреса Web-сайтов инструментальных средств управления изменениями и конфигурацией, упомянутых в этой главе, помещены в расположенную ниже таблицу 3.4. Посетите Web-сайты, чтобы изучить самые последние изменения в этих инструментальных средствах. Если связь с Web-сайтами невозможна, попытайтесь установить причину происшедшего (с инструментальным средством и/или продавцом). Поищите в Интернете информацию о самых последних тенденциях в инструментальных средствах управления изменениями и конфигурацией и в инструментальных средствах, формирующих заголовки. Создайте список важнейших инструментальных средств с текущими адресами Web-сайтов и кратким перечнем основных особенностей.

Таблица 3.4. Инструментальные средства управления изменениями и конфигурацией, на которые имеются ссылки в главе 3

Название инструментального средства	Адрес Web-сайта
Perforce [77]	http://www.perforce.com/
Rational Clear Quest [86]	http://www.rational.com
Visual SourceSafe [69]	http://msdn.microsoft.com/ssafe/default.asp
Ant [3]	http://www.apache.org/

5. Посетите Интернет, чтобы изучить Java-технологии. Какие существуют основные Java-технологии? Это те же самые, что рассмотрены в книге? Имеются ли новые технологии, не упомянутые в книге?
6. Рассмотрите рис. 3.23. Почему программист установил точку останова на строке 27? Предположите, что первоначально значение `myID` (мой идентификатор) равно 1, а размер `size` первоначально равен 0.

Глава 4

Планирование и отслеживание проекта программного обеспечения

В этой главе рассматриваются вопросы планирования и отслеживания проекта ПО, а управление процессом создания и эксплуатации ПО будет обсуждено в следующей главе. Все это составляет содержание процесса управления в пятиугольнике разработки ПО (см. вводную часть главы 1). Различие между планированием проекта и управлением процессом довольно неопределенно. Планирование и отслеживание проекта ПО является непрерывной операцией оценки того, сколько времени, денег, трудозатрат и ресурсов должно быть израсходовано на проект.

Будучи не однократной, а непрерывной операцией, планирование проекта ПО можно считать всеобъемлющей операцией управления продуктами и процессами разработки ПО. В частности, планирование проекта ПО включает не только само планирование, составление бюджета и отслеживание, но также и анализ рисков, гарантию качества, управление людскими ресурсами и управление конфигурацией. Однако в данной главе считается, что планирование эквивалентно собственно планированию и отслеживанию проекта. Сопутствующие проблемы представлены здесь только по мере необходимости, а полностью они рассмотрены в главе 5.

Интересно, что сопутствующие проблемы анализа рисков, гарантии качества, управления конфигурацией и т. д. имеют многопроектный смысл. Они актуальны и в случае наличия нескольких проектов. Это — другая причина отделить планирование для единственного проекта от многопроектных аспектов. Эти многопроектные аспекты обсуждаются более подробно в главе 5.

Планирование проекта ПО начинается там, где завершается стратегическое планирование и бизнес-моделирование. Стратегическое планирование и бизнес-моделирование решают вопросы, связанные с организационными задачами, показателями и целями. Эти показатели и цели определяют потребность проектирования ПО.

4.1. Разработка плана проекта

Как сказано в главе 1, **планирование проекта** — не отдельная стадия жизненного цикла разработки. Это — всеобъемлющая операция, охватывающая все стадии жизненного цикла. Хотя некоторые модели жизненного цикла, например, RUP (раздел 1.3.2), рассматривают планирование как самую первую операцию в начале проекта, они поясняют, что это — только начальное пла-

нирование. Никакие разумные оценки стоимости (бюджета) и времени (планы) невозможны, пока не будут определены, по крайней мере, требования пользователя.

Исследование Бёма и др. [11] указывает, что первоначальные плановые оценки после изучения выполненного проекта могут измениться в одну или другую сторону в четыре раза по сравнению с реальной стоимостью/временем, затраченными на выполнение проекта. Если реальные израсходованные стоимость/время равны некоторой величине `x`, то первоначальные оценки могли бы быть в диапазоне от `0.25x` до `4x`. Как показано на рис. 4.1, последовательные коррекции плана оценивают все точнее реальное значение `x`. Приближения на рис. 4.1 являются все же только приближениями. Они различаются для разных проектов и отличаются от результатов работы Бёма для различных стадий жизненного цикла, используемых на рис. 4.1.

Оценка стоимости/времени (бюджет/план) — только один аспект планирования проекта ПО. Практически документ бюджета/плана может быть одним из нескольких документов планирования. Другие документы могут включать план качества, план тестирования, план использования людских ресурсов, план управления конфигурацией и т. д. Они могут быть отдельными документами или частями общего планового документа проекта. Соммервиль [96] рекомендует следующие разделы типичного плана проекта:

1. *Введение* — определить проектные показатели, основную стоимость и ограничения по времени.
2. *Организация проекта* — описать организацию коллектива разработчиков.
3. *Анализ рисков* — определить риски и способы управления ими.

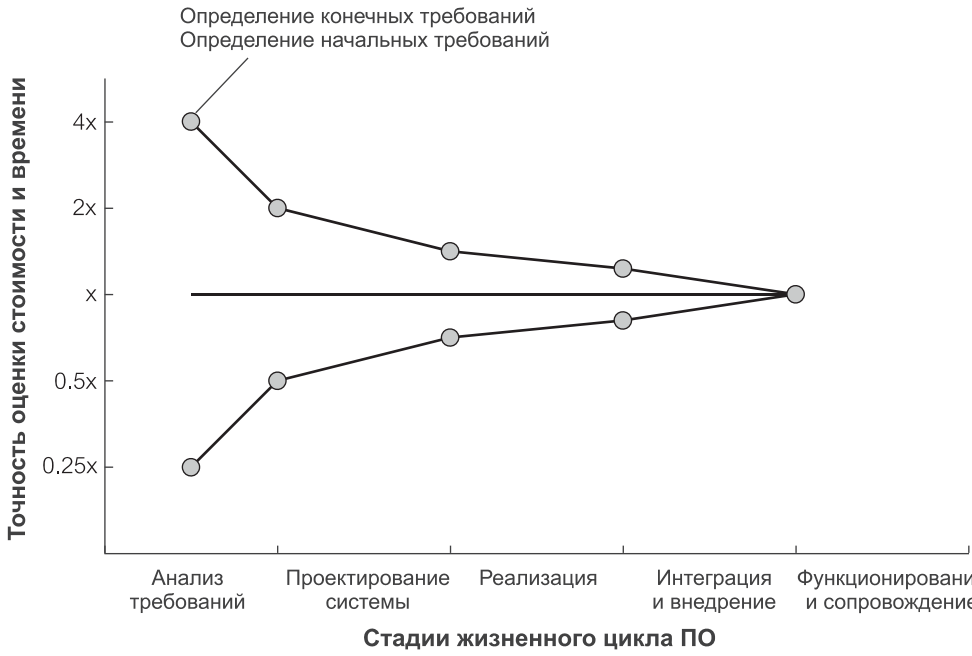


Рис. 4.1. Оценка плана проекта

4. *Требования к ресурсам аппаратного и программного обеспечения* — определить аппаратное/программное обеспечение, необходимые для разработки.
5. *Выполнение* — определить проектные показатели, контрольные точки и подлежащие сдаче продукты.
6. *План проектных работ* — определить распределение времени и ресурсов по операциям.
7. *Механизмы контроля и передачи сообщений* — определить потребности в механизмах контроля и управляющих сообщениях.

Шаги в разработке плана проекта ПО подобны тем, которые требуются в других видах творческих проектов [59]. Рис. 4.2 представляет UML-диаграмму операций, которая иллюстрирует типичную последовательность шагов.

Планирование проекта начинается с определения *показателей*, *возможностей* и *ограничений* проекта. На практике показатели и возможности проекта будут определены до начала планирования проекта (поскольку они являются составными частями стратегического планирования и бизнес-моделирования). Два типичных ограничения проекта — время и деньги, так как проекты имеют сроки и ограниченные бюджеты. Другие ограничения могут касаться пригодности ресурсов, в частности, людских ресурсов.

Подлежащие сдаче продукты проекта — это программные продукты или сервисы, которые удовлетворяют проектным показателям. Они поставляются заинтересованным сторонам проекта (владельцам проекта и пользователям). Эти продукты должны удовлетворять требованиям пользователя, соответствовать проектной спецификации и удовлетворять ряду стандартов качества. В современных методах итеративных и пошаговых разработок проект, вероятно, будет иметь несколько подлежащих сдаче продуктов.

Как только подлежащие сдаче продукты станут известны, в работе можно выбрать контрольные точки, стадии и задачи. Иногда это называется **структурной декомпозицией работы**. **Контрольная точка** — существенное событие в проекте. Она отмечает конец некоторой операции процесса и используется, чтобы контролировать продвижение в проектировании. Контрольная точка обычно используется, чтобы зафиксировать завершение создания подлежащего сдаче продукта, но она может зафиксировать и другие виды результатов работы над проектом, например, просто завершение важного шага проектирования. Контрольная точка может также показать завершение проектной стадии (а подлежащий сдаче продукт обычно является результатом некоторой заключительной стадии). Проекты (и стадии) состоят из **задач**, которые являются **операциями** с ясными началом и концом.

Планирование первых шагов проектирования обращает внимание на то, *что* это за проект. Оценка потребностей ресурсов — из области *как* выполнить проект — необходима для решения, как выполнить проектные задачи. **Ресурсы** — это люди, оборудование, материалы, поставки, программное и аппаратное обеспечение и т. д., необходимые для решения проектных задач. Оценка потребностей ресурсов может быть только начальным шагом в целом ряду операций планирования, таких как планирование комплектации штата и необходимых приобретений.

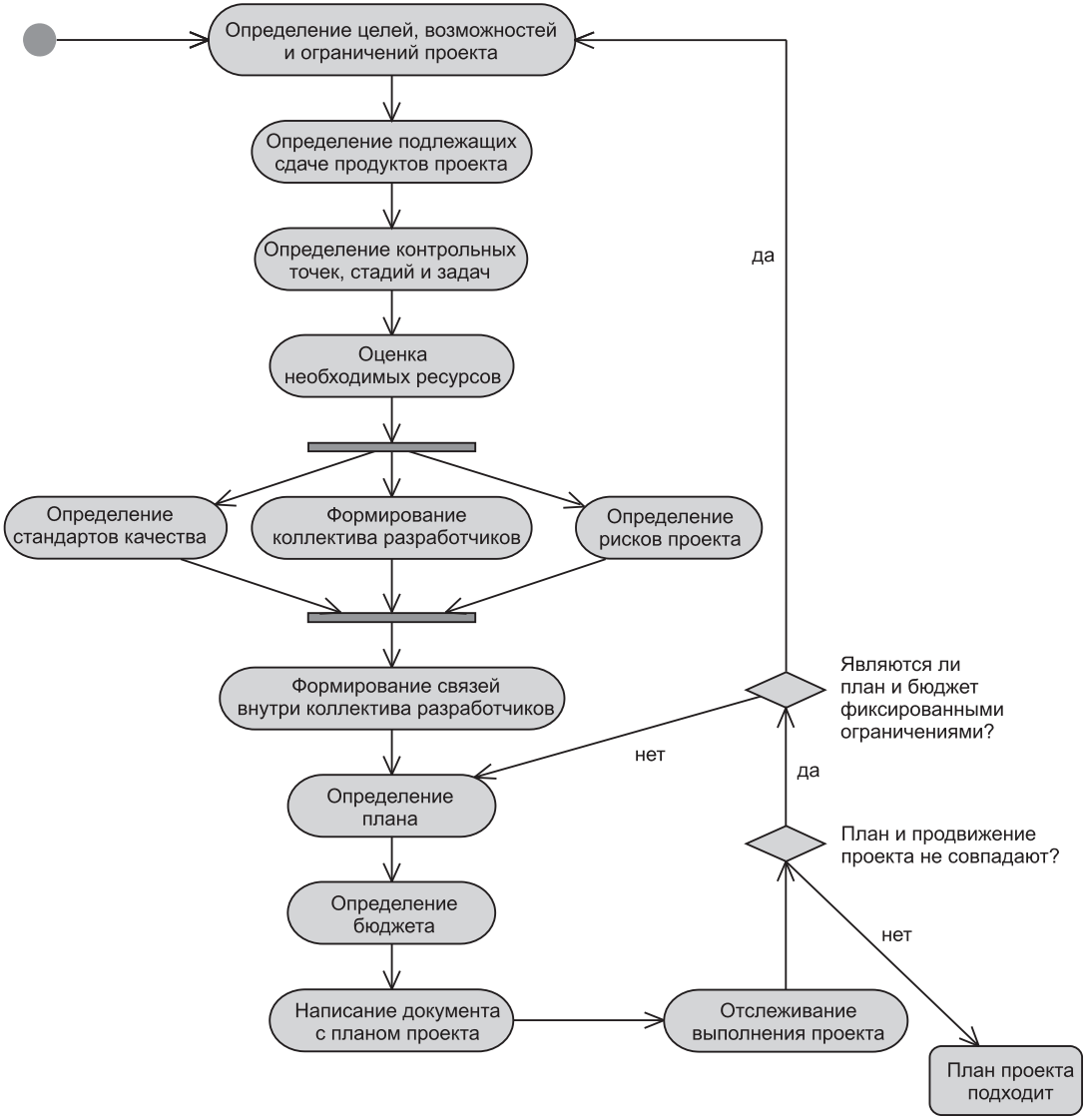


Рис. 4.2. Шаги планирования проекта

Формирование *коллектива разработчиков проекта* — это операция, которая управляет потребностями людских ресурсов. Одно дело — знать, какие людские ресурсы необходимы, а вот формирование коллектива — совсем другой вопрос. Некоторые члены коллектива могут быть взяты непосредственно в организацию, другие должны быть наняты на полный рабочий день или как работающие по контракту. Создание хорошего коллектива с дополняющими навыками, наличием духа товарищества, без личных напряженных отношений и заинтересованного в проекте — одна из наиболее серьезных проблем менеджера проекта.

Планирование проекта включает определение стандартов качества. **Качество** — относительное понятие. Оно на виду у всех. В конечном счете, изделие высокого качества — это изделие, которое удовлетворяет пользователя. Это означает программный продукт, который был поставлен вовремя и в пределах выделенного бюджета, реализует требования пользователя, имеет дружественный интерфейс, не отказывает в работе и т. д. Более высокое качество может означать более высокую стоимость. Определение стандартов качества должно принимать во внимание воздействие политики качества на продолжительность проектирования и закладываемый бюджет и даже на возможности проекта.

Планирование проекта включает также определение проектных рисков. **Риски** — неблагоприятные события, которые часто трудно ожидать, и даже когда они определены и известны, еще неясно произойдут они или нет. На работнике, занимающемся планированием проекта, лежит ответственность за определение стратегии *предотвращения рисков* и выработку *планов на случай непредвиденных обстоятельств*, если риск произойдет.

Установление связей в коллективе разработчиков — важная задача в любых, но особенно в малых проектах. В малых проектах члены коллектива могут связываться через неофициальные встречи, обмены по электронной почте и т. д. В больших проектах связи внутри коллектива должны быть запланированы и обеспечены технологией. Технологические решения включают формирование рабочей группы, управление временем, управление конфигурацией и связанные с этим системы ПО, способные обеспечивать обмен мультимедийными документами.

Шаг определения **плана** проекта — нечто связанное с оценкой времени, необходимого не только для самой разработки, но также и для управления проектом, гарантии качества изделий и управления рисками. Планирование включает также и распределение ресурсов по задачам.

Шаг определения **бюджета** проекта — нечто вроде оценки стоимости ресурсов, которые нужны для задач. Имеются различные технологии такой оценки. Ни одна из них не совершенна и, по крайней мере, пара их должна использоваться для проекта в целях проверки. В некоторых случаях затраты могут быть рассчитаны на основе плана, используя тарифные ставки и систему оплаты для располагаемых людских ресурсов, систему затрат на задачи и т. д. В других случаях затраты могут быть получены на основе известных метрик затрат подобных проектов. Можно также оценить бюджет, используя алгоритмические модели, которые предсказывают стоимость, основанную на проектных характеристиках (параметрах), заложенных в математическую модель.

Написание **документа с планом** проекта — кульминация ранее выполненных шагов планирования. Этот документ должен быть передан работникам управления и членам коллектива проектировщиков. Передача может быть в электронной форме с помощью электронной почты, или путем размещения документа на Web-сайте Intranet.

План проекта регулярно пересматривается в процессе **отслеживания выполнения проекта** в случае выявления несовпадения с планом. Если несовпадение можно устранить, обновляя график проектных работ и бюджет, то

пересмотр осуществляется относительно просто. В противном случае пересмотр должен начаться с самого первого шага планирования проекта, то есть, из переопределения показателей, возможностей и ограничений.

4.2. Планирование проекта

Планирование проекта и его отслеживание — ежедневная операция менеджеров проекта. Создание временных планов предполагает, что известны *структурная декомпозиция работы* и список задач. Простое, как это может казаться, определение задач в большом проекте на самом деле — очень трудная работа. Задачи — это операции с ясным временем начала и конца работ; идеально продолжительность лежит в пределах между одним днем и двумя неделями. Различные модели жизненного цикла (раздел 1.3) завершаются различными структурными декомпозициями работы и различными списками задач. Многие задачи возникают лишь как реакция на неожиданные события при проектировании и не могут быть просто предсказаны.

Планирование проекта — это подвижная цель. Имеются сложные взаимозависимости между задачами, различными ограничениями на сроки их выполнения и разнообразием возможных прерываний в работе. Отсутствие крайнего срока выполнения единственной задачи влияет на выполнение остальной части графика (так называемый «эффект ряби»). *Ресурсы*, выделенные проекту, и в особенности людские ресурсы, строго ограничены, и их часто невозможно просто перепланировать.

К счастью, инструментальные средства управления проектами, типа Microsoft Project, используемого для примеров в этой главе, могут автоматически создавать и корректировать графики, чтобы отразить изменения в задачах и ресурсах. Они также могут определять критические пути в графиках. **Критический путь** — любая последовательность связанных задач в проекте, которая должна быть выполнена вовремя, чтобы проект мог быть закончен к крайнему сроку. **Пробуксовка** (задержка) любой задачи на критическом пути приведет к нарушению сроков создания проекта. С другой стороны, задачи, которые не находятся на критическом пути, могут позволить себе задержки (они имеют так называемое **резервное время**). CPM-**диаграммы сети** (рис. 3.1) — хорошая технология для визуализации критических путей и пробуксовок.

4.2.1. Задачи, контрольные точки и подлежащие сдаче продукты

Создание графика начинается с определения списка **задач** для каждой стадии проекта. Задачи могут быть иерархически организованы в многоуровневые подзадачи. Каждой задаче назначается *продолжительность* — оценка, сколько времени потребуется, чтобы закончить задачу. Продолжительность может быть задана с использованием различных временных шкал типа часов, дней и недель. На практике желательно, чтобы продолжительность была не больше двух недель.

Как только продолжительности задач будут определены, график может быть построен либо *в прямом направлении* (с даты начала проекта), либо *в обратном* (с даты завершения проекта). Обратное формирование может быть предпочтительно, если проект следует закончить к некоторой дате. Продолжительности используются, чтобы вычислить количество необходимой работы. Если список задач и дата завершения являются строгими ограничениями, график может быть отрегулирован только изменением распределения ресурсов по задачам.

График отражает **календарь проекта**. Календарь определяет рабочее и нерабочее время. Рабочее время — это рабочие часы в будних днях от понедельника до пятницы с 9.00 утра до 5.00 пополудни. Нерабочее время — все остальное время, попадающее на период выполнения проекта. Оно включает выходные дни, праздники, обеденные перерывы и т. д.

Контрольная точка (веха) — это задача, которая завершается существенным событием или результатом. Хотя задача с некоторой продолжительностью может быть помечена как заканчивающаяся в контрольной точке, общепринято создавать контрольные точки как отдельные задачи с нулевой продолжительностью. Контрольная точка может быть признаком конца стадии проекта или конца всего проекта.

Контрольная точка может представлять **подлежащий сдаче продукт**. Факт, что контрольная точка обозначает подлежащий сдаче продукт, может быть отражен в имени задачи, представляющей контрольную точку. Можно также создать отдельную задачу для подлежащего сдаче продукта с нулевой продолжительностью. И контрольные точки, и подлежащие сдаче продукты представляют отдельные состояния проекта. Раз так, они должны быть названы, начиная с существительного (например, «БД, загруженная и готовая к тестированию»). В противоположность этому, имена других задач начинаются с глагола (например, «реализовать триггеры БД»).

Рис. 4.3 — пример интерфейса Microsoft Project для создания списка задач проекта. Задачи последовательно введены в строки. Им даны названия, и введены оценки их продолжительности. Рисунок использует отвлеченные названия, отражающие типы задач. Задачам, которые являются контрольными точками или подлежащими сдаче продуктами, дают нулевую продолжительность. Горизонтальные полосы символизируют задачи. Контрольные точки и подлежащие сдаче продукты изображены маленькими ромбами и определенными датами.

Кроме регулярных задач на рис. 4.3 продемонстрирована рекуррентная задача, а также суммарная задача с двумя уровнями подзадач. **Рекуррентная задача** — задача, которая повторяется через регулярные интервалы времени, такие как каждый день, раз в неделю и т. д. (например, «провести обсуждение просмотренного кода каждую пятницу»). Рекуррентная задача на рис. 4.3 выполняется каждую пятницу и имеет четыре экземпляра двухчасовой продолжительности (четыре последовательные пятницы).

Задачи могут быть организованы в иерархии подзадач. Задача в верхней части иерархии называется **суммарной задачей**. Продолжительность и другие характеристики суммарной задачи вычисляются из характеристик вклю-

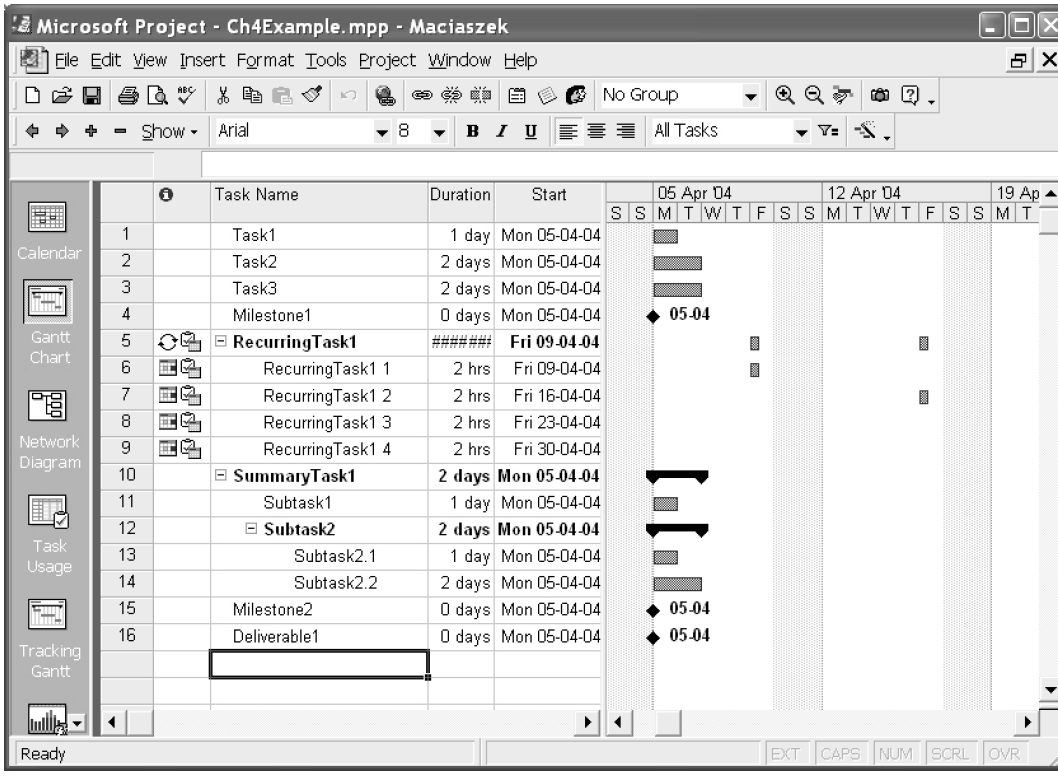


Рис. 4.3. Создание списка задач
Источник: образ экрана Microsoft® Office Project (2003), перепечатано по разрешению Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

ченных подзадач. Подзадача — это любая задача, которая является частью суммарной задачи или части какой-либо подзадачи более высокого уровня.

4.2.2. Планирование задач в виде ленточной диаграммы

Задачи связаны друг с другом и могут зависеть от определенных дат. **Зависимости задач** должны быть определены прежде, чем может быть рассчитан график проектных работ. Зависимости задач возникают между двумя или большим количеством *связанных задач*. Несвязанные задачи рассматриваются как независимые, так что они могут выполняться параллельно. Имеются два основных вида зависимостей задач [59]:

- зависимости задач, связанные с организацией работ;
- зависимости задач из-за использования одних и тех же ресурсов (раздел 4.2.3).

Зависимости задач, связанные с организацией работ, различают задачи-предшественники и задачи-преемники. Microsoft Project использует четыре основные категории зависимостей «предшественник-преемник»:

- «*конец-начало*» (finish-to-start — FS) — задача-преемник не может начинаться, пока не закончится задача-предшественник;
- «*начало-начало*» (start-to-start — SS) — задача-преемник не может начинаться, пока не начата задача-предшественник;
- «*конец-конец*» (finish-to-finish — FF) — задача-преемник не может заканчиваться, пока не закончена задача-предшественник;
- «*начало-конец*» (start-to-finish — SF) — задача-преемник не может заканчиваться, пока не начата задача-предшественник.

Как только зависимости между связанными задачами установлены, изменение дат начала и продолжительности предшественников воздействует на даты преемников. Можно также определить и более сложные зависимости «предшественник-преемник», в частности:

- *зависимости с задержками*, когда между задачей-предшественником и задачей-преемником вводится *время задержки*;
- *зависимости с наложением*, в которых начало задачи-преемника *опережает по времени* окончание задачи-предшественника.

Зависимости с задержками включают задержку (время задержки) между связанными задачами. Например, время задержки в три дня задается, если задача-преемник может начаться только через три дня после завершения задачи-предшественника. В противоположность этому **зависимости с наложением** представляют наложение (время сдвига) связанных задач. Это позволяет начать задачу-преемника, прежде чем задача-предшественник будет закончена. Время задержки и время сдвига могут быть определены в абсолютных величинах (например, +3 дня для времени задержки или –2 дня для времени сдвига). Они могут также быть определены в процентах от продолжительности задачи-предшественника.

При наличии зависимостей между задачами может быть автоматически построена ленточная диаграмма. **Ленточная диаграмма Ганта** показывает задачи на вертикальной оси, а даты начала и конца задач — на горизонтальной оси. Диаграмма на рис. 4.4 называется диаграммой Ганта по имени ее изобретателя. Она может быть использована для планирования времени всего проекта, конкретной стадии проекта или для конкретных людских ресурсов. Инструментальное средство управления проектами может синхронизировать основную диаграмму Ганта для проекта с индивидуальными диаграммами.

На рис. 4.4 стрелки представляют временные зависимости. Вначале все зависимости в диаграмме были созданы в режиме «от конца к началу» (finish-to-start — FS). Далее между `Subtask2.2` (подзадача 2.2) и `Subtask2.3` (подзадача 2.3) была установлена зависимость с задержкой, а между `SummaryTask1` (суммарная задача 1) и `Task4` (задача 4) установлена зависимость с наложением. Некоторые задачи не имеют никаких зависимостей и могут выполняться параллельно (например, `Task1` и `Task2`).

Обычно задачи планируются по принципу «*как можно скорее*» (As Soon As Possible — ASAP). Это означает, что выполнение задачи будет намечено как можно скорее, учитывая зависимости с задачами-предшественниками и при условии наличия ресурсов. Все задачи на рис. 4.4 намечены на основе принципа ASAP.

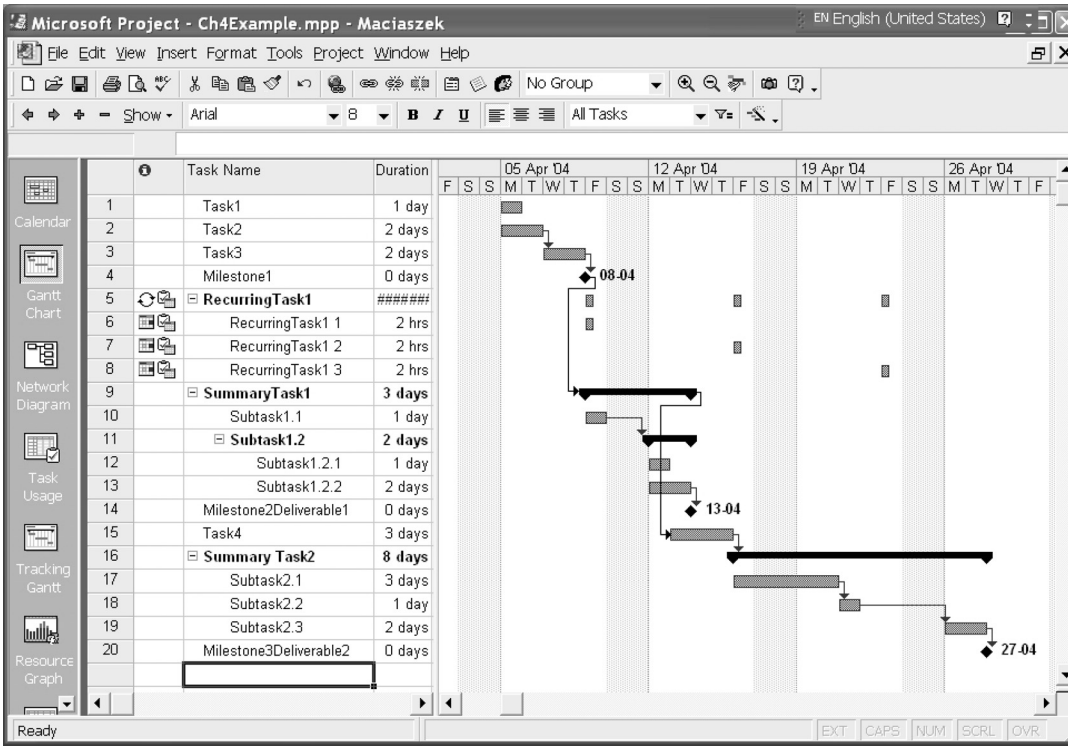


Рис. 4.4. Планирование задач с помощью диаграммы Ганта
Источник: образ экрана Microsoft® Office Project (2003), перепечатано по разрешению Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

Как противоположность этому, задачи могут быть намечены по принципу «*как можно позже*» (As Late As Possible — ALAP). Это полезно в двух ситуациях. Во-первых, принцип ALAP позволяет задаче получить максимальное количество ресурсов (данных) от более ранних задач до того, как начнется ее реализация. Наличие большого количества информации может позволить избежать дорогостоящих ошибок при выполнении задачи. Во-вторых, принцип ALAP позволяет произвести подсчет, сколько задач может быть отсрочено без задержки даты окончания проекта (то есть без увеличения критического пути проекта).

В некоторых случаях может потребоваться наложить дополнительные ограничения, которые определяют конкретные даты начала или конца задач. Они называются **негибкими ограничениями**. Microsoft Project позволяет задать шесть видов негибких ограничений:

- должен закончиться в... (Must Finish On — MFO);
- должен начаться в... (Must Start On — MSO);
- закончить не ранее чем... (Finish No Earlier Than — FNET) — негибкое ограничение для проектов, планируемых с конечной даты;

- закончить не позже чем... (Finish No Later Than — FNLT) — негибкое ограничение для проектов, планируемых с даты начала;
- начать не ранее чем... (Start No Earlier Than — SNET) — негибкое ограничение для проектов, планируемых с конечной даты;
- начать не позже чем... (Start No Later Than — SNLT) — негибкое ограничение для проектов, планируемых с даты начала.

Негибкие ограничения должны использоваться только в крайних случаях, поскольку они нарушают гибкость планирования проекта. На рис. 4.4 в списке задач нет никаких негибких ограничений.

4.2.3. Ресурсы и календари ресурсов

Каждой задаче должны быть назначены ресурсы. Грубо **ресурсы** можно расклассифицировать на рабочие и материальные. **Рабочие ресурсы** состоят из людей и оборудования, включая аппаратное и программное обеспечение. **Материальные ресурсы** — это расходные материалы и поставки (типа бумаги для принтеров). Задача может требовать многих ресурсов. С точки зрения людских ресурсов над задачей могут работать много людей. В зависимости от сложности проекта и уровня детализации, требуемой в плане, может быть желательно выделить задаче только *ответственные* людские ресурсы (то есть, только людей, ответственных за задачу).

Назначение ресурсов включает и гибкость, и ограничения планирования. С одной стороны, возможно «перемещать ресурсы», чтобы минимизировать нежелательные зависимости между задачами. С другой стороны, задачи могут быть отсрочены из-за необходимости ожидать наличие требуемого ресурса.

Назначение ресурсов обладает важной *функцией отслеживания* (раздел 4.4). С назначенными ресурсами можно отслеживать количество работы, выполненной людьми и оборудованием, а также контролировать использование материалов. Не полностью использованные ресурсы могут быть переданы задачам, нуждающимся в них или сомнительным. Стоимости ресурсов также могут быть отслежены.

До назначения ресурсов задачам эти ресурсы должны быть определены. Как показано на рис. 4.5, определение ресурса включает данное ему имя, определение его типа и указание числа доступных *единиц ресурса*. Последняя характеристика применяется только к рабочим ресурсам. В случае людских ресурсов рассматриваются отдельные индивидуумы, а не группы людей; максимальное значение единицы ресурса — 100 процентов. 100 процентов показывает ресурс, нанятый на основании полной рабочей недели. Для служащих, нанятых на половину недели, максимальное значение единицы ресурса — 50 процентов.

Служащий полной рабочей недели, со 100-процентной единицей, тем не менее, может и не работать над задачей все 100 процентов времени (скажем, 8 часов в день). Это зависит от **календаря ресурса** (в противоположность календарю проекта). Календарь ресурса разрешает вводить любые дни и часы, когда ресурс не доступен, а также задать варианты работы, отличные от определенных в календаре проекта (такие как работа в выходные дни) — рис. 4.6.

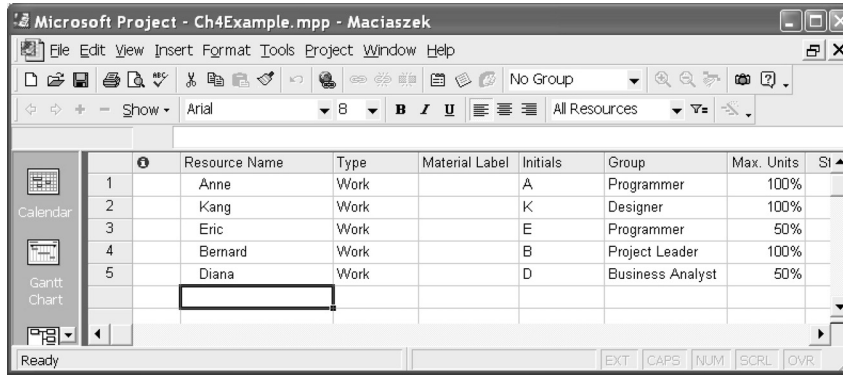


Рис. 4.5. Список ресурсов

Источник: образ экрана Microsoft® Office Project (2003), перепечатано по разрешению Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

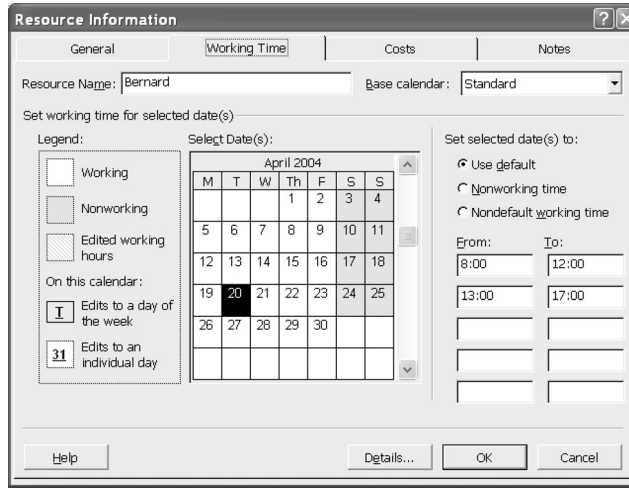


Рис. 4.6. Календарь ресурса

4.2.4. Планирование, определяемое трудозатратами, в виде ленточной диаграммы

Назначение ресурсов задачам основано на максимальном количестве рабочего времени, которое ресурс может предложить. Максимальное рабочее время определяется максимальным доступным числом единиц ресурса и всеми ограничениями в календаре ресурса. Если ресурс — частично занятый служащий (скажем, 50 процентов от единицы ресурса) без ограничений в календаре ресурса, то (в данном примере) ресурс доступен в течение 4 часов в день. Если работа, назначенная ресурсу, превышает его ежедневную доступность, то такой ресурс распределен избыточно.

Назначение дополнительных ресурсов задаче может сократить ее продолжительность и, следовательно, может повлиять на график проектных работ. Подход к планированию, когда продолжительность выполнения задач может изменяться (как в одну, так и в другую сторону) добавлением или удалением ресурсов, известен как **планирование, определяемое трудозатратами**. В таком подходе трудозатраты, требуемые для завершения задачи, остаются неизменными, но продолжительность задачи может изменяться.

Например, продолжительность `Task3` на рис. 4.4 — 2 дня. Это означает 16 часов трудозатрат. На рис. 4.7 `Task3` использует 50 процентов от ресурса Kang (Кэнг), что означает 4 часа в день. Это в свою очередь означает, что ресурсу Kang нужно 4 дня, чтобы закончить `Task3`. В планировании, определяемом трудозатратами, используемом в диаграмме Ганта на рис. 4.7, график проектных работ отрегулирован на основе требуемых трудозатрат и имеющихся ресурсов.

Если требуется, чтобы продолжительности задач не изменялись при использовании планирования, определяемого трудозатратами, планировщик должен назначить количество ресурса, которое может обеспечить трудозатраты в течение предписанной продолжительности. Например, `Task1` и `Task2` получают два ресурса, каждый из которых может выполнить 50 процентов от требуемых трудозатрат. В результате продолжительности этих двух задач не изменились. Тот же самый принцип используется и в других местах диаграммы Ганта на рис. 4.7.

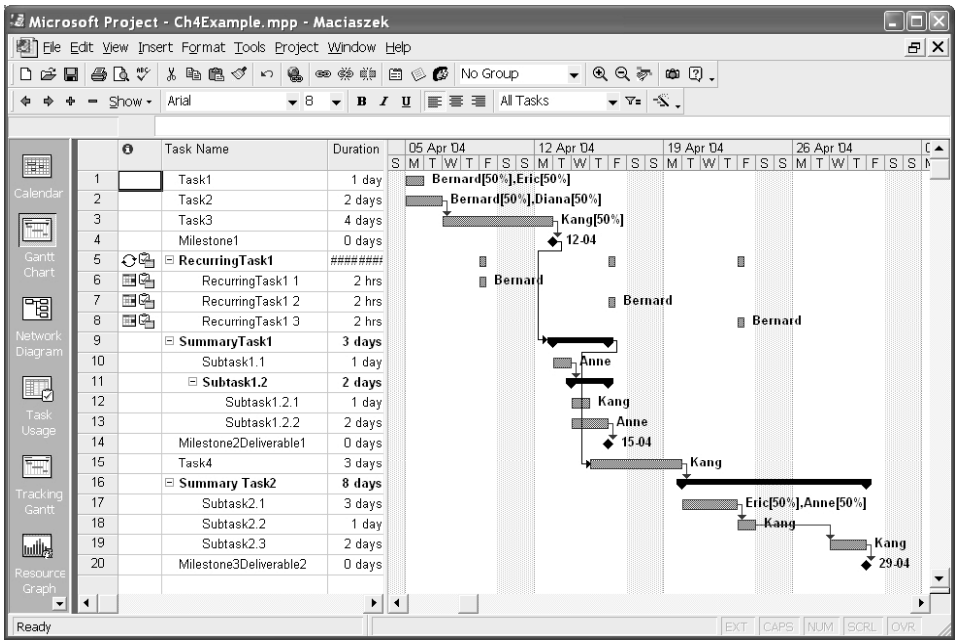


Рис. 4.7. Планирование, определяемое трудозатратами, на диаграмме Ганта
Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

4.2.5. Неполное и избыточное распределение ресурсов

Инструментальные средства управления проектами могут обеспечить различные представления информации относительно планирования проекта. **Использование ресурсов** — одно из таких представлений (рис. 4.8). Представление использования ресурсов включает часы запланированной работы каждого ресурса и часы, распределенные между задачами. Анализ использования ресурсов (распределения) позволяет определить *неполное и избыточное распределение ресурсов*. На рис. 4.8 нет никаких избыточно распределенных ресурсов, но ясно, что все они — не полностью распределенные в том смысле, что не имеется никакой работы, полностью определенной для них на несколько дней.

Неполное распределение ресурса порой является фиктивным. Ресурсы часто используются во многих проектах, и единственный график проектных работ — не свидетельство неполного использования ресурса. Также вероятно, что некоторые аспекты работы ресурсов не охвачены какими-то планами управления.

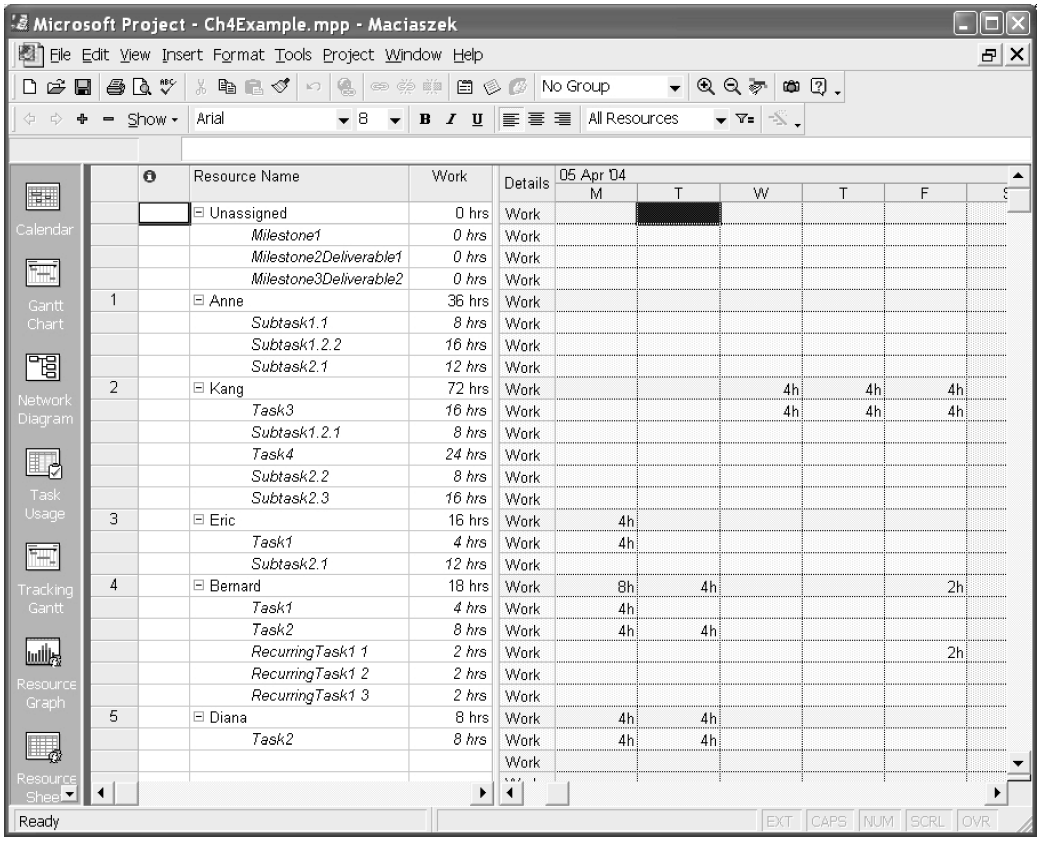


Рис. 4.8. Использование ресурсов

Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

В противоположность этому чрезмерное использование ресурса создает непосредственные проблемы планирования и на него следует обратить внимание. Рассмотрим изменение распределения ресурсов для `Task1` на рис. 4.9. Ресурс Diana (Диана) заменил ресурс Eric (Эрик). На первый взгляд, все выглядит прекрасно, поскольку Diana обеспечивает 50 процентов трудозатрат в `Task1` и в `Task2`, что означает 4 часа в каждой задаче. Однако Diana — частично занятый ресурс на рис. 4.5; это означает, что Диана не может работать 8 часов в понедельник 5 апреля 2004 г.

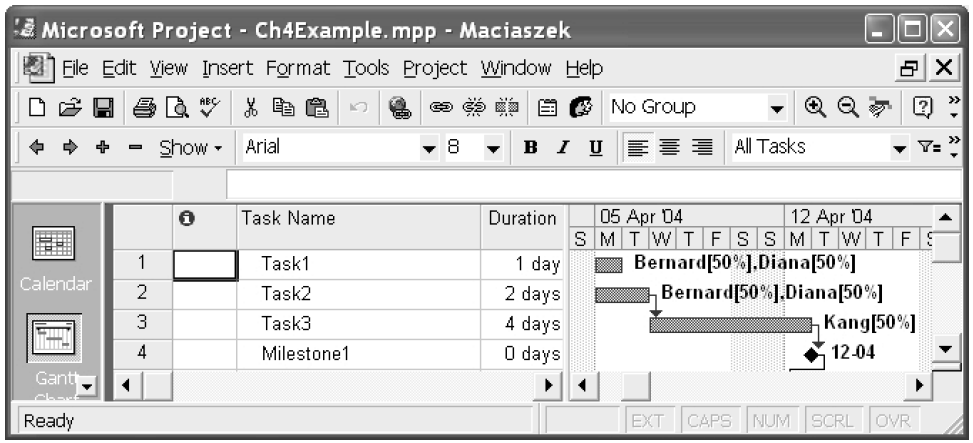


Рис. 4.9. Изменение в распределении ресурса, которое приводит к избыточному распределению Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

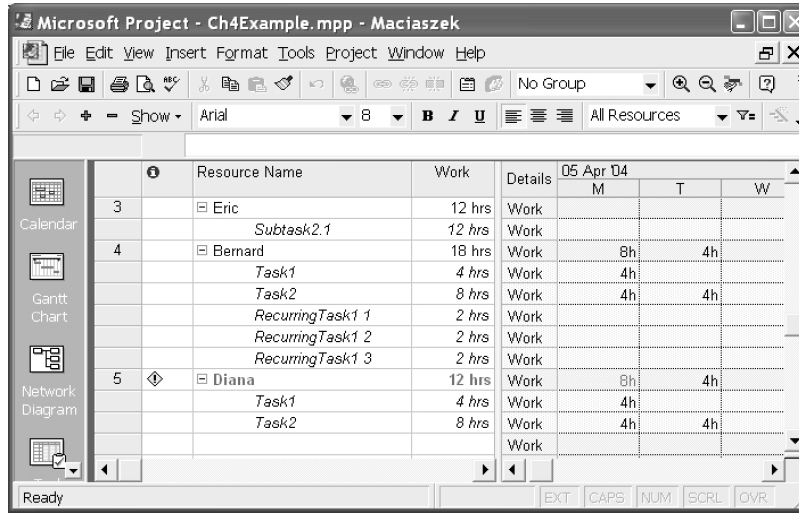


Рис. 4.10. Использование ресурсов, показывающее избыточное распределение ресурсов Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

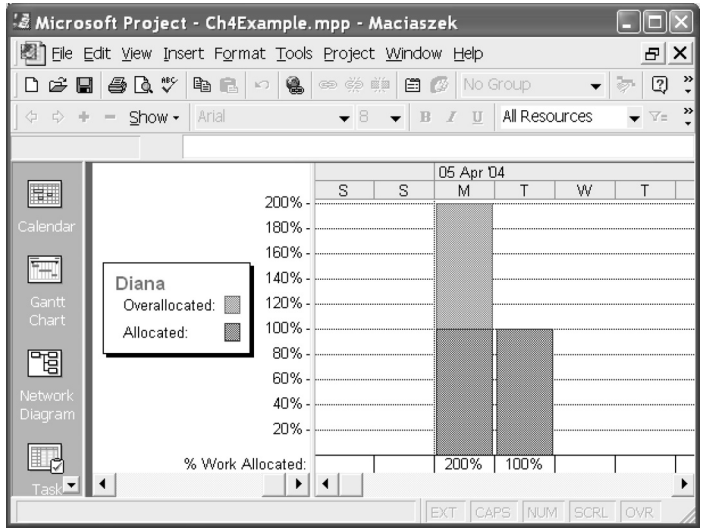


Рис. 4.11. График ресурсов для избыточно распределенного ресурса
Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

Инструментальное средство управления проектами немедленно замечает избыточное распределение ресурса Diana. Рис. 4.10 — представление использования ресурса, показывающее Diana как избыточно распределенный ресурс (показанный красным цветом и знаком предупреждения в столбце «information» (информация)). Распределение в 8 часов в понедельник 5 апреля 2004 г. вдвое больше, чем может обеспечить Diana как работник с 50-процентной загрузкой.

Тот факт, что ресурс является избыточно распределенным, может быть отображен на графике ресурса, как показано на рис. 4.11. График ясно показывает, что работа, размещенная для Diana 5 апреля 2004 г., составляет 200 процентов к тому, что она может сделать (здесь снова окраска полос указывает на это).

4.3. Оценка бюджета проекта

Оценка бюджета проекта — тернистая проблема. Имеется много предложенных подходов к оценке затрат, но только пара из них имеет научное обоснование. Кроме того, научные подходы дорогостоящи. Имеется компромисс между трудозатратами, необходимыми для достижения более точного и достоверного бюджета, и величинами, которые этот бюджет обеспечивает.

Как замечено Ленцем и Риа [59] и показано на рис. 4.12, трудозатраты по формированию бюджета быстро растут с уровнем детализации, рассматриваемой при определении факторов, которые влияют на вычисления. Точность бюджета, по-видимому, устойчиво улучшается с ростом детализации. Удобство сопровождения бюджета, однако, является высоким для упрощенных методов оценки, но начинает в некоторый момент быстро уменьшаться, ког-

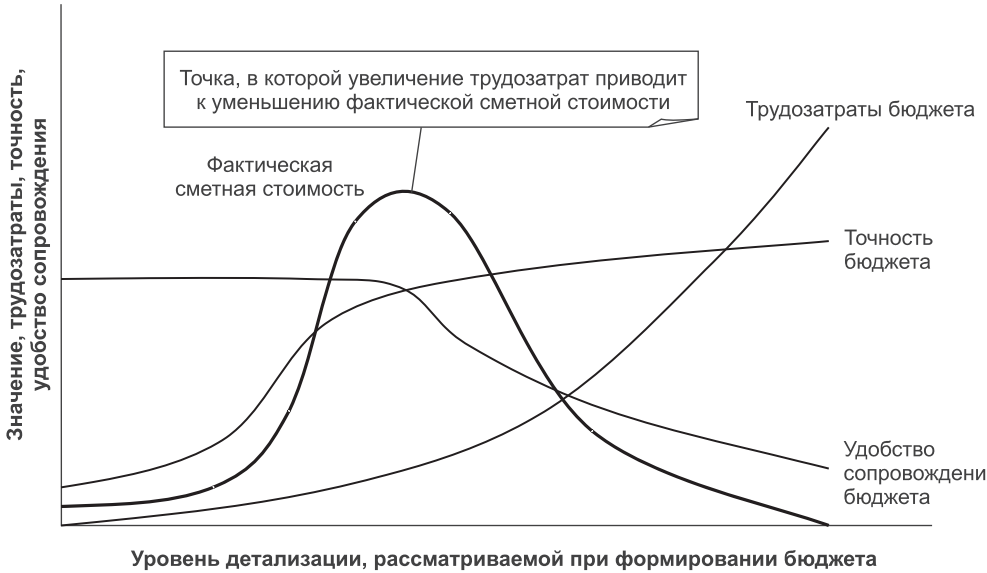


Рис. 4.12. Фактическая сметная стоимость по сравнению с затратами на создание бюджета

да детализация переходит в стадию быстрых и корректных уточнений. Реальная сметная стоимость сама по себе является компромиссом между точностью и удобством сопровождения. До некоторой точки совокупный эффект этих трех факторов улучшает сметную стоимость. Однако на некотором уровне детализации фактическая сметная стоимость начинает уменьшаться.

Графики типа, показанного на рис. 4.12, являются некоторым оправданием использованию таких методов оценки затрат, как экспертная оценка или оценка по аналогии [97]. **Экспертная оценка** — метод, в котором оценки запрашивают от различных экспертов, а затем они обсуждаются и выверяются, чтобы достичь согласованного результата. **Оценка по аналогии** может использоваться, когда имеются подобные законченные проекты в той же самой предметной области и фактические их затраты известны. Бюджет проекта тогда оценивается по аналогии с тем, что произошло в прошлом.

Более сложные методы оценки бюджета могут быть поделены на две большие группы. Первая группа состоит из восходящих методов, которые рассматривают задачи и ресурсы, размещенные в подготовленном графике проектных работ ранее, и получают бюджет, суммируя затраты на ресурсы плюс другие фиксированные расходы. Эти методы обсуждаются ниже под названием «оценка бюджета на основе графика выполнения».

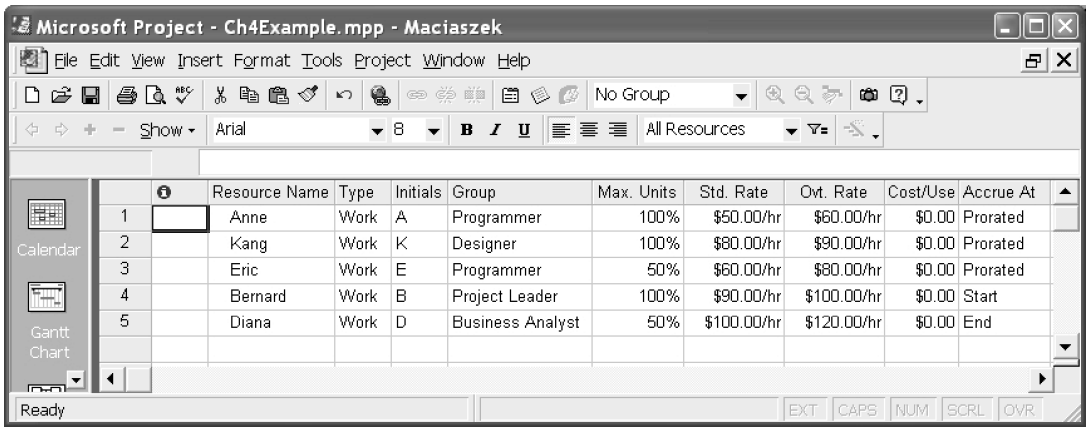
Второй надежный подход к оценке бюджета известен как «алгоритмическая оценка бюджета». В этом подходе используется для получения затрат некоторая оцененная метрика размера ПО, типа числа строк кода или числа функциональных единиц. Затраты получают, задавая в математическую модель значения метрик и большого числа параметров настройки. Сама модель является результатом обширных изучений прежних проектов.

4.3.1. Оценка бюджета на основе графика выполнения

Оценка бюджета на основе графика выполнения предполагает, что существует план исходных данных с ресурсами, распределенными по задачам. Он называется планом на основе **исходных данных** или просто планом. Поскольку планы имеют тенденцию изменяться, инструментальное средство управления проектами может обслуживать различные исходные данные и формировать отдельные оценки бюджета для каждого набора исходных данных.

Теоретически кажется, что оценку бюджета на основе графика выполнения легко произвести. Так как ресурсы в исходных данных назначены задачам, все, что остается сделать, это назначить стоимости затрат для ресурсов, добавить накладные расходы и другие фиксированные затраты и воспользоваться инструментальным средством управления проектами, чтобы вычислить бюджет для проекта. Дьявол запрятан в деталях. Например, ресурсы могут накапливать затраты в различные моменты времени в процессе выполнения задачи. Нормы и фиксированные затраты могут быть разными в различные периоды времени. Имеются некоторые узаконенные правила, влияющие на ставки заработной платы и затраты. На вычисления могут влиять принципы бухгалтерского учета и т. д. Однако первоначальное вычисление бюджета тривиально по сравнению с отслеживанием бюджета в ответ на изменения плана, изменения исходных данных и других проектных условий (раздел 4.4).

Как сказано, оценка бюджета требует ввода **ставок** и других затрат для рабочих и материальных ресурсов. Как показано на рис. 4.13, кроме **стандартных ставок** могут быть ставки за сверхурочное время и ставки за использование. **Ставки за сверхурочную работу** оплачиваются из рабочего ресурса, если ресурс используется за пределами обычных рабочих часов. Сверхурочная работа — не дополнительная работа для задачи, но она может использо-



	Resource Name	Type	Initials	Group	Max. Units	Std. Rate	Ovt. Rate	Cost/Use	Accrue At
1	Anne	Work	A	Programmer	100%	$50.00/hr	$60.00/hr	$0.00	Prorated
2	Kang	Work	K	Designer	100%	$80.00/hr	$90.00/hr	$0.00	Prorated
3	Eric	Work	E	Programmer	50%	$60.00/hr	$80.00/hr	$0.00	Prorated
4	Bernard	Work	B	Project Leader	100%	$90.00/hr	$100.00/hr	$0.00	Start
5	Diana	Work	D	Business Analyst	50%	$100.00/hr	$120.00/hr	$0.00	End

Рис. 4.13. Назначение ставок оплаты и других параметров затрат для ресурсов
Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

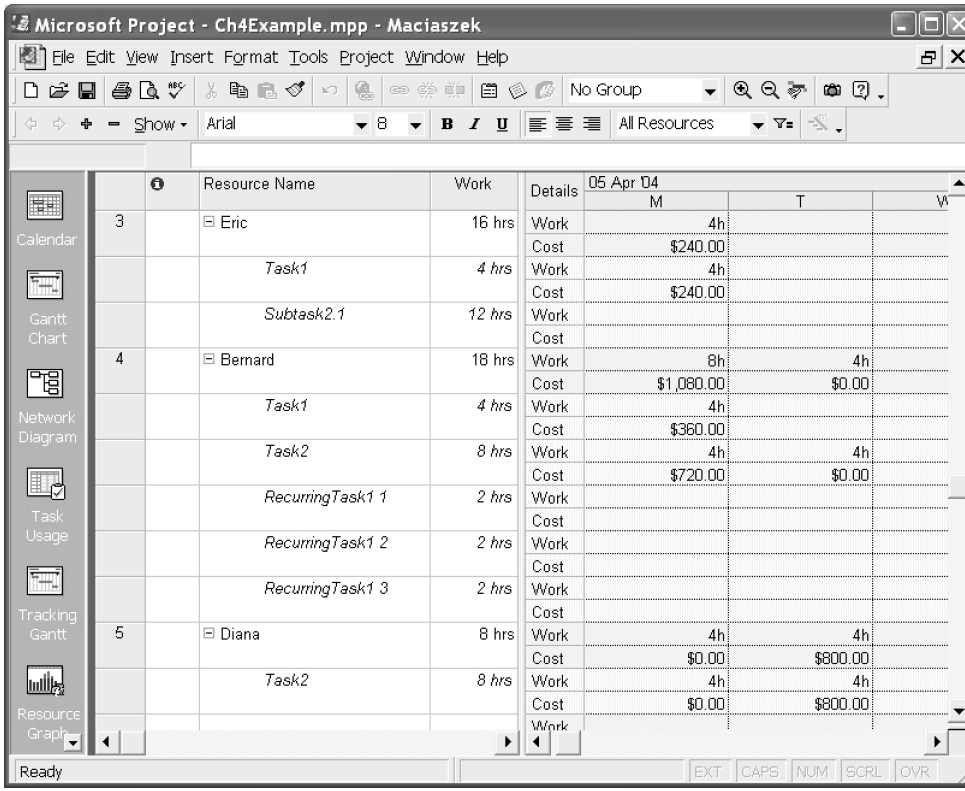


Рис. 4.14. Затраты ресурсов с различными методами наращивания
Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

ваться, чтобы сократить продолжительность ее выполнения. **Ставки за использование** фактически не являются ставками, а совокупностью затрат, связанных с использованием ресурса. Они могут использоваться как замена других ставок или в дополнение к ним.

Методы наращивания затрат должны быть определены для ресурсов наряду со ставками и другими затратами. Наращивание можно выполнять с начала задачи, с конца задачи или пропорционально (как выполняется работа). Пропорциональный метод наиболее часто используется и накапливает затраты в зависимости от процента завершения задачи. Методы наращивания применяются как к гибким затратам, так и к фиксированным затратам. На рис. 4.13 пропорциональный метод используется для первых трех ресурсов. Метод наращивания для ресурса Bernard выполняется с начала задачи, а для Diana — с конца задачи.

Рис. 4.14 демонстрирует, как различные методы наращивания влияют на вычисление затрат. Затраты ресурса Eric накапливаются по принципу «день ото дня». Затраты ресурса Bernard накоплены в первый день, когда начинается задача. В противоположность этому затраты ресурса Diana накоплены только тогда, когда задача завершается.

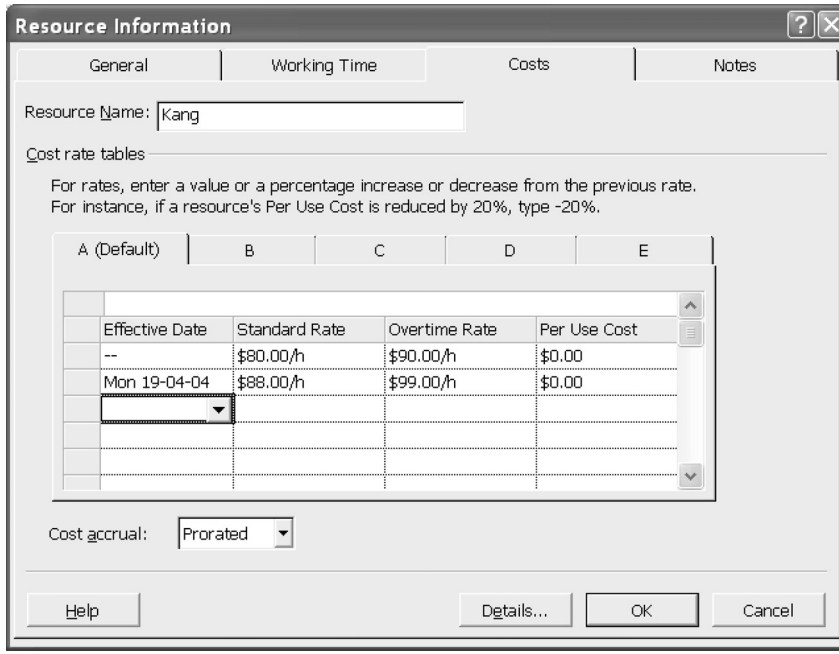


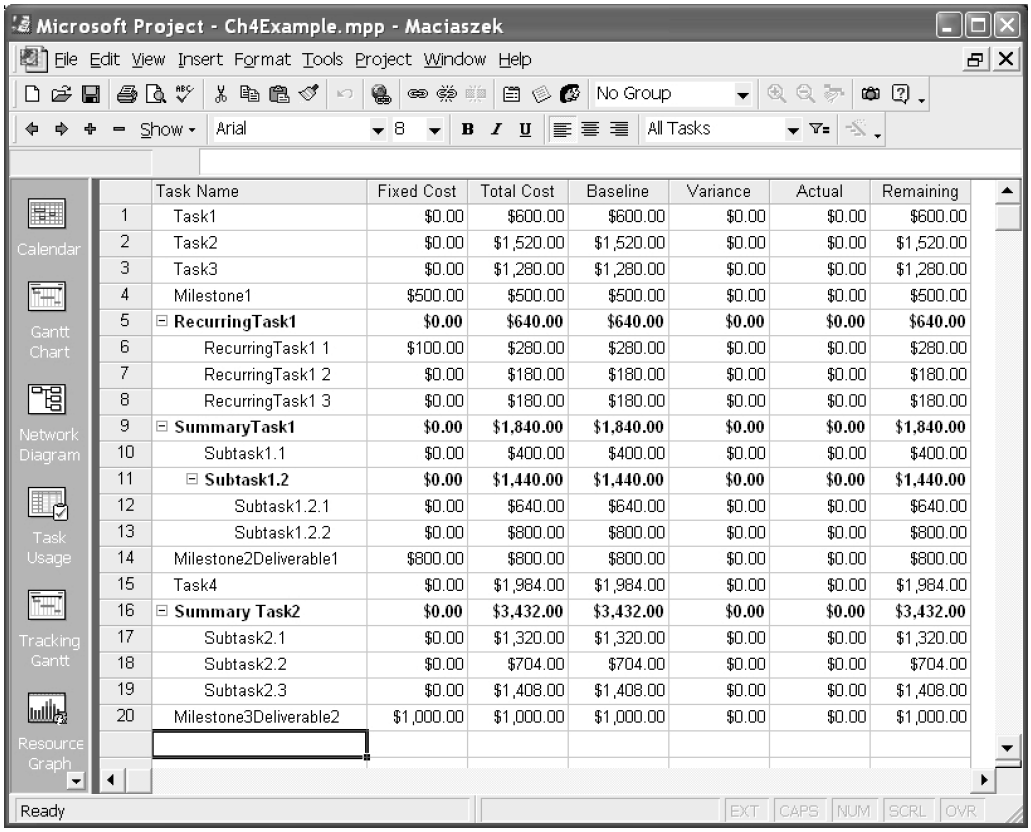
Рис. 4.15. Изменения ставок со временем

Ставки и другие затраты могут изменяться в зависимости от времени, когда использован ресурс (например, в результате повышений платы). Рис. 4.15 демонстрирует диалоговое окно, которое может использоваться, чтобы определить изменения, вступающие в силу с некоторой конкретной даты.

Как только затраты ресурсов будут известны, можно вычислить общие стоимости задач, как показано на рис. 4.16. Обратите внимание, что затраты на задачи могут включать фиксированные расходы, назначенные скорее на задачу, чем на ресурс, использованный задачей. Например, фиксированные затраты могут быть в отношении расходных материалов и подобных расходов. На рис. 4.16 фиксированные расходы добавлены ко всем контрольным точкам (которые не имеют никаких размещенных ресурсов). Фиксированные затраты также добавлены к `RecurringTask1`. Обратите внимание, что текущий бюджет был сохранен как исходные данные, и что проект не затратил никаких денег, как видно из столбца `Actual` (фактическое значение).

Инструментальное средство управления проектами может также обслуживать распределение затрат между ресурсами (рис. 4.17). Как и раньше, распределение затрат ресурсов все же не является совершенным.

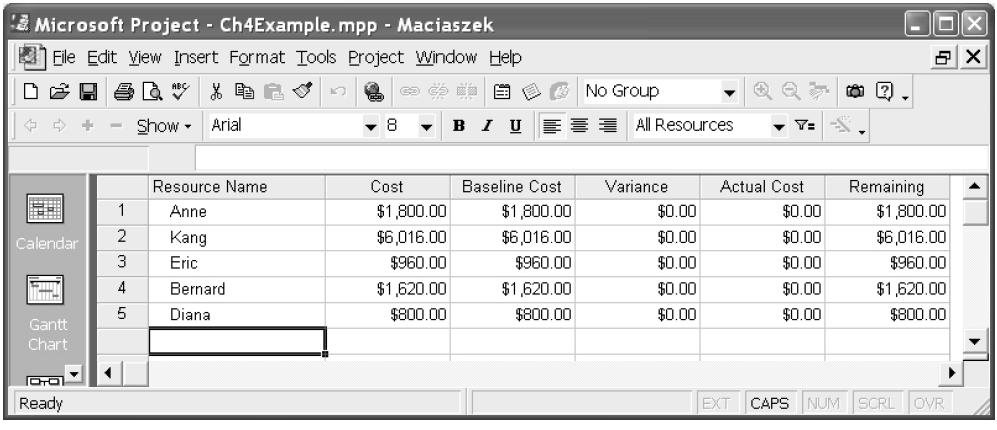
Планировщик теперь находится в состоянии вычисления затрат на весь проект. Рис. 4.18 показывает такое вычисление. Поскольку проект все еще не начался, текущие, фактические и остаточные затраты одни и те же. Фактические затраты нулевые. Согласно плану проект должен завершиться в пределах 19 дней, иметь рабочую нагрузку 150 часов и затраты в 13 596 долларов.



	Task Name	Fixed Cost	Total Cost	Baseline	Variance	Actual	Remaining
1	Task1	$0.00	$600.00	$600.00	$0.00	$0.00	$600.00
2	Task2	$0.00	$1,520.00	$1,520.00	$0.00	$0.00	$1,520.00
3	Task3	$0.00	$1,280.00	$1,280.00	$0.00	$0.00	$1,280.00
4	Milestone1	$500.00	$500.00	$500.00	$0.00	$0.00	$500.00
5	**RecurringTask1**	**$0.00**	**$640.00**	**$640.00**	**$0.00**	**$0.00**	**$640.00**
6	RecurringTask1 1	$100.00	$280.00	$280.00	$0.00	$0.00	$280.00
7	RecurringTask1 2	$0.00	$180.00	$180.00	$0.00	$0.00	$180.00
8	RecurringTask1 3	$0.00	$180.00	$180.00	$0.00	$0.00	$180.00
9	**SummaryTask1**	**$0.00**	**$1,840.00**	**$1,840.00**	**$0.00**	**$0.00**	**$1,840.00**
10	Subtask1.1	$0.00	$400.00	$400.00	$0.00	$0.00	$400.00
11	**Subtask1.2**	**$0.00**	**$1,440.00**	**$1,440.00**	**$0.00**	**$0.00**	**$1,440.00**
12	Subtask1.2.1	$0.00	$640.00	$640.00	$0.00	$0.00	$640.00
13	Subtask1.2.2	$0.00	$800.00	$800.00	$0.00	$0.00	$800.00
14	Milestone2Deliverable1	$800.00	$800.00	$800.00	$0.00	$0.00	$800.00
15	Task4	$0.00	$1,984.00	$1,984.00	$0.00	$0.00	$1,984.00
16	**Summary Task2**	**$0.00**	**$3,432.00**	**$3,432.00**	**$0.00**	**$0.00**	**$3,432.00**
17	Subtask2.1	$0.00	$1,320.00	$1,320.00	$0.00	$0.00	$1,320.00
18	Subtask2.2	$0.00	$704.00	$704.00	$0.00	$0.00	$704.00
19	Subtask2.3	$0.00	$1,408.00	$1,408.00	$0.00	$0.00	$1,408.00
20	Milestone3Deliverable2	$1,000.00	$1,000.00	$1,000.00	$0.00	$0.00	$1,000.00

Рис. 4.16. Затраты на задачи

Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation



	Resource Name	Cost	Baseline Cost	Variance	Actual Cost	Remaining
1	Anne	$1,800.00	$1,800.00	$0.00	$0.00	$1,800.00
2	Kang	$6,016.00	$6,016.00	$0.00	$0.00	$6,016.00
3	Eric	$960.00	$960.00	$0.00	$0.00	$960.00
4	Bernard	$1,620.00	$1,620.00	$0.00	$0.00	$1,620.00
5	Diana	$800.00	$800.00	$0.00	$0.00	$800.00

Рис. 4.17. Затраты ресурсов

Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

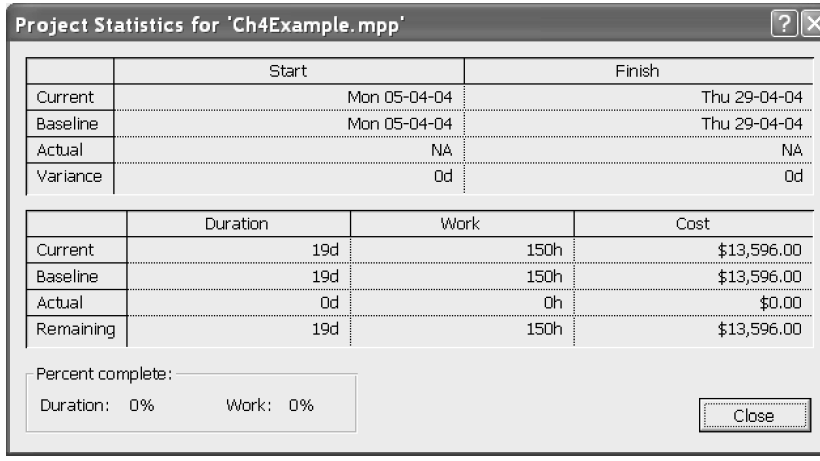


Рис. 4.18. Затраты проекта

4.3.2. Алгоритмическая оценка бюджета

Ну, хорошо, оценка бюджета на основе графика выполнения дает приемлемые значения при разумных трудозатратах (рис. 4.12). Однако на практике оценка бюджета не должна полагаться только на одну технологию оценки. Точность оценки бюджета должна быть проверена путем применения другой технологии. Проблема напоминает использование различных цен для сервисов, предоставляемых продавцами: единая цена может оказаться недостаточной для конкретного случая.

Алгоритмическая оценка бюджета — вторая лучшая технология оценки после оценки на основе графика выполнения. Она использует некоторую меру (метрику) масштаба задачи и применяет к ней полученный опытным путем алгоритм, чтобы определить бюджет, необходимый для решения задачи. В производственных отраслях типа выпуска автомобилей проблема проста. Учитывая время и стоимость создания одного автомобиля, можно получить время и стоимость всего производственного «проекта».

В технических дисциплинах, и в особенности в разработке ПО, проблема намного хитрее. Для начала технический проект производит единственное уникальное изделие, а не большое количество одного и того же продукта. Даже если может быть получена метрика, которая позволяет разложить изделие на несколько компонентов, эти компоненты совершенно разные. ПО простым образом не масштабируется (раздел 1.1.6). Каждый компонент требует различных трудозатрат, даже если он столь же примитивен, как строка кода или функциональная единица. Большинство затрат относится к рабочим ресурсам, и людским ресурсам в частности, но не к материальным ресурсам. Наконец, так как каждое изделие уникально, трудно заранее предсказать размер проблемы.

Принципы алгоритмических моделей

Алгоритмические модели используют полученные опытным путем формулы для оценки стоимости людских ресурсов (**трудозатрат**) как функции от размера проекта. **Размер** обычно выражается в строках кода (line of code — LOC), в функциональных единицах (function point — FP) или в объектных единицах (object point — OP). Оценивается также и размер ПО [81].[1]

Строки кода определяются числом тысяч (K — kilo-) исходных операторов программы за исключением комментариев. По этой причине LOC-переменная показывается в формулах как KLOC. KLOC — одна из возможных наиболее детальных калибровочных метрик и потенциально может дать наиболее важные оценки бюджета (рис. 4.12). Минусом здесь является то, что в современном производстве ПО, которое использует автоматическую генерацию кода и повторное использование кода, очень трудно оценить новые строки кода и отличить их от многократно используемого или сгенерированного кода.

Функциональные единицы дают более высокий уровень декомпозиции кода, чем строки кода (то есть, они обеспечивают менее детальную калибровочную информацию). Функциональные единицы — не модули ПО или компоненты, как можно предполагать по их имени. В наиболее частом использовании функциональные единицы вычисляют, анализируя ПО с точки зрения пяти особенностей (входы, выходы, файлы данных, запросы и внешние интерфейсы), определяя их число, одновременно корректируя вычисления в зависимости от представляемой сложности этих особенностей.

Объектные единицы находятся даже на еще более высоком уровне декомпозиции кода, чем функциональные единицы. Подобно функциональным единицам их получают, используя вычисленное количество различных особенностей ПО. Особенностями здесь являются экраны пользовательского интерфейса, сообщения ПО и компоненты ПО. Как и с функциональными единицами, вычисление корректируют, используя весовые коэффициенты, отражающие сложность особенностей. Вычисление также корректируется для удаления многократно используемых объектных единиц. Это приводит к мере, называемой новыми объектными единицами (new object point — NOP).

Имея калибровочную информацию, алгоритмическая модель оценки затрат будет обладать следующей номинальной формой:

$$\texttt{effort} = \texttt{c} * \texttt{size}^{\texttt{k}},$$

где `effort` — трудозатраты; `size` — размер ПО.

Константы `c` и `k` устанавливаются опытным путем и задаются самой моделью. Показатель `k` отражает сложность задачи и непропорциональное увеличение трудозатрат в больших проектах. Этот показатель обычно находится в диапазоне от 1 до 1.5. Множитель `c` — другой коэффициент вычисления

[1] К сожалению, перевод на русский язык терминов «function point» и «object point» еще не устоялся. Так, первый термин переводится и как «функциональная точка» (дословный, но не очень понятный перевод), и как «функциональная единица», и как «единица функциональности», и т. д. Мы будем использовать термин «функциональная единица», т. е. единица измерения параметра, определяющего количество выполняемых ПО функций. Аналогично термин «object point» будем переводить как «объектная единица», т. е. единица измерения параметра, определяющего количество объектов в ПО. (*Прим. перев.*)

проектной сложности, основанной на оценках атрибутов типа требуемой надежности изделия, навыков коллектива, доступных инструментальных средств ПО, и т. д.

Наиболее известные алгоритмические модели — так называемые COCOMO-модели. Аббревиатура COCOMO получена из COnstructive COst MOdel — конструктивная модель затрат [8, 11, 34, 81, 92, 96]. Имеются две COCOMO-модели, обе разработаны под руководством Барри Боэма. Первоначальная модель, разработанная в 1980-х гг., теперь известна как COCOMO 81. Улучшенная версия, опубликованная в 2000 г., известна как COCOMO II.

COCOMO 81

COCOMO II эффективней **COCOMO 81** почти для всех традиционных проектов ПО, использующих языки третьего поколения типа КОБОЛ или С. Однако есть педагогические основания объяснить принципы COCOMO 81 перед рассмотрением COCOMO II.

Фактически COCOMO 81 представляет множество трех подходов к оценке затрат. Три подхода используются для небольших, среднего размера и сложных систем. Каждый подход имеет базовую формулу оценки, основанную на предсказанном размере кода. В действительности, имеется ряд формул, позволяющих выбрать из них нужную для каждого подхода на основе характеристик проекта. Эти три способа и связанные с ними базовые формулы следующие [9]:

- Базовый (или основной) COCOMO:

$$\texttt{effort} = 3.2 * \texttt{size}^{1.05}$$

- Промежуточный (или полуизолированный) COCOMO — этот способ включает в оценку ряд коэффициентов, определяющих величину затрат (или коэффициентов затрат), чтобы масштабировать трудозатраты номинальной разработки. Эти коэффициенты основаны на оценках изделия, аппаратного обеспечения, персонала и атрибутов проекта:

$$\texttt{effort} = 3.0 * \texttt{size}^{1.12}$$

- Продвинутый (или вложенный) COCOMO — этот способ получен из промежуточной версии, но коэффициенты затрат оценивают воздействия на каждой стадии жизненного цикла разработки:

$$\texttt{effort} = 2.8 * \texttt{size}^{1.20}$$

Например, используя базовый COCOMO и предполагая, что система будет состоять из 100000 строк кода (100 KLOC), оцененные трудозатраты, выраженные в человеко-месяцах:

$$\begin{aligned}\texttt{effort} &= 3.2 * 100^{1.05}\\ &= 3.2 * 125.9\\ &\approx 403\ \texttt{person-months}\end{aligned}$$ (человеко-месяцев)

Одним из наиболее интересных косвенных вкладов COCOMO 81 было определение относительной важности различных **коэффициентов затрат** в стоимости проекта и необходимых трудозатратах [8]. Например, диапазон значений для коэффициента затрат, известного как «сложность изделия», говорит, что различие между простым и сложным изделием характеризуется коэффициентом 2.36 (1.65/0.70) в оценке затрат. Если значение этого

коэффициента было бы единственным фактором дифференциации между двумя изделиями, то трудозатраты по разработке сложного изделия будут в `2.36` раз больше, чем для простого изделия.

Для сравнения некоторые другие коэффициенты затрат следующие (в порядке уменьшения важности) [9]:

- «аналитическая способность» — коэффициент `2.06 (1.46/0.71)`;
- «квалификация программистов» — коэффициент `2.03 (1.42/0.70)`;
- «требуемая надежность ПО» — коэффициент `1.87 (1.40/0.75)`;
- «ограничения на время выполнения» — коэффициент `1.66 (1.66/1.00)`;
- «опыт создания приложений» — коэффициент `1.57 (1.29/0.82)`;
- «использование современных методов программирования» — коэффициент `1.51 (1.24/0.82)`;
- «опыт использования языка программирования» — коэффициент `1.20 (1.14/0.95)`.

Можно вынести интересные уроки из изучения вышеупомянутых чисел. Будучи получены на эмпирических исследованиях последней четверти прошлого столетия, приведенные значения коэффициентов затрат находятся в некотором противоречии с современным производством ПО. Например, нельзя больше соглашаться, что опыт создания приложений дает больший вклад в проект, чем опыт использования языка программирования. В конце концов, можно изучать программирование, даже не работая с компьютером, как самоцель. Получение же опыта создания приложений может быть сделано только «в области» взаимодействия с другими работниками и наблюдая рабочую среду. Даже и тогда опыт, полученный в одной среде, не может быть полностью перенесен в другую среду.

В оценке бюджета на основе графика выполнения (раздел 4.3.1) план используется, чтобы получить бюджет для всего проекта, а также частные бюджеты для отдельных задач и ресурсов. Алгоритмические методы не связывают с планом, за исключением оценки времени разработки, соответствующего полученным трудозатратам (`effort`). Эмпирические формулы оценки полного времени разработки для трех подходов COCOMO будут соответственно [9]:

- простой COCOMO: $time = 2.5 * effort^{0.38}$;
- промежуточный COCOMO: $time = 2.5 * effort^{0.35}$;
- продвинутый COCOMO: $time = 2.5 * effort^{0.32}$.

Ясно, что COCOMO не столь прост, как взятие размера ПО и использование его в формуле, чтобы оценить трудозатраты. Точность оценки в первую очередь определяется точностью коэффициентов затрат и других масштабирующих коэффициентов. Это в свою очередь зависит от пригодности метрик из других проектов. Организация, разрабатывающая ПО, должна знать свою историю (и иметь ее оценку в числах), чтобы планировать свое будущее.

COCOMO 81 обеспечил большой объем показателей прошлых лет от большого количества организаций в надежде, что опыт других организаций может использоваться там, где локальные метрики не пригодны. Это было полезно во

времена, когда КОБОЛ использовался как универсальный язык программирования бизнес-системы, а приложения не были столь разнообразны, как сегодня. Никакой такой исторической информации нет для современных проектов. Каждая организация должна собрать метрики от предыдущих проектов, чтобы иметь хоть какую-то надежду уверенно планировать в будущем.

COCOMO II

COCOMO II [11] основан на тех же принципах, что и COCOMO 81, но он учитывает пару важных недостатков COCOMO 81. Модель была калибрована результатами анализа опытных данных, собранных (первоначально) из 83 проектов ПО. Подобно всем алгоритмическим моделям, COCOMO II требует калибровочной информации. Кроме LOC и FP COCOMO II вводит объектные единицы (object point — OP) как вариант калибровки. Подобно COCOMO 81 COCOMO II использует разнообразные коэффициенты затрат, масштабирующие коэффициенты и другие процедуры настройки.

COCOMO II существенно более сложен, чем COCOMO 81, что отражает:

- возросшую сложность ПО и предметных областей;
- более широкое разнообразие моделей жизненного цикла и технологий разработки ПО;
- пропорциональный рост генерации и повторного использования ПО в сравнении с ручным программированием;
- непрерывный характер планирования и оценки затрат.

В отличие от COCOMO 81, который обеспечивал оценки для жизненного цикла типа водопада и процедурного программирования, COCOMO II приспособлен к итеративным жизненным циклам и объектно-ориентированному программированию. Он признает использование опытных образцов, шаги, короткие циклы, использование генераторов программ, разработку с использованием объединения компонентов и т. д.

COCOMO II чувствителен к стадии жизненного цикла, в которой получаются оценки. Это является достоинством, так как позволяет повысить точность оценки затрат и необходимого времени с продвижением в реализации проекта (рис. 4.1). Соответственно COCOMO II состоит из трех различных моделей, относящихся к трем последовательным стадиям проекта. Модели называются:

- *модель компоновки приложения* (application composition model) — используется на первоначальных стадиях анализа или раннего использования опытных образцов;
- *модель раннего проектирования* (early design model) — используется после завершения анализа требований;
- *пост-структурная модель* (post-architecture model) — используется после того, как станет известен проект структуры ПО.

Модель компоновки приложения использует взвешенные объектные единицы (OP) для калибровочной информации. OP — число ожидаемых экранов пользовательского интерфейса, сообщений ПО и компонентов ПО. Число OP уменьшается на процент повторно используемых объектных еди-

ниц, так что для оценки трудозатрат рассматриваются только новые объектные единицы (new object point — NOP). Формула для NOP:

```
NOP = OP * [(100 − %reuse)/100],
```

где `%reuse` — процент повторно используемых OP.

Таблица 4.1. Метрики производительности в режиме компоновки приложения с помощью COCOMO II

Объединенные опыт разработчиков/навыки и организационная компетентность/навыки	Производительность в NOP/месяц
Очень низкие	4
Низкие	7
Номинальные	13
Высокие	25
Очень высокие	50

Модель компоновки приложения использует только два весовых коэффициента для определения уровня производительности разработки (навыка). Эти два коэффициента — опыт/навыки разработчиков и организационная компетентность/навыки. Эти два коэффициента объединены, чтобы определить пять уровней производительности. Объединенные в пять уровней по производительности «коэффициенты затрат» выражены в терминах NOP, определяющих, что коллектив проектировщиков способен создать за месяц. Пять уровней и соответствующие метрики производительности показаны в таблице 4.1.

Как только метрика производительности будет выбрана, формула для вычисления трудозатрат в модели компоновки приложения, используемой в COCOMO II:

```
effort = NOP/productivity,
```

где `effort` — трудозатраты, `NOP` — новые объектные единицы, `productivity` — производительность.

Например, если число новых объектных единиц — 500, а производительность — 4, то трудозатраты (в человеко-месяцах) будут оценены в 125. В противоположность этому, если производительность — 50, то трудозатраты будут только 10 человеко-месяцев.

Модель раннего проектирования использует функциональные единицы (FP) для калибровочной информации в предсказании стоимости. Практически, однако, COCOMO II (подобно COCOMO 81) формирует стандартные таблицы для преобразования FP в KLOC, и размер, в конечном счете, выражается в KLOC. Формула оценки:

$$effort = c * size^{k} * m + autoeffort,$$

где `effort` — трудозатраты, `size` — размер, `autoeffort` — автотрудозатраты.

Исследования, основанные на COCOMO II, задают для постоянного коэффициента `c` в ранней модели проекта значение, равное 2.5. Величина показателя `k` изменяется от 1.01 до 1.26.

Множитель m отражает влияние коэффициентов затрат на оцениваемые трудозатраты. COCOMO II определяет семь наборов коэффициентов затрат: 1) надежность и сложность изделия, 2) требуемые повторные использования, 3) сложность платформы, 4) способность персонала, 5) опыт персонала, 6) план и 7) средства поддержки. Каждому из этих семи коэффициентов затрат задается значение. Значение 1 означает, что коэффициент затрат несущественен — он не влияет на оценку. В этом случае значение m является результатом перемножения этих семи чисел.

Последний элемент в формуле — autoeffort — вычисляется отдельно согласно его собственному алгоритму. Значение autoeffort — трудозатраты, выполняемые разработчиками при автоматическом формировании кода и интеграции этого кода с программами, созданными вручную. Поскольку остальная часть формулы относится только к коду, созданному вручную, автотрудозатраты добавляются к результату этого вычисления.

Ниже приведена формула вычисления трудозатрат в предположении, что все компоненты формулы были определены заранее.

Таблица 4.2. Коэффициенты затрат в пост-структурной модели COCOMO II

Категория коэффициента затрат	Кодировка коэффициента затрат	Название коэффициента затрат
Атрибуты продукта	RELY	Требуемая надежность системы
	CPLX	Сложность модулей системы
	DOCU	Объем требуемой документации
	DATA	Размер используемой БД
	RUSE	Требуемый процент повторно используемых компонентов
Атрибуты компьютера	TIME	Ограничения времени выполнения
	PVOL	Изменчивость платформы разработки
	STOR	Ограничения на память
Атрибуты персонала	ACAP	Способность анализировать проект
	PCON	Преемственность персонала
	PEXP	Опыт программиста в предметной области проекта
	PCAP	Способности программиста
	AEXP	Опыт аналитика в предметной области проекта
	LTEX	Опыт в языках программирования и инструментальных средствах
Атрибуты проекта	TOOL	Использование инструментальных средств разработки ПО
	SCED	Сжатие плана разработки
	SITE	Объем работы с несколькими файлами и качество связей сайтов

$$\begin{aligned} effort &= 2.5 * 100^{1.15} * 1.3 + 35 \\ &= 2.5 * 199.5 * 1.3 + 35 \\ &\approx 683\ person\text{-}months \quad \text{(человеко-месяцев)} \end{aligned}$$

Пост-структурная модель использует в качестве калибровочной информации для определения затрат отобранные строки кода (LOC). Строки кода отбираются из предыдущих моделей проекта, учитывая два фактора: 1) изменчивость требований и 2) объем повторного использования. Первый фактор рассматривает LOC, необходимые при любой переделке модели из-за изменяющихся требований. Второй фактор отвечает за объем инвестиций, необходимых для поиска и осмысления компонентов многократного использования. Формула оценки та же самая, что и в модели раннего проектирования. Изменения касаются деталей того, как задаются коэффициенты и компоненты.

Пост-структурная модель использует 17 коэффициентов затрат и 5 масштабирующих коэффициентов. Коэффициенты затрат сгруппированы в 4 категориях. Нет необходимости рассматривать все эти коэффициенты. Коэффициенты затрат с номинальным значением 1 не влияют на вычисления. Другие значения перемножаются, чтобы получить величину параметра `m`. Таблица 4.2 содержит список коэффициентов затрат [11, 96].

Масштабирующие коэффициенты используются в пост-структурной модели для получения значения показателя `k`, применяемого к параметру `size`. Каждый коэффициент представляет собой целое число в диапазоне от 5 до 0. Величины складываются, делятся на 100, а результат добавляется к номинальной величине показателя, равной 1.01, чтобы дать значение, которое следует использовать в формуле. Масштабирующие коэффициенты и их объяснение приведены в таблице 4.3 [11, 96].

Алгоритмическая оценка бюджета — важная альтернатива и дополнение к оценке на основе графика выполнения. Каждый подход может рассматриваться как хорошая «проверка на дурака» для других подходов. Использование этих двух подходов превращает оценку стоимости ПО при планировании проекта и ее отслеживание в инженерную дисциплину.

Таблица 4.3. Масштабирующие коэффициенты в пост-структурной модели COCOMO II

Масштабирующий коэффициент	Описание
Наличие прецедентов	Учитывает предшествующий опыт работы организации с подобными проектами
Гибкость разработки	Учитывает объем, в котором клиент проекта вмешивается в разработку. Высокая гибкость означает, что клиент не вмешивается или устанавливает только базовые цели
Определение структуры/рисков	Учитывает объем, в котором для проекта обеспечено управление рисками
Единство коллектива	Учитывает, насколько коллектив разработчиков объединен и эффективен
Компетентность	Учитывает компетентность организации в разработке ПО при выполнении проектов

4.4. Отслеживание выполнения проекта

Отслеживание выполнения проекта включает два основных множества сравнений:

- *план на основе исходных данных* по сравнению с *фактическим планом* (это включает отслеживание работы — работу с исходными данными, распределенными по ресурсам, по сравнению с фактически выполненной работой) и
- *бюджет на основе исходных данных* по сравнению с *фактическим бюджетом*.

Кроме первоначального плана на основе *исходных данных* могут быть созданы планы на *определенные промежутки времени* как контрольные моменты выполнения проекта. План на определенный промежуток времени можно сравнивать с исходными данными, чтобы оценить ход выполнения проекта или его пробуксовку. Он может также использоваться как дополнительные (или новые) исходные данные для оценки текущего плана.

По этой причине отслеживание хода выполнения проекта зависит от ранее созданного и сохраненного плана на основе исходных данных и их бюджета. После того, как проект стартует, по ходу дела начнут накапливаться затраты, которые нельзя было точно запланировать. Новая информация поступает не только от корректировки плана, но также и от системы бухгалтерского учета, которая фиксирует затраты на проект [59]. Информация, поступающая из этих двух источников, может оказаться противоречивой.

Осложнения из-за *изменений в запланированном графике* могут быть существенными. Задачи могут быть прерваны, и поэтому их придется разбить на части. Разбиение задачи вызывает трудности в вычислении стоимости, потому что затраты могут быть отнесены к началу задачи, ее концу или могут быть распределены пропорционально. Ресурсы могут стать недоступными из-за пробуксовок. При этом задачам могут быть заданы фиксированные даты начала и завершения, что очень плохо для перепланирования.

Осложнения же, вызываемые *системой бухгалтерского учета*, могут быть даже более хитрыми. Система бухгалтерского учета использует табели учета рабочего времени и счета для фиксации затрат на проекты. Людские ресурсы могут работать на нескольких проектах и на многих задачах в пределах одного и того же проекта. Пока система отсчета времени не синхронизирована с системой управления проектом и пока затраты не будут четко распределены, система бухгалтерского учета не будет иметь никаких сведений относительно того, сколько времени было потрачено на различные задачи. Счета поступают в бухгалтерию после того, как работа будет сделана и обычно по возможности оплачена.

Картина ясна. Сравнение **исходных данных** и **фактических** — реальная проблема. Кроме того, отслеживание времени и отслеживание стоимости могут давать противоречивую информацию. Проект может быть выполнен досрочно, при этом находиться в бюджете или с недорасходованным бюджетом. Он может также быть в пределах графика или превышать его, но превышать бюджет. Возможны и другие комбинации типа досрочного выполнения и с недорасходом бюджета или с превышением графика и с превышением бюджета.

4.4.1. Отслеживание графика

Изменения в запланированном графике должны отслеживаться в процессе создания проекта (это называется **отслеживанием графика**). Завершающее состояние каждой задачи должно быть зафиксировано и показано на гистограмме. То же самое применяется и в случае разбиения задач из-за прерываний, и при перераспределении ресурсов. Ход выполнения задачи можно скорректировать в абсолютных величинах, изменяя продолжительность выполнения или дату начала/завершения, или в относительных единицах, определяя процент завершения.

В предположении, что текущая дата проекта — `12 Apr 04` (12 апреля 2004 г.), на рис. 4.19 показано фактическое выполнение задач `Task1`, `Task2` и `Task3` в сравнении с планом на основе исходных данных, показанном на рис. 4.7. Узкая черная полоса внутри линии задачи указывает долю завершения этой задачи.

`Task1` закончена, как было намечено. Однако `Task2` началась одним днем позже и продолжалась на один день больше. В результате `Task3` могла начаться только в пятницу вместо среды. Это привело к риску пробуксовки проекта на два дня. Чтобы разрешить эту проблему, планировщик добавил к задаче ресурс Eric, что позволит закончить `Task3` согласно плану на основе исходных данных. После этого в один день (пятницу) `Task3` была помечена как завершенная на 50 процентов.

Эти объяснения относительно фактического состояния плана по сравнению с исходными данными могут быть проиллюстрированы с помощью **диаграммы Ганта, отображающей отслеживание графика** (рис. 4.20). Диаграмма Ганта, отображающая отслеживание графика, показывает пару полос для каждой задачи, одну над другой. Нижняя полоса представляет исходные

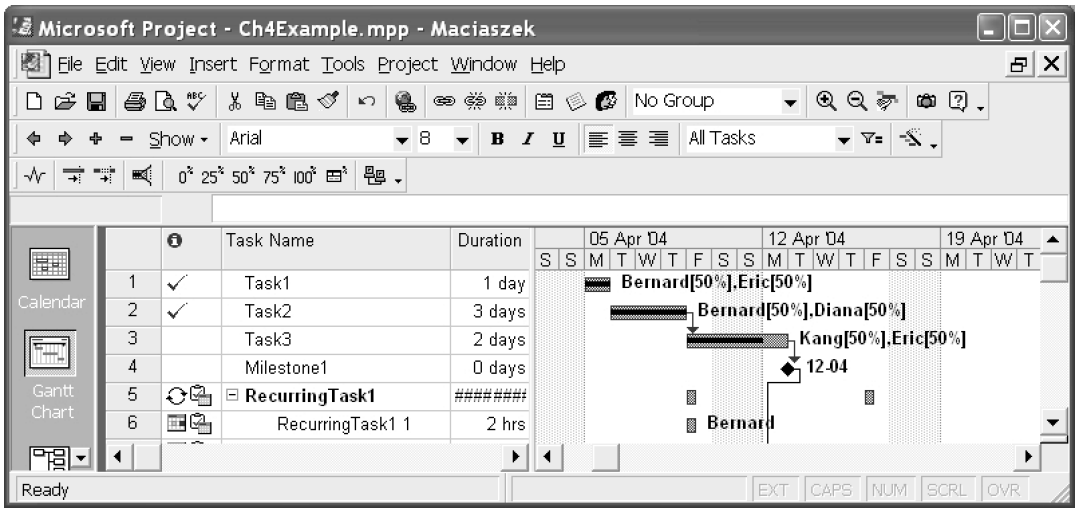


Рис. 4.19. Диаграмма Ганта, показывающая завершение задач
Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

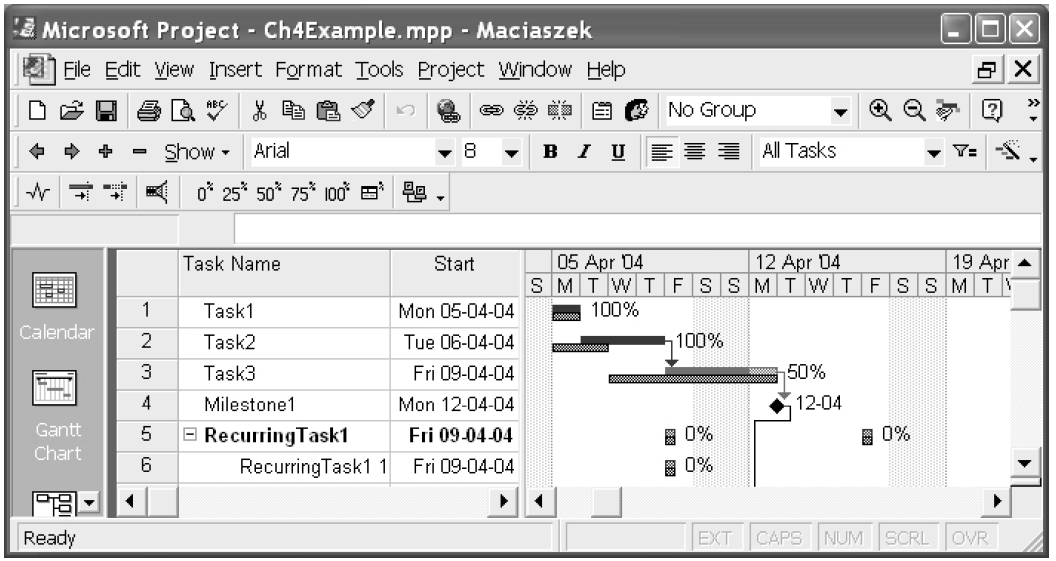


Рис. 4.20. Диаграмма Ганта, отображающая отслеживание графика
Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

данные задачи. Верхняя полоса представляет фактическое выполнение задач. Верхние полосы окрашены так, чтобы указать, закончены ли задачи, вовсе не начаты или частично выполнены.

Кроме графического представления диаграмма Ганта, отображающая отслеживание графика, представляет текстовую информацию для каждой задачи: Baseline Start (начало реализации исходных данных), Baseline Finish (завершение реализации исходных данных), Start Variance (отклонение начала) и Finish Variance (отклонение завершения). **Отклонение** (variance) представляет собой разницу между исходными данными и фактическим выполнением задачи.

Так как ресурсы распределены между задачами, отслеживание завершения задач дает также информацию относительно того, какую работу выполнил каждый ресурс. Эта информация может быть представлена в виде «от задачи к задаче», как показано на рис. 4.21. Вид, представленный на рис. 4.21, может также использоваться для корректировки величины фактической работы, выполненной ресурсом. Сравнивая запланированное количество работы с фактическим выполнением работы, можно вычислить работу, остающуюся для завершения задачи.

Чтобы можно было управлять людьми, важно отслеживать фактическую работу, выполненную каждым ресурсом. Рис. 4.22 демонстрирует соответствующее представление. Отклонение фактической работы от работы, запланированной в исходных данных, является индикатором того, как ресурс используется по сравнению с планом.

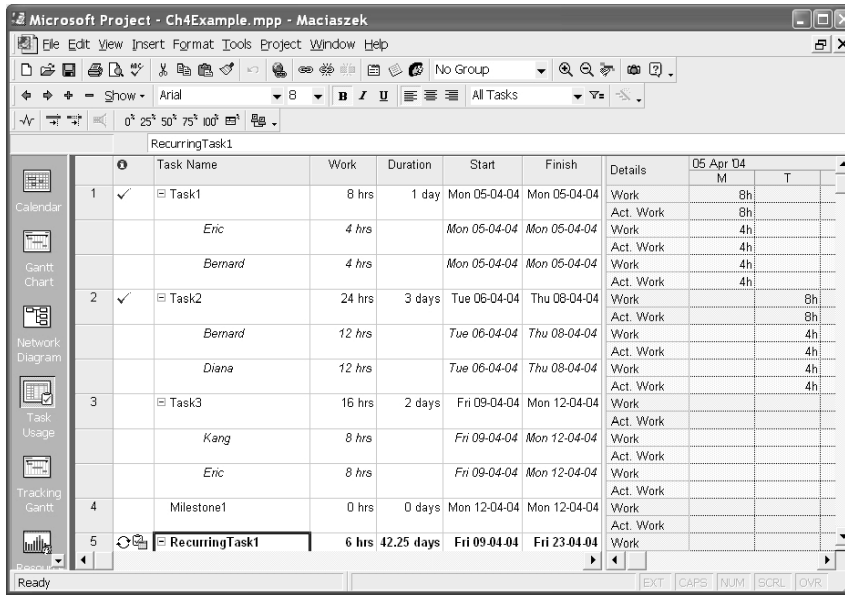


Рис. 4.21. Отслеживание работы «от задачи к задаче»
Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

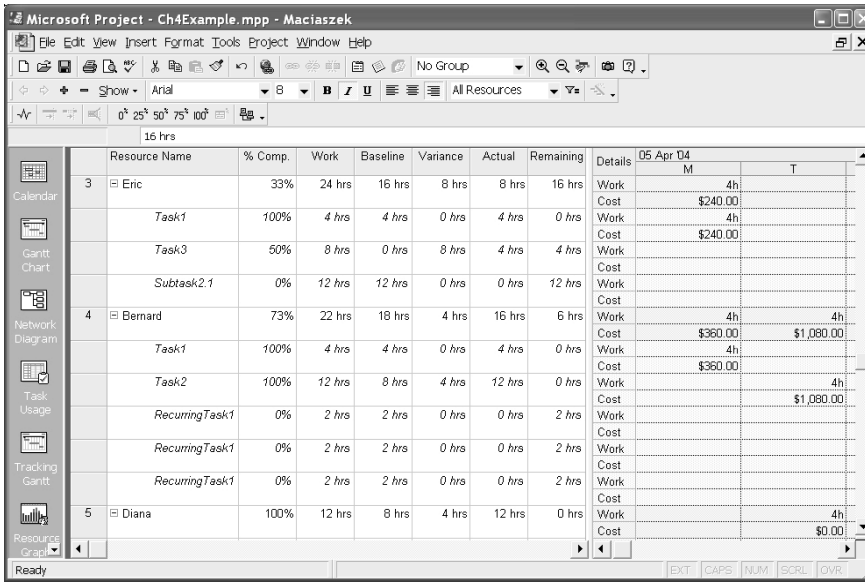


Рис. 4.22. Отслеживание работы «от ресурса к ресурсу»
Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

4.4.2. Отслеживание бюджета

Одним из путей **отслеживания бюджета** является рассмотрение его как прямое продолжение отслеживания графика. Это возможно, автоматически отслеживая фактические затраты законченных задач либо учитывая нормы ресурсов, методы периодической бухгалтерской отчетности и любые фиксированные расходы в процессе выполнения задачи. Этот подход может отражать любые изменения в плане, типа изменений в продолжительности выполнения задач или перераспределения ресурсов.

Однако, как упоминалось ранее, фактические затраты, предоставляемые *системой бухгалтерского учета*, могут отличаться от затрат, вычисленных по фактически выполненной работе. В некоторых случаях система бухгалтерского учета может быть единственным источником информации о фактической стоимости. Это может произойти, например, когда компромисс между сметной стоимостью и трудозатратами бюджета (рис. 4.12) не гарантирует уровень детализации плана, который обеспечил бы точный расчет бюджета.

Отслеживание бюджета необходимо для принятия оперативных и тактических решений. Соответственно, результаты отслеживания бюджета должны быть суммированы и представлены в форме, подходящей для менеджеров. Один из наиболее полезных методов, который кратко информирует, находится ли проект в рамках бюджета, называется *анализом выполненной стоимости*.

Фактические затраты, полученные из графика выполнения

Инструментальное средство управления проектами может вычислять полные **фактические затраты** задач, которые использовали ресурсы. Рис. 4.23 показывает такое вычисление для задач, начатых до понедельника 12 апреля 2004 г. Вычисления выполнены путем умножения почасовых ставок ресурса (рис. 4.13) на часы работы. Фактическое выполнение задач `Task2` и `Task3`, показанное на рис. 4.23, отличается от плана, построенного на основе исходных данных для этих задач (рис. 4.7). Рис. 4.23 демонстрирует вычисления. Обратите внимание, что задача `Task3` до `12 Apr 04` (12 апреля 2004 г.) выполнена только на 50 процентов (рис. 4.22), но вычисление выполнено в предположении ее 100-процентного завершения и в плане, построенном на основе исходных данных, и в фактическом графике.

Знание исходных данных, фактических затрат и остаточной стоимости для отдельных задач позволяет вычислить текущую стоимость проекта. **Текущие затраты** — это текущая запланированная стоимость или общая стоимость проекта после рассмотрения выполнения проекта и любой связанной с ним корректировки стоимости. Текущие затраты представляют собой фактические затраты на рассматриваемую дату плюс затраты на реализацию исходных данных для всех остающихся задач. **Остаточная стоимость** представляет собой разность между текущими и фактическими затратами.

Рис. 4.24 представляет затраты для всего проекта. Цифры показывают текущие затраты, затраты в соответствии с исходными данными, фактические затраты и остаточную стоимость проекта. Величины отражают предыдущее обсуждение.

Task2	Исходные данные 2 дня	Факт 3 дня
Bernard 50%	8h * $90 = $720	12h * $90 = $1080
Diana 50%	8h * $100 = $800	12h * $100 = $1200
Сумма	$1520	$2280

Task3	Исходные данные 4 дня	Факт 2 дня
Kang 50%	16h * $80 = $1,280	8h * $80 = $640
Eric 50%		8h * $60 = $480
Сумма	$1,280	$1,120

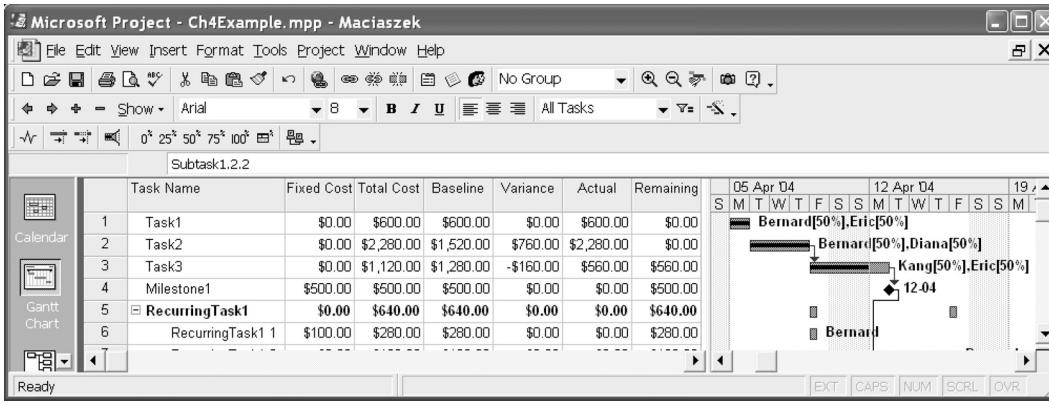


Рис. 4.23. Подсчет затрат на исходные данные и фактических затрат
Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

Фактические затраты, полученные из бухгалтерского учета

Подсчет фактических затрат на основе графика выполнения редко дает истинный и окончательный расчет совершенных затрат. Во-первых, графики вряд ли будут включать все ресурсы, используемые задачами, или даже

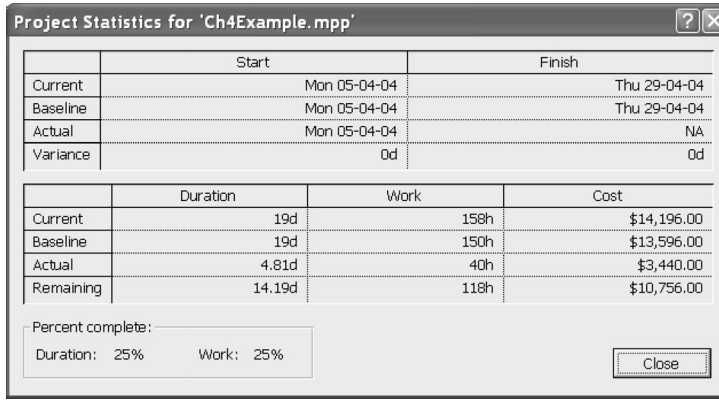


Рис. 4.24. Затраты проекта и другая статистика

включать все задачи. Во-вторых, неповоротливость системы бухгалтерского учета может исказить реальную историю. Во всех случаях, когда действительность такова, что не соответствует цифрам бюджета, бюджет должен быть изменен вручную. С технической точки зрения это требует отказа от автоматического обновления фактических затрат в инструментальном средстве управления проектами, а также отказа от учета фактических затрат задач и ресурсов.

В некоторых случаях фактическая стоимость, полученная из системы бухгалтерского учета, может быть добавлена к *фиксированным затратам* задачи, которую не удается оценить во время планирования (вспомним, например, столбец Fixed Cost (фиксированные затраты) на рис. 4.23). В других случаях фактическая стоимость может быть добавлена к *затратам за использование* рабочих ресурсов в дополнение к обычным *платам, основанным на ставках* (столбец Cost/Use — стоимость/использование — на рис. 4.13). В случаях, подобных этому, модификации в плане исходных данных могут удалить выявленные нарушения.

Обсуждение в этой главе концентрируется на людских ресурсах. Часто несоответствия между ассигнованными и фактическими затратами, сообщенными системой бухгалтерского учета, возникают не из-за этих ресурсов. Все несоответствия между ассигнованными и фактическими затратами, будь то из-за людских или других ресурсов, должны быть исправлены в бюджете (чтобы соответствовать фактическим затратам). В случае людских ресурсов несоответствия можно исправить, делая изменения в плане. Чтобы проиллюстрировать последствия ручной модификации фактических затрат, рис. 4.25 показывает модификации фактических затрат двух (оконтуренных) ресурсов, после того как система бухгалтерского учета выдаст данные по затратам.

Изменения на рис. 4.25 составляют увеличение в 400 долларов у Bernard в `Task1` и увеличение в 180 долларов у Kang в `Task3`. Эти увеличения автоматически отражены в новом бюджете (рис. 4.26) и в статистике стоимости проекта (рис. 4.27).

Выполненная стоимость

Анализ выполненной стоимости — количественная технология, определяющая, находится ли проект в рамках бюджета [81, 59]. Технология использует оценку исходных данных плана и фактического выполнения на текущий момент, чтобы определить состояние завершенности («здоровье» завершенности) проекта. Анализ проводится как для отдельных задач, так и для всего проекта в целом. Он может быть также доведен до уровня ресурсов в пределах каждой задачи. Анализ выполненной стоимости также известен как *управление реализацией* и *управление показателями*.

Анализ выполненной стоимости может быть сделан только при условии, что ресурсы (их стоимостные показатели) распределены по задачам. С количественной точки зрения задача, не обладающая ресурсами, не может иметь никакой выполненной работы и не может иметь затрат или *выполненной стоимости*.

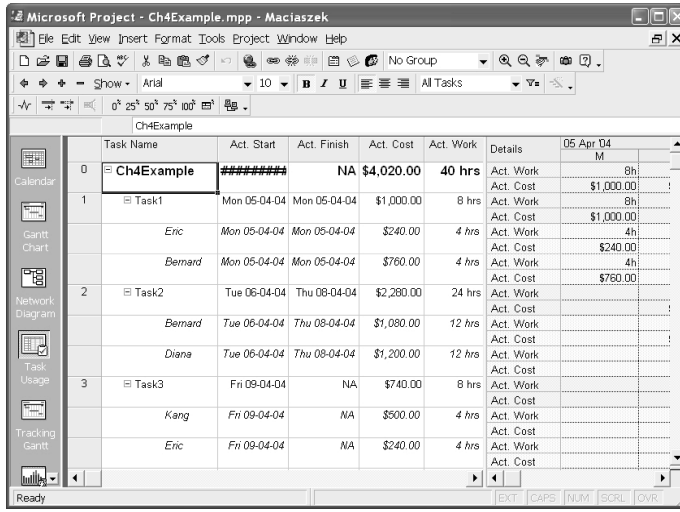


Рис. 4.25. Изменения бюджета, вызванные системой бухгалтерского учета
Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

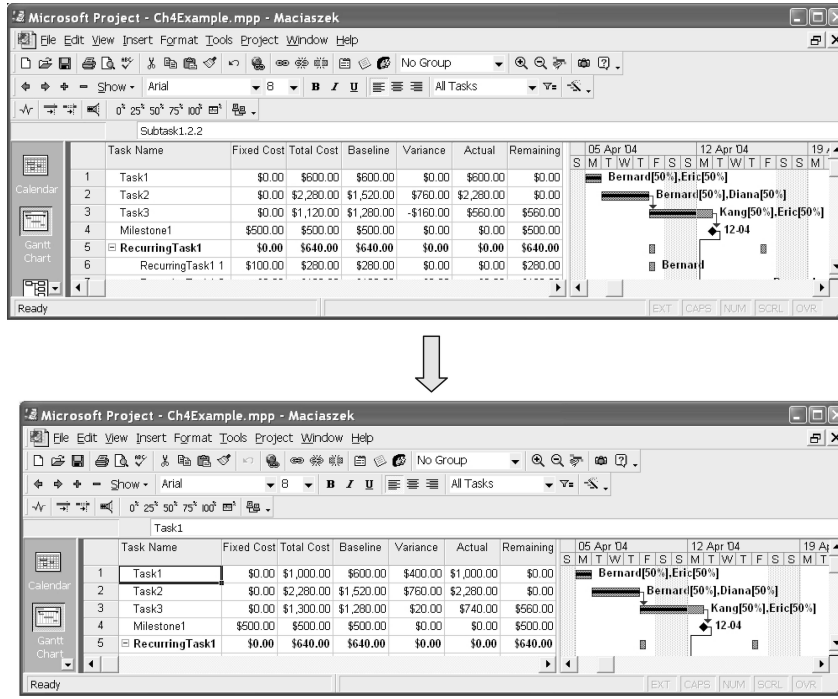
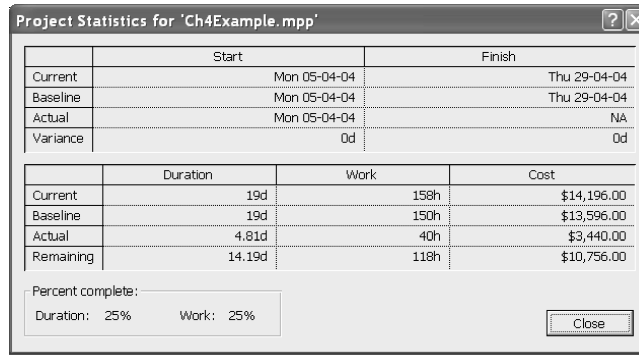


Рис. 4.26. Бюджет до и после оценки фактических затрат, полученной из бухгалтерского учета
Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation



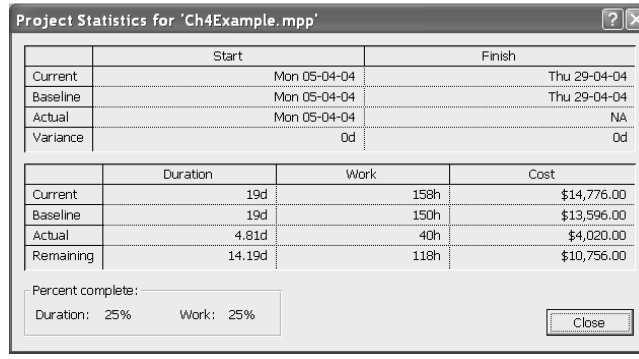
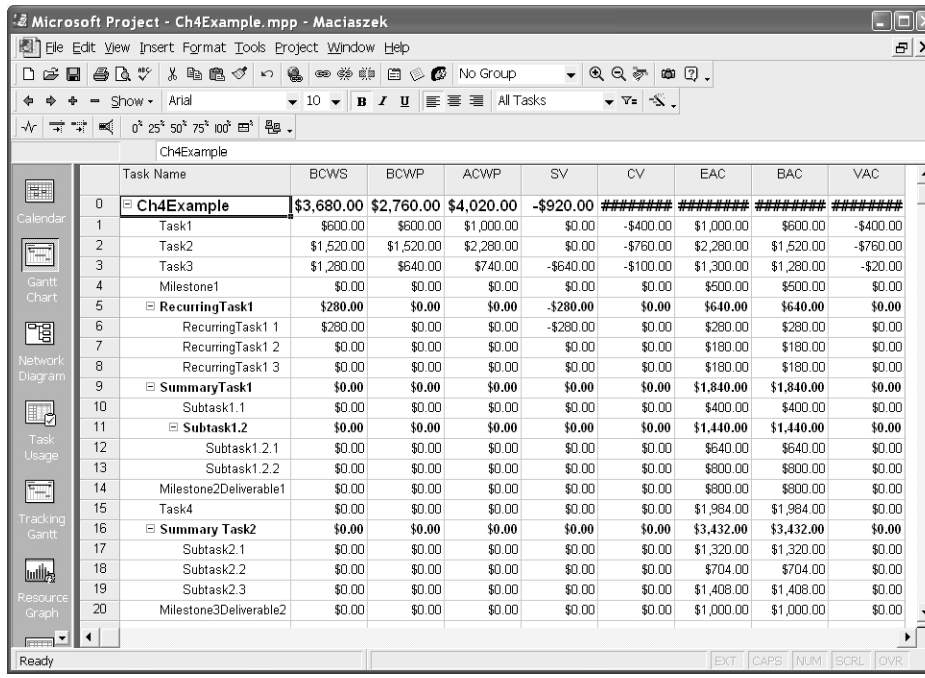


Рис. 4.27. Статистика стоимости проекта до и после оценки фактических затрат, полученной из бухгалтерского учета

Как показано на рис. 4.28, анализ выполненной стоимости обеспечивает разнообразие дополнительных критериев качества работы. Здесь используются показатели:

- BCWS — Budgeted Cost of Work Scheduled — бюджетная стоимость запланированных работ;
- BCWP — Budgeted Cost of Work Performed — бюджетная стоимость выполненных работ;
- ACWP — Actual Cost of Work Performed — фактическая стоимость выполненных работ;
- SV — Scheduled Variance — отклонение по срокам (SV = BCWP – BCWS);
- CV — Cost Variance — отклонение по стоимости (CV = BCWP – ACWP);
- EAC — Estimate at Completion — оценка по завершению;
- BAC — Budget at Completion — бюджет по завершению;
- VAC — Variance at Completion — отклонение по завершению (VAC = BAC – EAC).



	Task Name	BCWS	BCWP	ACWP	SV	CV	EAC	BAC	VAC
0	**Ch4Example**	**$3,680.00**	**$2,760.00**	**$4,020.00**	**-$920.00**	########	########	########	########
1	Task1	$600.00	$600.00	$1,000.00	$0.00	-$400.00	$1,000.00	$600.00	-$400.00
2	Task2	$1,520.00	$1,520.00	$2,280.00	$0.00	-$760.00	$2,280.00	$1,520.00	-$760.00
3	Task3	$1,280.00	$640.00	$740.00	-$640.00	-$100.00	$1,300.00	$1,280.00	-$20.00
4	Milestone1	$0.00	$0.00	$0.00	$0.00	$0.00	$500.00	$500.00	$0.00
5	**RecurringTask1**	**$280.00**	**$0.00**	**$0.00**	**-$280.00**	**$0.00**	**$640.00**	**$640.00**	**$0.00**
6	RecurringTask1 1	$280.00	$0.00	$0.00	-$280.00	$0.00	$280.00	$280.00	$0.00
7	RecurringTask1 2	$0.00	$0.00	$0.00	$0.00	$0.00	$180.00	$180.00	$0.00
8	RecurringTask1 3	$0.00	$0.00	$0.00	$0.00	$0.00	$180.00	$180.00	$0.00
9	**SummaryTask1**	**$0.00**	**$0.00**	**$0.00**	**$0.00**	**$0.00**	**$1,840.00**	**$1,840.00**	**$0.00**
10	Subtask1.1	$0.00	$0.00	$0.00	$0.00	$0.00	$400.00	$400.00	$0.00
11	**Subtask1.2**	**$0.00**	**$0.00**	**$0.00**	**$0.00**	**$0.00**	**$1,440.00**	**$1,440.00**	**$0.00**
12	Subtask1.2.1	$0.00	$0.00	$0.00	$0.00	$0.00	$640.00	$640.00	$0.00
13	Subtask1.2.2	$0.00	$0.00	$0.00	$0.00	$0.00	$800.00	$800.00	$0.00
14	Milestone2Deliverable1	$0.00	$0.00	$0.00	$0.00	$0.00	$800.00	$800.00	$0.00
15	Task4	$0.00	$0.00	$0.00	$0.00	$0.00	$1,984.00	$1,984.00	$0.00
16	**Summary Task2**	**$0.00**	**$0.00**	**$0.00**	**$0.00**	**$0.00**	**$3,432.00**	**$3,432.00**	**$0.00**
17	Subtask2.1	$0.00	$0.00	$0.00	$0.00	$0.00	$1,320.00	$1,320.00	$0.00
18	Subtask2.2	$0.00	$0.00	$0.00	$0.00	$0.00	$704.00	$704.00	$0.00
19	Subtask2.3	$0.00	$0.00	$0.00	$0.00	$0.00	$1,408.00	$1,408.00	$0.00
20	Milestone3Deliverable2	$0.00	$0.00	$0.00	$0.00	$0.00	$1,000.00	$1,000.00	$0.00

Рис. 4.28. Анализ выполненной стоимости
Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

BCWS — выполненная стоимость задачи (проекта, ресурса), основанная на запланированной работе. В просторечии BCWS известна как **намеченная работа**. BCWS рассчитывается для задач, которые должны были использовать ресурсы и быть хотя бы частично выполнены на дату оценки выполнения проекта (то есть, в рассматриваемом примере на `12 Apr 04` — 12 апреля 2004 г.). Задачи, которые еще не начались, имеют BCWS, равную нулю.

BCWP — выполненная стоимость задачи (проекта, ресурса), основанная на проценте фактически сделанной работы от запланированной на дату, когда оценивается выполнение проекта. Она также известна как **выполненная работа**. Например, на рис. 4.28 BCWS для `Task3` — 1280 долларов. Эта задача к `12 Apr 04` — 12 апреля 2004 г. — выполнена на 50 процентов. Следовательно, ее BCWP — 640 долларов.

ACWP — сумма всех фактических затрат к рассматриваемой дате. Она включает вычисляемые по нормам и фиксированные расходы, полученные из плана, и любые вручную введенные затраты, полученные из системы бухгалтерского учета. Например, ACWP — 1000 долларов для `Task1` и 740 долларов для `Task2`, соответствующие изменениям, полученным от бухгалтерского учета (рис. 4.25). ACWP для `Task3` — 2280 долларов из-за изменений, связанных с распределением дополнительного ресурса для этой задачи (рис. 4.23).

SV — разность между BCWP и BCWS. Запланированное отклонение, равное нулю, указывает, что задача должна быть завершена (кроме, конечно, случая, когда и BCWP, и BCWS равны нулю, означающего, что задачи не начались). В большинстве других случаев SV отрицательно и отражает (в абсолютных денежных единицах) факт, что работа не была завершена.

CV — разность между BCWP и ACWP. Это признак фактического сбережения или превышения стоимости по сравнению с бюджетом на основе исходных данных. В рассматриваемом примере для `Task1`, `Task2` и `Task3` расходы превышены. На работу, выполненную для этих задач, было потрачено большее количество денег, чем было намечено.

EAC — оцененная стоимость задачи по завершению проекта, основанная на текущем выполнении. Для законченных задач EAC равна ACWP. Для задач, которые все еще не начались, EAC равна намеченным затратам на основе исходных данных (рис. 4.16). Для задач, которые начались, но не закончены, EAC — оцененная общая стоимость, включая изменения, поступившие от системы бухгалтерского учета. EAC для `Task3` — 1300 долларов, как показано на рис. 4.26.

BAC — бюджетные затраты задачи по завершению, полученные на основе текущего выполнения проекта. Для законченных задач BAC равны BCWP. Для задач, которые все еще не начались, BAC равны намеченным затратам в исходных данных (рис. 4.16). Для задач, которые начались, но не закончены, BAC представляют фактическую стоимость, полученную из графика выполнения. BAC для `Task3` — 1280 долларов, как показано на рис. 4.23.

VAC является просто разностью между BAC и EAC. Для законченных задач VAC равна CV. Она равна нулю для задач, которые еще не начались. Для задач, частично законченных, VAC указывает превышение (отрицательная величина) или недорасход (положительная величина) бюджета, когда рассматриваются фактические затраты, полученные от системы бухгалтерского учета.

Анализ выполненной стоимости — обязательное инструментальное средство для руководителей проектов и менеджеров. Оно позволяет выполнять *оперативное управление* выполнением проекта, выявлять на ранних стадиях тенденции произведенных затрат и производить корректирующие действия, чтобы избежать превышения основного бюджета.

Резюме

1. *Планирование и отслеживание проекта ПО — непрерывная* операция оценки того, сколько времени, денег, трудозатрат и ресурсов должно быть израсходовано на проект. Это не отдельная стадия, а всеобъемлющая операция, охватывающая стадии жизненного цикла.
2. *Подлежащие сдаче продукты* проекта — это программные продукты или сервисы, которые удовлетворяют проектным показателям. Как только подлежащие сдаче продукты будут известны, в работе можно выбрать *контрольные точки (вехи)*, *стадии* и *задачи*. Работа нуждается в *ресурсах*, то есть, людях, оборудовании, материалах, поставках, программном и аппаратном обеспечении и т. д., требуемых проектными задачами.

3. *Планирование* проекта представляет собой оценку времени, необходимого для разработки, и распределение ресурсов между задачами. *Составление бюджета* для проекта включает оценку затрат, требуемых ресурсами и необходимых для задач.
4. *Планирование* проекта — это подвижная цель. Имеются сложные взаимосвязи между задачами, различные ограничения на сроки их выполнения и разнообразие возможных прерываний в работе. Существуют *зависимости между задачами* из-за способа организации их работы и из-за конкуренции в использовании одних и тех же ресурсов. *Негибкие ограничения* устанавливают конкретные даты начала или конца задач.
5. Грубо *ресурсы* делятся на рабочие и материальные. *Рабочие ресурсы* состоят из людей и оборудования, включая аппаратное и программное обеспечение. *Материальные ресурсы* — это расходные материалы и поставки. Назначение ресурсов включает и гибкость, и ограничения планирования.
6. Подход к планированию, когда продолжительность выполнения задач может изменяться (как в одну, так и в другую сторону) добавлением или удалением ресурсов, известен как *планирование, определяемое трудозатратами*. Предоставление *использования ресурсов* включает часы запланированной работы каждого ресурса и часы, распределенные между задачами. Анализ использования ресурса позволяет определить неполное и избыточное распределение ресурсов.
7. Методы *оценки бюджета* имеют диапазон от упрощенных методов типа экспертной оценки и оценки по аналогии до более строгих методов типа оценок на основе графика выполнения и алгоритмических оценок.
8. *Оценка бюджета на основе графика выполнения* предполагает, что существует план *исходных данных* с ресурсами, распределенными по задачам. Оценка требует ввода *ставок оплаты* и других затрат для работы, а также материальных ресурсов. Она требует также определения методов периодической бухгалтерской отчетности для ресурсов.
9. *Алгоритмическая оценка бюджета* использует некоторую меру (*метрику*) размера задачи и применяет ее к алгоритму, полученному опытным путем, чтобы получить бюджет для решения задачи. Бюджет оценивает стоимость людских ресурсов (*трудозатрат*) как функцию размера проекта. *Размер* обычно выражается в строках кода (lines of code — LOC), в функциональных единицах (function points — FP) или в объектных единицах (object points — OP).
10. COCOMO 81 и COCOMO II — наиболее известные алгоритмические модели. Обе модели признают относительную важность различных *коэффициентов затрат* в проектной стоимости и трудозатратах. Имеются три модели COCOMO 81: основной, промежуточный и продвинутый. Существуют также три модели COCOMO II: компоновки приложений, раннего проектирования и пост-структурная.
11. *Отслеживание выполнения проекта* включает два основных множества сравнений: 1) план на основе исходных данных по сравнению с фактическим планом и 2) бюджет на основе исходных данных по сравнению с фактическим бюджетом. Отслеживание рассматривает изменения в запла-

нированном графике и фактических затратах, поступающих от системы бухгалтерского учета.

12. Диаграмма Ганта может использоваться в *отслеживании графика выполнения*, чтобы представить фактическое состояние плана по сравнению с исходными данными.
13. *Отслеживание бюджета* — прямое продолжение отслеживания графика выполнения. Оно отслеживает фактические затраты законченных или выполняющихся задач, используя нормы ресурса, методы периодической бухгалтерской отчетности и любые фиксированные расходы.
14. *Анализ выполненной стоимости* — количественная технология определения, находится ли проект в рамках бюджета. Технология использует оценку исходных данных плана и фактического выполнения на дату, для которой определяется состояние законченности проекта.

Ключевые термины

Обзорные вопросы

1. Какова типичная последовательность шагов в планировании проекта?
2. Что такое критический путь? Изобразите простой пример CPM-графика сети, чтобы показать важность критического пути для планирования и отслеживания проекта.
3. Объясните различия между резервным временем, временем задержки и временем опережения.
4. Объясните суть принципов планирования «Как можно скорее» и «Как можно позже».
5. Объясните наблюдение, что имеется точка, в которой увеличение бюджетных ассигнований приводит к уменьшению фактической сметной стоимости.
6. Каковы методы периодической бухгалтерской отчетности? Как они затрагивают вычисления затрат? Приведите простой пример.
7. Объясните три «калибровочные» опции оценки бюджета: строки кода, функциональные единицы и объектные единицы. Как они используются? Как они соотносятся друг с другом?
8. Как различные коэффициенты затрат влияют на оценку бюджета? Чем они отличаются? Имеются ли какие-то интересные уроки относительно разработки ПО, в основном касающиеся изучения коэффициентов затрат?
9. Каково различие между коэффициентами затрат и масштабирующими коэффициентами в COCOMO II?
10. Объясните различия между исходными данными, фактическими данными, отклонениями и остаточными стоимостями.
11. Что такое выполненная стоимость? Объясните концепции запланированной и выполненной работы.

Примеры задач

1. Обратимся к диаграмме Ганта на рис. 4.29. Рассмотрим следующие два изменения в календаре проекта. Дни 8, 9 и 12 апреля 2004 г. являются нерабочими из-за пасхальных праздников. Пусть также Bernard работает только половину дня (4 часа) каждый понедельник. Сделайте необходимые изменения в диаграмме Ганта на рис. 4.29. Какова новая дата поставки? Имеются ли изменения в полной продолжительности выполнения проекта?

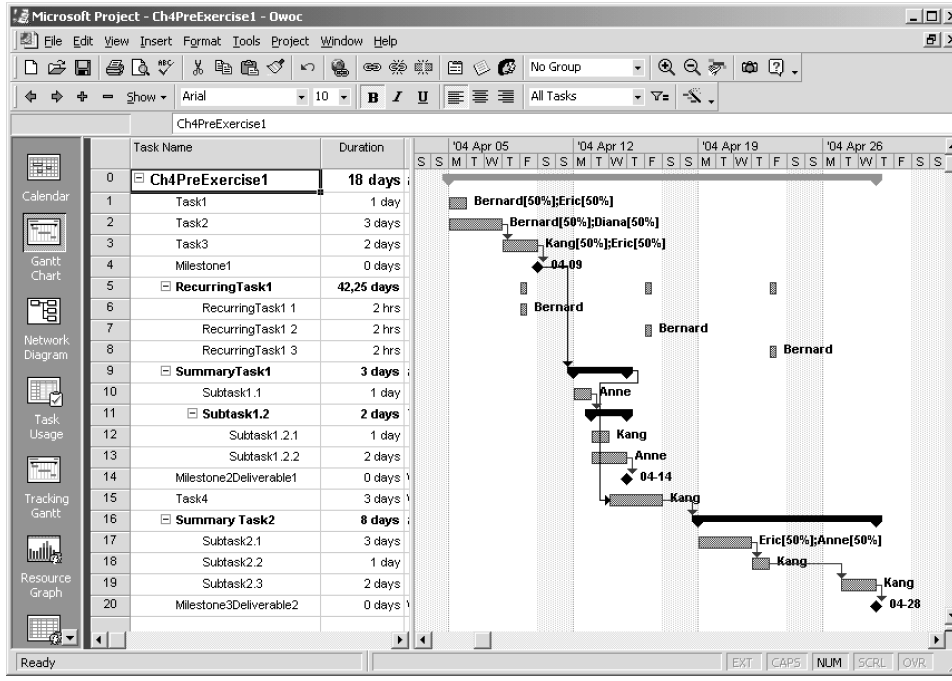


Рис. 4.29

Источник: образ экрана Microsoft® Office Project (2003), перепечатано с разрешения Microsoft Corporation, Copyright © 1998–2003 Microsoft Corporation

2. Снова обратимся к диаграмме Ганта на рис. 4.29 (см. выше вопрос 1). Рассмотрим следующие три изменения: 1) `Task1` связана с `Subtask1.2.1`, 2) `Task1` запланирована по принципу «Как можно позже» и 3) для `Subtask1.2.1` установлено ограничение, что она должна быть закончена к `20 Apr 04` (20 апреля 2004 г.). Сделайте необходимые изменения в диаграмме Ганта на рис. 4.29. Какова будет новая дата поставки?

3. Рассмотрим пост-структурную модель COCOMO II и следующую формулу оценки трудозатрат:

$$effort = c * size^{k} * m + autoeffort$$

Предположим, что величина постоянного коэффициента `c` — 2.5, предполагаемая величина `size` — 100 KLOC и что автотрудозатраты составляют 35 человеко-месяцев. Рассмотрим далее следующие величины коэффициентов затрат и масштабирующих коэффициентов в двух проектах ПО — Проект A и Проект B (таблицы 4.4 и 4.5). Вычислите величины показателя `k` и множителя `m` для двух проектов. Используйте эти величины, чтобы оценить трудозатраты, требуемые для этих двух проектов.

Таблица 4.4. Коэффициенты затрат для Проектов A и B

Код коэффициента затрат	Название коэффициента затрат	Проект A	Проект B
RELY	Требуемая надежность системы	1.23 (высокая)	0.73
CPLX	Сложность модулей системы	1.31 (очень высокая)	0.80
DOCU	Объем требуемой документации	Не имеется	0.76
DATA	Размер используемой БД	1.15 (умеренный)	0.87
RUSE	Требуемый процент повторно используемых компонентов	1.00 (нейтральный)	0.94
TIME	Ограничения времени выполнения	1.30 (высокие)	Не имеется
PVOL	Изменчивость платформы разработки	Не имеется	0.84
STOR	Ограничения на память	Не имеется	Не имеется
ACAP	Способность анализировать проект	Не имеется	Не имеется
PCON	Преемственность персонала	1.18 (довольно низкая)	Не имеется
PEXP	Опыт программиста в предметной области проекта	1.25 (низкий)	0.77
PCAP	Способности программиста	1.24 (низкие)	0.79
AEXP	Опыт аналитика в предметной области проекта	1.33 (очень низкий)	Не имеется
LTEX	Опыт в языках программирования и инструментальных средствах	1.15 (довольно низкий)	0.90
TOOL	Использование инструментальных средств разработки ПО	1.21 (низкое)	Не имеется
SCED	Сжатие плана разработки	Не имеется	Не имеется
SITE	Объем работы с несколькими файлами и качество связей сайтов	1.05 (неплохой)	Не имеется

Таблица 4.5. Масштабирующие коэффициенты для Проектов A и B

Масштабирующий коэффициент	Проект A	Проект B
Наличие прецедентов	5 (очень низкое)	1
Гибкость разработки	4 (низкая)	2
Определение структуры/рисков	3 (относительно низкое)	1
Единство коллектива	2 (неплохое)	0
Компетентность	5 (очень низкая)	1

Глава 5

Управление процессом создания и отслеживания программного обеспечения

Планирование и отслеживание ПО, рассмотренные в предыдущей главе, составляют важную и необходимую часть управления процессом создания и эксплуатации ПО. Планирование и отслеживание проекта задают правила работы инженеров ПО. Однако чтобы говорить об управлении как таковом, необходимо управлять и другими операциями. Менеджеры проектов должны также управлять людьми, рисками, качеством, изменениями и т. д. Управление процессом применяется к нескольким проектам и его цель — полное совершенствование производственного процесса в организации.

Организации, разрабатывающие ПО, постоянно стремятся улучшать методику и практику своих разработок. Это непрерывное усовершенствование процесса требует предварительного определения уровня **зрелости процесса**. Требуется также, чтобы в организации знали об *основных факторах*, к которым приведет усовершенствование методики и практики процесса в этой организации. Концепции зрелости процесса и основных факторов в усовершенствовании процесса подкрепляет **модель зрелости возможностей** (Capability Maturity Model — CMM), разработанная Институтом программной инженерии (Software Engineering Institute — SEI) в Университете Карнеги в Питтсбурге (Carnegie Mellon University in Pittsburg, USA) [22].

Как видно на рис. 5.1, CMM-модель определяет пять возрастающих уровней зрелости процесса: 1) начальный (initial), 2) повторяемости (repeatable), 3) регламентируемости (defined), 4) управляемости (managed) и 5) оптимизируемости (optimizing) [76, 78]. CMM стала эффективным стандартом качества процесса. Модель имеет связанный с ней механизм аккредитации, который позволяет организациям подвергнуться процедуре ревизии, которая засвидетельствует зрелость их процесса. Многие контракты на разработку ПО требуют доказательства конкретного минимального уровня зрелости процесса от поставщиков ПО и подрядчиков.

На *начальном уровне* зрелости организация на самом деле не имеет никакого предсказуемого процесса. Любой успешный проект — вопрос удачи, а не результат управляемого процесса. Очевидно, результат хорошей работы определяется некоторыми основными индивидуумами. Если эти индивидуумы оставляют организацию или когда возникает некоторая другая кризисная ситуация, нет никаких установленных порядков, которым нужно следовать, чтобы вернуть проект в нужное русло.

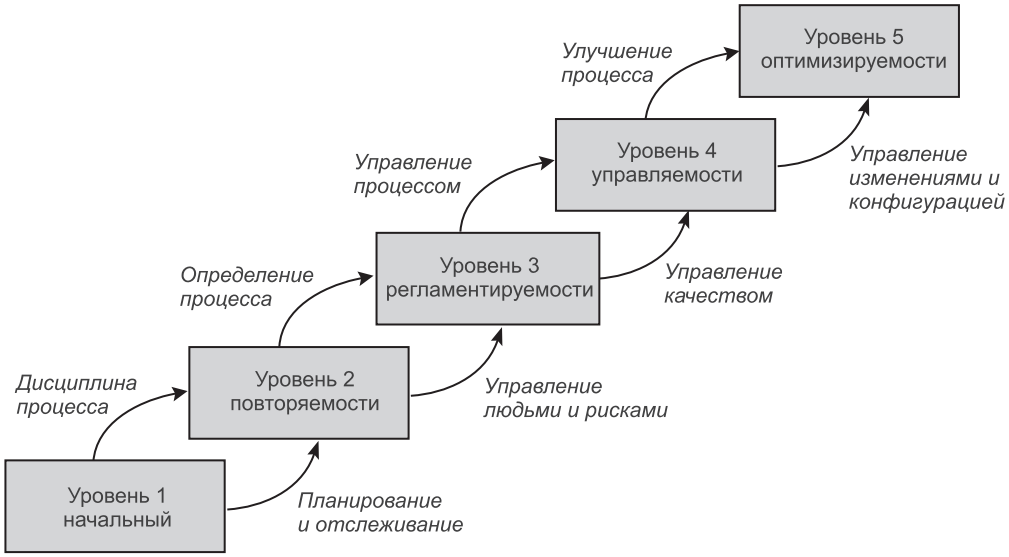


Рис 5.1. CMM-уровни

Переход на *уровень повторяемости* требует усовершенствования дисциплины процесса. Дисциплина — результат планирования и отслеживания операций, чему до некоторой степени способствуют гарантии качества и управление конфигурацией. Процесс на этом уровне все еще более интуитивен, нежели запланирован, но опыт планирования и отслеживания предыдущих проектов может быть успешно использован (повторен) в текущем проекте.

Переход на *уровень регламентируемости* требует улучшения в определении процесса. Основные факторы в достижении уровня регламентируемости — управление людскими ресурсами и рисками. Определяющие процессы занимаются рисками и кризисными ситуациями. Эти процессы облегчаются широким использованием инструментальных средств разработки ПО.

Переход на *уровень управляемости* требует улучшения управления производственным процессом. На этом уровне тщательно отбираются процессы и метрики продуктов, чтобы измерить качество ПО и производительность людей. Метрики позволяют на ранних этапах определить проблемы и предпринять обходные или корректирующие действия.

Переход на *уровень оптимизируемости* требует улучшения самой области совершенствования процесса. Основной фактор в совершенствовании процесса — управление изменениями и конфигурацией, включая управление дефектами и повышением темпов роста, а также непосредственными изменениями в технологии и процессе. Должна быть запланирована и субсидирована целая область усовершенствований процесса.

Организация этой главы находится в соответствии с CMM-моделью и основными факторами для перехода между уровнями. Основной фактор перехода с уровня 1 к уровню 2 (планирование и отслеживание) был рассмотрен в главе 4. Остальным основным факторам посвящены главные разделы этой главы.

5.1. Управление людьми

Управление людьми — это не то, что обычно предполагается под этим термином. Связано это с рядом причин:

- Технология изменяет все аспекты бизнеса.
- Изменение бизнес-среды влияет на организационные структуры, приводя к формированию сетевых организаций, нежестко связанных сотрудничающих единиц.
- Новые организационные структуры подразумевают новые стили управления.
- Новые стили управления должны быть допускающими риски и творческими.
- Допущение риска и творчество не сочетаются с бюрократическим управлением людьми, основанным на ответственности и методиках.
- Сильной стороной современных рабочих коллективов является гибкое сотрудничество людей, где отдельные индивидуумы не обязательно являются лояльными к организации, но полагаются на рабочую среду и ряд абстрактных, не связанных с работой потребностей.

Современные организации имеют тенденцию выглядеть как *сетевые структуры* нежестко связанных единиц, объединенных скорее коллективной общностью, нежели собственностью. Люди в **сетевой организации** обладают уникальными представлениями о мотивации, лояльности, организационном воздействии, лидерстве, достижениях, удовлетворении, вознаграждении и т. д. [7]. Старый стиль иерархического управления не применим для коллективов в сетевой организации. Коллективы разработчиков ПО находятся на переднем крае этой волны изменений.

5.1.1. Привлечение и мотивация людей

Привлечение и мотивация людей является сущностью процесса создания коллектива. Коллектив формируется на основе плана проекта (глава 4) и связанного с этим **плана управления комплектацией персонала**. Реализация формирования коллектива включает подбор менеджеров проектов, руководителей проектов и всех других членов коллектива. В зависимости от ситуации членами коллектива могут быть текущие служащие, назначенные на проект, или может потребоваться сформировать коллектив, нанимая новых работников или привлекая внешних подрядчиков и консультантов.

Современные сетевые организации одобряют **самоуправляемые коллективы**, но хорошее управление и лидерство — необходимые условия для успеха любого проекта. Чем больше проект, тем это утверждение более вер-

но. Подбор хорошего менеджера проекта, который знает, как и когда управлять, как и когда руководить, является критическим решением владельца проекта.

Хелдман [37] задает следующий набор различных характеристик **руководителей проектов** и **менеджеров**:

Руководители имеют ловкость убедить других делать то, что нужно, организовать и сплотить их вокруг своего видения предмета. Хорошие руководители подбирают членов коллектива, которые верят в их видение. Руководители задают направление и временные границы и имеют способность привлечь хорошие таланты работать на них. ... Менеджеры обычно ориентированы на задачи, связанные с понятиями типа планов, управления, бюджетов, стратегий и методик. Они — универсалы с широкой базой планирования и организационных навыков, и их первичная цель — удовлетворять потребности участников разработки. Они также обладают мотивационными навыками и способностью распознавать и вознаграждать поведение.

Важный момент — это то, что менеджер проекта должен также быть и руководителем, в то время как обратное не обязательно. Хороший менеджер должен и управлять, и вдохновлять. Руководство особенно эффективно на больших проектах. В таких проектах расстояние между менеджерами и исполнителями решений может включать несколько уровней управления. Чтобы выполнить то, что надо, менеджеры должны доверять членам коллектива, а члены коллектива должны быть как вдохновлены, так и заинтересованы.

Формирование коллектива

Члены коллектива привлекаются путем назначения служащих на проект и/или найма новых работников. *Перевод служащих* между проектами является прерогативой высшего руководства. Менеджер проекта должен пытаться найти подходящих, потенциально доступных и заинтересованных служащих и запросить их перевод. Подходящие не обязательно означает лучшие служащие [59]. Лучшие люди находятся в большом спросе. Переназначение их означает, что пострадают другие проекты. Наличие лучших людей повышает надежды на удачное управление проектом. Наконец, лучшие люди могут и не вписаться в коллектив и могут вызвать нежелательные напряженные отношения.

Наем новых служащих дает лучшую возможность нахождения людей, которые могут обеспечить определенные потребности проекта и коллектива, но это всегда связано с большим риском неправильной оценки претендентов. Решение, кого нанимать, принимается с учетом трех видов входной информации [96]:

- информация, представляемая претендентами, об их портфеле занятости;
- рекомендации, полученные на претендентов от их прошлых работодателей и других внешних поручителей;
- результаты интервью и любых профессиональных испытаний, которым были подвергнуты претенденты.

Полное использование всех трех видов информации обязательно, пока нет другого «вызывающего доверие» вида информации от людей, которые знают

претендента. Хотя такой вариант часто «для протокола», он, конечно, играет роль в практике найма.

Испытания с целью оценки претендентов делятся на две группы: 1) испытания способностей и 2) психометрические испытания [96]. **Испытания способностей** предназначены раскрыть навыки человека выполнять некоторые задачи типа программирования на конкретном языке. **Психометрические испытания** предназначены раскрыть позицию человека и пригодность для некоторых типов задач, типа способности принимать решения. Испытания обычно проводятся в ограниченное время и с другими экзаменационными воздействиями. Они не могут быть истинным отражением способности человека решать задачи в нормальной рабочей среде.

Люди привлекаются, чтобы заполнить определенные потребности проекта и коллектива. Эти потребности разнообразны и дополняют друг друга. Типичные факторы, которые определяют выбор штата, следующие [96, 59]:

- предметная область и опыт в бизнес-процессе;
- технические знания, опыт и знакомство с методами и инструментальными средствами;
- образовательная база;
- способность решать задачи;
- коммуникабельность;
- адаптируемость, готовность и пригодность;
- позиция, амбиции и инициатива;
- индивидуальность.

Как сказано мимоходом, сильной стороной современных рабочих коллективов является гибкое сотрудничество людей, где отдельные индивидуумы не являются обязательно лояльными к организации, но полагаются на рабочую среду и ряд абстрактных, не связанных с работой потребностей. Привлечение служащих, вероятно, будет меньшим беспокойством в управлении, чем их мотивация и сохранение. Как отмечено Бенсоном и Стандингом [7], инженеры ПО обычно бывают богатыми в денежном плане и бедными в смысле свободного времени. Мотивация и сохранение таких служащих требует нетрадиционных подходов типа разрешения работы на дому или предоставления большего количества свободного времени.

Теории мотивации

Мотивация может быть **внешней** или **внутренней**.

Внешние факторы мотивации — это материальные блага, которые могут включать премии, использование служебной машины, фондовые опционы, дарственные, возможности обучения и т. д. Внутренние факторы мотивации различны для разных индивидуумов. Некоторые люди по своей природе стремятся получить результат — это часть их характера. Культурные и религиозные влияния также являются внутренними факторами мотивации. Награды и признания ... являются примерами внешних факторов мотивации [37].

Имеется много мотивационных теорий, которые пробуют определить, что побуждает людей в их действиях и какие факторы влияют на профессиональное поведение людей. Явно или неявно, все популярные теории подтверждают, что люди мотивированы лучше всего, когда дана возможность реализовать абстрактные потребности и цели высокого уровня для «души». Фундаментальные потребности и цели низкого уровня для «тела» — получение материальных благ типа пищи, одежды и даже денег являются слабыми факторами мотивации в добавление к «стандартному уровню существования». Стандартный уровень существования означает здесь, что эти факторы мотивации предоставляются человеку только в необходимом количестве. Увеличение этого количества для лучшей мотивации людей приносит только краткосрочные эффекты.

Хелдман [37] резюмирует четыре хорошо известные мотивационные теории:

- Иерархия потребностей Маслоу.
- Теория гигиены.
- Теория ожидания.
- Теория достижения.

Иерархия потребностей Маслоу классифицирует мотивационные потребности на пять групп: 1) физиологические потребности в продовольствии, крове и т. д., 2) потребности в безопасности для физического благосостояния и безопасности имущества, 3) социальные потребности во взаимодействии с другими людьми: чувство причастности, любовь, дружба и т. д., 4) потребности в признании, уважении, успехах и т. д. и 5) потребности самовыражения в личном плане и в продвижении. Теория Маслоу доказывает, что как только потребность низшего уровня была выполнена и непрерывно поддерживается, она уже не служит больше фактором мотивации. Следующий более высокий уровень становится новым фактором мотивации.

Теория гигиены Фредерика Херцберга (известная также как теория мотивации гигиены) фиксирует два мотивационных фактора: 1) факторы гигиены и 2) факторы мотивации. Факторы гигиены имеют отношение к условиям работы: персональным отношениям, оплате, выгодам и т. д. Факторы гигиены предотвращают неудовлетворенность, но не являются факторами мотивации в истинном смысле слова. Факторы мотивации — это аспекты работы, которые приносят удовлетворение от хорошо сделанной работы. Факторы мотивации — это возможности для персонального продвижения, программы обучения, работа с многообещающими, но выполнимыми задачами и т. д.

Теория ожидания утверждает, что люди действуют и ведут себя согласно ожиданиям, определяемым их окружающей средой, если эти ожидания приносят награду. Желание получить результат, который удовлетворяет ожиданиям, приводит к мотивации. Чтобы быть истинным фактором мотивации, любая обещанная награда должна всегда выполняться. Менеджеры, стимулирующие высокие ожидания от выполнения работы и открыто награждающие за высокое качество, вероятно, создадут коллективы, которые будут соответствовать ожиданиям, и наоборот.

Теория достижения различает три мотивационных фактора: 1) достижение (желание преуспеть), 2) власть (желание влиять на других) и 3) причастность (желание принадлежности к коллективу и товарищеские отношения). Теория не производит дифференциацию среди этих трех факторов мотивации. Она просто утверждает, что сила каждого из этих желаний определяет совокупность, определяющую работу коллектива.

5.1.2. Организация связи в проекте

Связь — это операция обмена информацией и знаниями. Это процесс передачи сообщений от отправителей к получателям. Для многих задач и ролей выполняемой работы возможности связи могут быть более важны, чем технические возможности. Возможно, это истинно и для работ по управлению. Каждый аспект творческой деятельности, типа проектов разработки ПО, обеспечивается связью.

Управление процессом создания ПО рассматривает связь проекта, по крайней мере, с четырех точек зрения [78, 96, 37, 59]:

- формы связи;
- линии связи;
- облегчающие (тормозящие) факторы связи;
- связь в разрешении конфликтов.

Формы связи

Сообщения передаются в некоторой закодированной форме. Код может быть устным, представленным в тексте, в рисунках, в символах, переданным выражением лица или интонацией и т. д. Получатель должен понять код отправителя, чтобы связаться с ним. Сообщение может быть конфиденциальным, публичным, формальным, неофициальным, внутренним, внешним, горизонтальным (между равными по положению), вертикальным (между руководителями и подчиненными) и т. д. Типичные **формы связи**:

- случайные связи по принципу «от человека к человеку»;
- встречи в помещении по принципу «от человека к человеку»;
- телефонные беседы по принципу «от человека к человеку»;
- конференции в Интернете по принципу «от человека к человеку» (текстовые, голосовые и/или видео);
- телефонный автоответчик;
- факсимильная почта;
- почта;
- курьерская почта;
- электронная почта;
- Web-страницы.

Различные формы связи имеют свои уникальные преимущества и недостатки [59]. Формы «*от человека к человеку*» могут передавать значительно большее количество информации изменением тона, выражениями лица, «дви-

жениями тела» и т. д. Если не осуществлять запись, связь «от человека к человеку» может передавать даже большее количество информации, допуская большую степень свободы в выражениях без опасения последствий от чрезмерного распространения записей текста/голоса/видео.

Электронная почта (Email) — мощная, но опасная среда связи. Мало того, что сообщения электронной почты могут быть распространены далее, они могут также быть отправлены или скопированы по ошибке. Электронная почта дает вводящее в заблуждение чувство конфиденциальности связи «от человека к человеку», но это может закончиться формальным и широким распространением информации. Кроме того, при передаче намного более вероятно, чем у других форм связи «от человека к человеку», что сообщение, первоначально обеспечивающее конфиденциальность, затем становится публичным. В отличие от *обычной почты* отправитель электронной почты, как правило, может знать, дошло ли сообщение до получателя или даже прочитано ли им. Вообще, электронная почта не должна использоваться в щекотливых ситуациях.

Web-страницы — мощная среда связи. Эти страницы обеспечивают быструю передачу информации большим группам людей. Они не разрушительны в том смысле, что адресуемые получатели могут посещать страницы по своему желанию. Однако связь через Web-страницы должна гарантировать своевременное обновление информации в них, иначе они будут восприниматься как ненадежные, и получатели перестанут к ним обращаться.

Из всех *почтовых* форм текстовая форма доминирует в формальной связи. Голосовые и видео-формы сравнительно нечасты. Они требуют слишком большого количества трудозатрат и времени для «расшифровки», чтобы играть существенную роль в формальных обменах информацией.

Линии связи

Связью нужно управлять. При этом число **линий связи** между людьми и коллективами должно быть минимизировано, оставаясь при этом достаточным для эффективного распространения информации и знаний. Проблема линий связи идет в параллель с проблемой сложности «проводов», упомянутой в разделе 1.1.6. Управление сложностью требует связи через иерархические структуры, группируя людей в коллективы и обеспечивая оптимальное соединение вертикальных и горизонтальных линий связи.

В плоских организационных структурах число линий связи растет по экспоненте с увеличением участников связи [78]. Рис. 5.2 демонстрирует увеличение линий связи, когда число участников растет от трех до пяти. Каждая линия связи разрешает обмен информацией в обоих направлениях — от отправителя к получателю и наоборот. Приведенная формула объясняет вычисление. Уменьшение линий связи может быть достигнуто созданием коллективов и проведением линий связи между коллективами через руководителей групп/менеджеров. Результат подобен показанному в разделе 1.1.6 на рис. 1.5.

На рис. 5.2 приведена формула для определения числа возможных коллективов, которые могут быть созданы из данного числа участников. Формула определяет число возможных комбинаций из данного числа элементов. На-

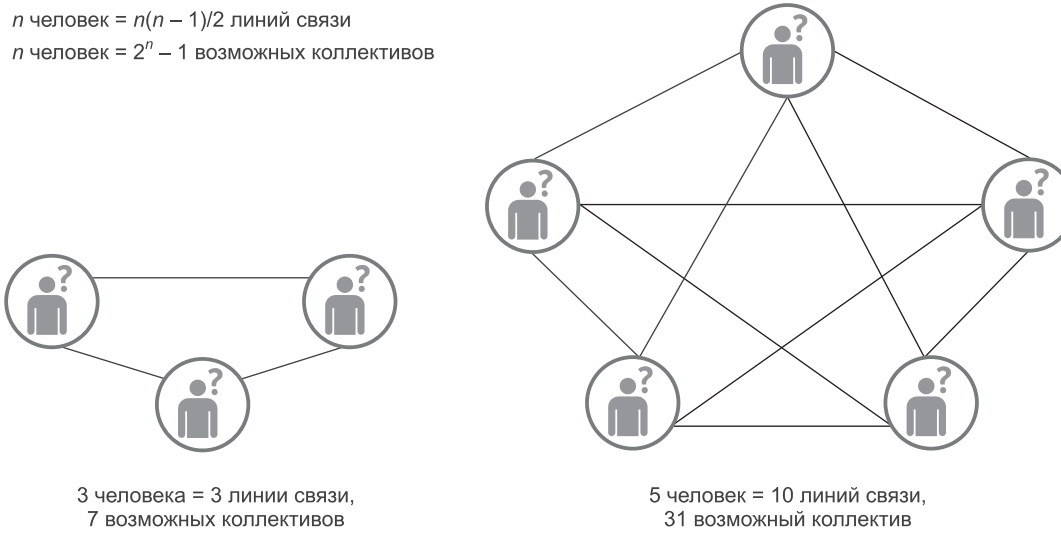


Рис. 5.2. Линии связи и возможные коллективы

пример, с тремя участниками A, B и C комбинации будут следующие: A, B, C, AB, BC, AC и ABC. Естественно, в любой конкретный момент времени максимальное число коллективов намного меньше и равно максимальному числу участников (то есть, числу коллективов из одного человека).

Показатели связи

На практике обычно не требуется, чтобы каждая конкретная часть информации была передана всем участникам проекта. И слишком большая, и слишком малая связь может вредить проекту. Члены коллектива должны быть информированы относительно всего, что воздействует или касается задач, которые они выполняют. Участники проекта должны также быть информированы относительно глобальных проблем, проектных направлений, отобранных политических решений и т. д., чтобы они чувствовали себя неотъемлемой частью проекта и организации.

Опасность иерархических структур и вертикальных сообщений состоит в том, что люди на одном и том же уровне имеют недостаточную связь [96]. Имеются различные пути улучшения связи внутри таких коллективов. Один из них — это такой подбор членов коллектива, чтобы гарантировать надлежащее объединение экстравертов и интровертов [78]. **Экстраверт** — человек, который легко вступает в связь с другими людьми. **Интроверт**, в противоположность этому, застенчивый, тихий человек, часто очень хороший слушатель. (Кстати, навыки внимательно слушать столь же важны в связи, как и способность говорить.)

Выше было сказано, что значительное количество связей является несущественным. Это и истинно, и желательно. Организация рабочего места может существенно облегчить или запретить заранее спланированную и/или **несущественную связь** [96]. Например, работа в помещениях с открытой пла-

нировкой с или без индивидуальных кабин и поощряет связь, и препятствует ей в одно и то же время. Отрицательное воздействие идет от прерываний со стороны других работников из-за нахождения в помещении с открытой планировкой и от недостатка уединенности.

Никакое количество индивидуальной связи не может заменить потребность в организационных формах **групповой связи**. Групповая связь требует комнат для собрания групп, должным образом оборудованных и размещенных, в легко доступных местах, и которые можно использовать как для формальных, так и для неофициальных встреч.

Связь в разрешении конфликтов

Конфликты в коллективах неизбежны, и связь является основным средством их разрешения. Хелдман [37] рассматривает пять технологий для **разрешения конфликтов**. Первые три производят длительный эффект; остальные две дают только временный результат. Эти технологии следующие:

- принуждение;
- компромисс;
- конфронтация;
- сглаживание;
- изъятие.

Принуждение в разрешении конфликта делает то, что оно и предполагает. Оно воздействует на членов коллектива, которые находятся в конфликте. Принуждение возможно в случае зависимости начальник-подчиненный. Принуждающая сторона должна принять решение, но другая, вероятно, не будет согласна с этим. Начальник — это менеджер проекта или руководитель проекта. Решения, которые воздействуют на людей, обычно увеличивают авторитет менеджера проекта (если не используются слишком часто), но ослабляют обаяние и влияние руководителя проекта. Передача такого решения заинтересованным сторонам может быть сделана в форме связи, которая не будет из категории «один на один».

В противоположность этому **компромисс** требует связь категории «один на один». Он требует, чтобы стороны конфликта высказали свое мнение и достигли средневзвешенного решения без победителей или проигравших. Поскольку компромисс предполагает согласование, решение может быть длительным.

Конфронтация — наиболее эффективное и наиболее желаемое решение, когда существует один правильный ответ на конфликт и когда конфликт может быть связан с определенной проблемой (например, столкновение индивидуальностей). Конфронтация называется также решением проблемы. Как только факты установлены и решение проблемы определено, это решение может быть принято, и конфликт исчезает.

Сглаживание — в какой-то мере временный путь решения конфликта, в котором проблема, подкрепляющая конфликт, выглядит менее важной, чем есть на самом деле. По этой причине масштаб конфликта уменьшается. Сглаживание часто является примером «промывания мозгов». Вероятно, что конфликт позже снова возникнет, причем с увеличенной силой. Только в некото-

рых удачных случаях, когда проблема устраняется другими, казалось бы, несвязанными решениями, сглаживание обеспечивает постоянное решение конфликта.

Изъятие — другой временный способ разрешения конфликта, в котором одна сторона не видит никакого смысла в обсуждении конфликта когда-либо в дальнейшем. Это, несомненно, самый плохой результат разрешения конфликта, по крайней мере, пока обе стороны остаются в коллективе: мало того, что связь будет не в состоянии разрешить проблему, но, кроме того, изъятие приведет к ослаблению связи между членами коллектива также и на всех других проектных проблемах.

5.1.3. Создание коллектива

Привлечение и мотивация людей (раздел 5.1.1), а также связи проекта (раздел 5.1.2) — две доминирующие проблемы в **создании коллектива** (называемом также **построением коллектива**). Другие проблемы включают организацию коллектива, размещение, обучение, аттестацию, финансовые отчеты, внешнюю обратную связь и т. д.

Коллективы проходят четыре стадии создания [37]:

- формирование;
- волнение;
- нормализация;
- выполнение.

Формирование коллектива начинается с первой встречи служащих на проекте [59]. Это должна быть «строго деловая» встреча, на которой представляются члены коллектива, объясняются их роли, представляется график проектных работ, поднимаются начальные проблемы, описываются методы и инструментальные средства и т. д. В более позднее время может последовать ряд неофициальных встреч, чтобы «сломать лед».

Неизбежно, если имеется поле сложных проблем, коллектив входит в стадию **волнения**. Некоторые проблемы проекта кончаются конфронтацией и конфликтами. Источники проблем могут быть разнообразные — планы, приоритеты, управление, деньги, методика, конкретные лица и т. д. На этой стадии определяются лидеры и аутсайдеры.

Следующая стадия — **нормализация**. Подталкивание и выталкивание уступают место сотрудничеству. Члены коллектива знают свои места в проекте. Они занимаются проблемами, а не отношением друг с другом. Совместные решения достигаются без трений. Зарождается дружба.

Стадия **выполнения** достигается только в больших коллективах. На этой стадии эффективность и результативность коллектива лучше, чем на стадии нормализации. В коллективе имеется большое понимание и очень сильные побуждающие мотивы для успеха. Все члены коллектива имеют ощущение принадлежности к общему делу и стоящей цели, так же как и высокое удовлетворение работой.

Создание коллектива включает его **организацию**. Имеются многочисленные модели, предложенные для организации коллектива, но практически редко можно увидеть коллективы, построенные в точном согласии с теоретичес-

кими моделями. Это, вероятно, отражает уникальность проектов ПО, различия в жизненных циклах ПО, специфику организационных культур и другие подобные проблемы. Шах [92] обсуждает и сравнивает семь различных подходов к организации коллектива.

Создание коллектива — длительный процесс. Оно требует постоянного подкрепления. Даже небольшие изменения в структуре коллектива, принципах работы, проектных направлениях могут вынудить начать заново цикл из четырех стадий. Большой коллектив может превратиться в неработоспособную группу служащих. Чтобы противодействовать этому, создание коллектива должно рассматриваться как важная и составная часть совершенствования процесса создания ПО. При создании коллектива необходимо определить операции совершенствования работы, включая рабочую среду, размещение служащих в рабочих помещениях, планы обучения и т. д. Существенный аспект создания коллектива — оценка работы. Хорошая работа должна быть публично признана после того, как это произошло, но без установления премий и наград, которые, вероятно, вызовут персональные напряженные отношения в коллективе.

5.2. Управление рисками

Понятия риска и **управления рисками** были представлены в разделе 3.1.6 и упоминались на протяжении всей главы 4. Ссылки на риски в главе 4 показывают, что управление рисками — важный момент в планировании и отслеживании ПО.

Как отмечается Линцем и Риа [59], проект может быть намного далее продвинут в смысле плана и бюджета, нежели в смысле проблем риска, потому что большинство их все еще требует решения. Скажем, например, что проект может быть на 80 процентов готов, но риск его неудачи может все еще составлять 50 процентов. Даже хуже, проект может быть закончен и представлен пользователям, в то время как некоторые проблемы все еще останутся нерешенными. Например, может быть весьма вероятно, что работа системы недопустимо ухудшится при случайных пиковых рабочих нагрузках или с ростом БД.

Стратегии управления рисками находятся в диапазоне от **реактивных** (reactive) до **упреждающих** (proactive). Реактивные стратегии рисков задерживают их обработку, пока те не станут в проекте критическими факторами. Упреждающие стратегии риска предлагают воздействовать на риски, как только они возникают или даже в ожидании их возникновения. Между этими двумя подходами имеются стратегии, которые поощряют упреждение, адресованное только наиболее неотложным проблемам риска в смысле потенциальных потерь (нежелательных воздействий) и в смысле срочности решения.

Хотя упреждающие подходы и одобряются, Линц и Риа [59] обращают внимание на нежелательные последствия «слишком быстрого погружения» в проблему:

- Не каждый симптом заканчивается проблемой (следовательно, ресурсы могут быть израсходованы на проблему, которая никогда бы не возникла).

- Функциональные руководители могут относиться к проблеме несерьезно и могут выработать неправильное мнение относительно сообщения менеджеров/руководителей проекта о проблеме как о ложной тревоге и паникерстве.
- Слишком быстрая реакция или принятие действий без полной информации может ухудшить проблему.
- Время и терпение могли бы дать лучшее решение.

В этом разделе рассматривается процесс управления рисками. Типичный процесс включает, по крайней мере, три стадии:

1. идентификация рисков;
2. оценка рисков;
3. обработка рисков.

5.2.1. Идентификация рисков

Идентификация рисков — утомительный процесс составления списка возможных проблем риска, полученных методом мозгового штурма и с использованием совета экспертов и опыта менеджеров. Список должен быть преднамеренно ограничен в размере, например, до 30 рисков. Риски с низким потенциальным вредом и с низкой вероятностью возникновения могут быть исключены из списка.

Чтобы помочь с идентификацией, сначала должны быть определены типы рисков. Обычно это следующие типы рисков:

- продукт проектирования (то есть риски из-за размера проекта, уникальности, сложности и т. д.);
- процесс проектирования (из-за неправильного учета или неточного следования процессу);
- люди (из-за столкновений индивидуальностей, некачественного управления, распределения людских ресурсов и т. д.);
- бизнес (из-за изменений в бизнес-условиях, политике и т. д.);
- организационный (из-за реструктурирования, финансовых трудностей и т. д.);
- технология (из-за несоответствий технологий или изменений технологий);
- инструментальные средства (из-за неспособности инструментальных средств разрабатывать нужное ПО, преобразования фирм-продавцов инструментальных средств и т. д.);
- клиенты (из-за трудностей связи, недостатка интереса к проекту и т. д.);
- требования (из-за изменений, незавершенности, неточно указанных требований и т. д.);
- другие проекты (из-за зависимости от воздействия других проектов);
- внешние факторы (из-за государственного регулирования, юридических воздействий и т. д.).

Конкретные риски сильно меняются от проекта к проекту. Типы рисков помогают в их обнаружении. Практически организации используют различные контрольные списки рисков, просматривают их и точно определяют те

риски, которые характерны для проекта. Соммервиль [96] предлагает краткий список частых рисков:

- реорганизация штата (потеря опытных работников во время работы над проектом);
- изменение управления (новое управление устанавливает другие приоритеты);
- непригодность аппаратных средств (аппаратные средства не доступны, когда это необходимо);
- изменение требований (большее изменение требований, чем предусмотрено в плане);
- задержки спецификаций (задержки со спецификациями требуемых компонентов и интерфейсов);
- недооценка размера (система оказывается больше и сложнее, чем ожидалось);
- недостатки CASE-средств (неспособность CASE-средств в достаточной мере помочь разработчикам);
- изменение технологии (изменения в выбранной технологии проекта);
- конкуренция изделия (конкурирующие изделия угрожают жизнеспособности проекта).

5.2.2. Оценка рисков

Оценка рисков (называемая также анализом рисков, расчетом рисков или проектированием рисков) анализирует идентифицированные риски двумя способами:

1. возможностью или вероятностью возникновения риска;
2. отрицательным воздействием на проект.

Возможность может быть задана одним из пяти значений: очень вероятно, весьма возможно, вероятно, маловероятно, вряд ли возможно. С другой стороны, **вероятность** может принимать одно из девяти значений, то есть, в десятках процентов: 10 процентов, 20 процентов и т. д. до 90 процентов. Риск со 100-процентной вероятностью — это не риск, а уже уверенность, т. е. проектное ограничение или требование.

Отрицательное **воздействие** риска может использовать масштаб с пятью значениями типа катастрофического, критического, серьезного, терпимого, минимального. Решение должно быть принято в зависимости от стандартных результатов рисков, в отношении которых может быть оценено отрицательное воздействие. Обычное множество таких стандартных результатов или **компонентов риска** следующее [81]:

- риск плана (что не удастся твердо придерживаться плана);
- риск бюджета (что произойдет перерасход бюджета);
- риск использования (что у изделия не будет спроса для использования);
- риск возможности сопровождения (что изделие не будет ремонтопригодно или расширяемо).

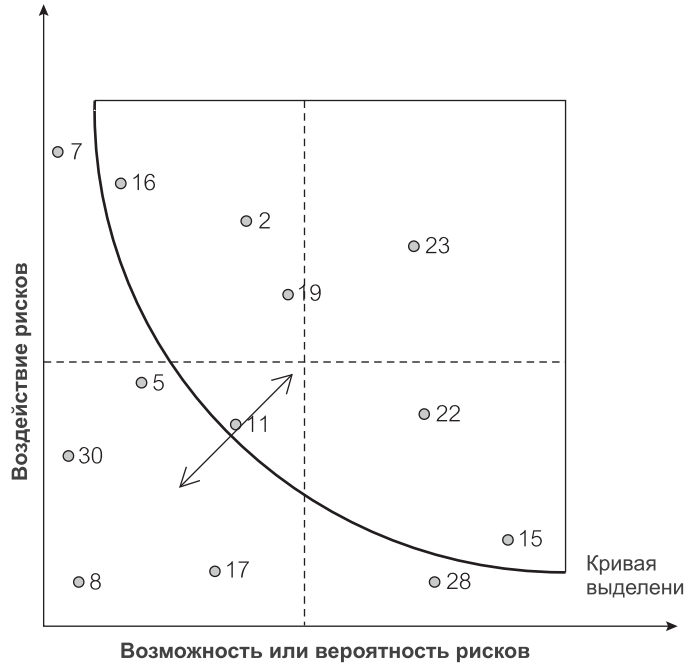


Рис. 5.3. Подверженность рискам и выделение области управления рисками

Чтобы иметь возможность оценить риски, отрицательное воздействие должно быть нормализовано и выражено в денежных величинах. Тогда можно вычислить **подверженность риску** для каждого риска, умножая возможность/вероятность на воздействие.

Подверженность риску можно также получить, используя графическое расположение, как показано на рис. 5.3, где одна из осей указывает масштаб возможности/вероятности риска, а другая — масштаб воздействия риска. Риски показываются в виде точек на пересечениях между величинами возможности/вероятности и воздействия. Риски под кривой выделения игнорируются в дальнейшем управлении рисками. Рисками выше кривой выделения необходимо дальше управлять. Чтобы изменить число рисков, которыми следует управлять, можно переместить кривую выделения в направлении стрелок.

Рис. 5.3 предполагает, что воздействие риска и возможность/вероятность риска взвешены одинаково в оценке важности рисков. Из тринадцати рисков, изображенных на рис. 5.3, семь находятся выше кривой выделения и ими будут в дальнейшем управлять. Шесть рисков находятся под кривой и в дальнейшем будут игнорироваться управлением рисками.

Перемещение кривой выделения в направлении стрелок, показанных на рис. 5.3, изменяет число рисков, которые нужно рассмотреть для дальнейшего управления. Рис. 5.4 демонстрирует результат двойного действия — перемещение кривой выделения слегка вверх и вращение ее, чтобы дать меньшее количество веса возможности/вероятности. В результате этого только пять рисков включены для дальнейшего управления.

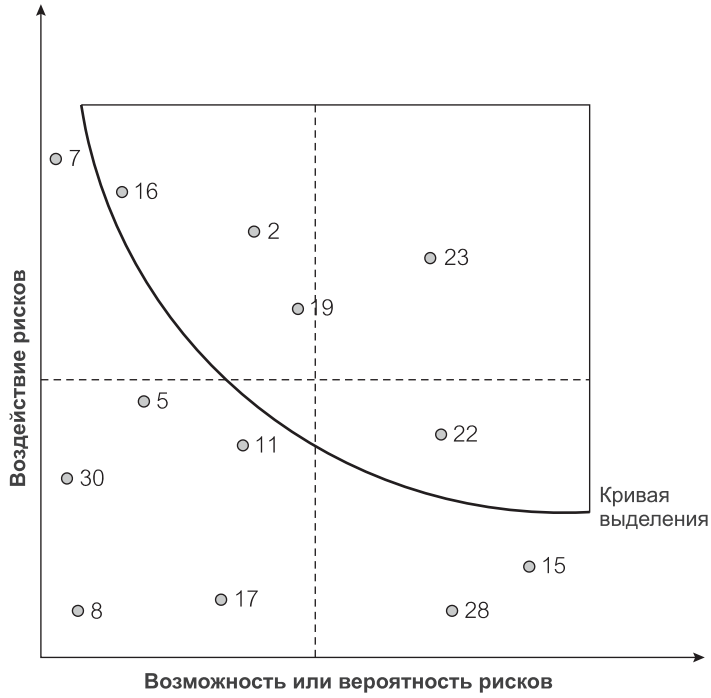


Рис. 5.4. Изменение кривой выделения рисков

Следует заметить, что через какое-то время некоторые риски уменьшаются или исчезают, а другие усиливаются или возникают. Это означает, что положение рисков на графике изменяется. Подобно планированию и составлению бюджета, управление рисками — перемещающаяся цель. Если необходимо, может потребоваться принять решение завершить проект из-за непреодолимых рисков.

Исследование Черитта [20] предлагает **контрольные уровни рисков** как основание для вычисления **точек завершения проекта**. Контрольный уровень рисков — комбинация компонентов риска (план, бюджет, использование, возможность сопровождения). Он определяет уровень проектной терпимости к рискам — точку останова, в которой проект должен быть закрыт. Рис. 5.5 — графическая визуализация контрольной кривой завершения для двух из этих четырех компонентов [96]. Проект должен быть закончен, если *контрольная точка* текущего проекта находится в области завершения проекта выше контрольной кривой.

На практике контрольный уровень не может быть графически представлен просто как контрольная кривая. Во-первых, графическую визуализацию непросто реализовать, если компонентов риска больше двух. Во-вторых, контрольные точки не формируют гладкую кривую. Они создают рваную линию или скорее зону с областями возможного завершения.

Понятно, что завершение проекта — тяжелое решение, обычно с важными персональными последствиями. Политика, связанная с завершением проекти-

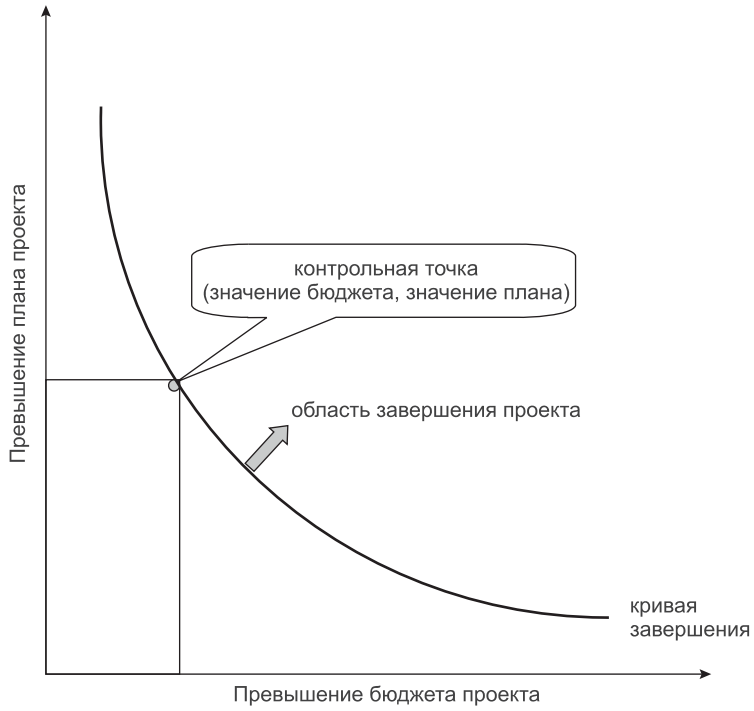


Рис. 5.5. Контрольные уровни рисков и точки завершения проекта

рования, часто основывается исключительно на оценке управления рисками. Исключая политику, оценка, действительно ли проект должен быть закончен, всегда полна субъективных суждений. Субъективность — не так уж и много в сравнении с вычислением контрольных уровней, как и с входными величинами для такого вычисления: величинами воздействия риска и возможностями/вероятностями.

5.2.3. Обработка рисков

Как было упомянуто, риски изменяются во времени, и их следует постоянно отслеживать. Результаты отслеживания риска используются для корректировки планов рисков, после чего цикл планирования-отслеживания продолжается. Планирование рисков и стратегия отслеживания могут формировать отдельный **план управления рисками** или могут быть включены в общий план управления проектом.

Кроме документа, который определяет операции, связанные с управлением рисками, план управления рисками вряд ли будет бумажным документом. Это скорее **БД рисков** — средство/приложение БД, позволяющее по своему желанию размещать, искать, задавать приоритеты, сортировать, корректировать данные и формировать отчеты. БД рисков используется для всех трех стадий управления рисками: идентификация, оценка и обработка.

Предполагая упреждающий подход к управлению рисками, **обработка рисков** включает комбинацию трех стратегий:

- предотвращение рисков;
- минимизация рисков;
- планирование непредвиденных обстоятельств.

Стратегия **предотвращения рисков** пытается устранить возможность (вероятность) появления риска. Например, если имеется риск потери ключевого члена коллектива из-за его/ее недовольства относительно гибкости рабочего дня, можно прелложить такому человеку работать в некоторые дни дома.

Стратегия **минимизации риска** пытается уменьшить воздействие риска, когда тот появляется. По отношению к предыдущему примеру, чтобы минимизировать воздействие потери штатного работника, может быть обучен другой член коллектива, чтобы соответствовать знаниям и навыкам члена коллектива, который может оставить работу.

Стратегия **планирования непредвиденных обстоятельств** предполагает, что риск может стать действительностью и необходимо знать, как реагировать, когда это случится. План на случай возникновения непредвиденных обстоятельств определяет последовательность действий, которые будут предприняты, когда другие стратегии не смогли ничего сделать, и риск стал фактом. Например, ключевой член коллектива уезжает с уведомлением за две недели, и нет никакой немедленной доступной замены. План на случай возникновения непредвиденных обстоятельств может тогда определить, что служащий (человек, находящийся на контракте) должен быть нанят, бюджет и план должны быть соответственно отрегулированы, а отбывающий служащий должен потратить оставшиеся две недели, чтобы задокументировать состояние своей работы.

5.3. Управление качеством

Управление качеством ПО — всеобъемлющая операция, которая переплетается с большинством других операций управления и охватывает весь жизненный цикл разработки и ее процесс. Оно переплетается и охватывает, однако, все же это независимая операция. Коллектив управления качеством должен быть отделен от коллектива разработчиков и иметь свои собственные каналы информации. Управление качеством должно быть независимым от управления проектом. План управления качеством должен иметь свой собственный бюджет и график, по крайней мере, настолько, насколько интересует сама гарантия качества (раздел 5.3.3).

Управление качеством было предметом интенсивных усилий по стандартизации. Модель зрелости возможностей (Capability Maturity Model — CMM), рассматриваемая в начале этой главы, является де-факто стандартом для процесса управления качеством. Международная организация по стандартизации (International Organization for Standardization) приняла серию стандартов качества ISO 9000. Стандарты применимы к любым отраслям промышленности и всем типам областей операций, включая разработку ПО. Вспомогательный

документ, известный как ISO 9000-3, поясняет использование ISO 9000 для разработки ПО.

Совсем недавно ключевые стандарты в пределах серии стандартов ISO 9000 были слиты в единый стандарт, известный как ISO 9001:2000 [41]. Акцент стандарта направлен на управление ключевыми процессами с целью непрерывного их улучшения (что аналогично уровню 5 CMM). Как и в случае CMM-сертификации, ISO обеспечивает возможность для организаций зарегистрировать свое соответствие процессам, определяемым ISO.

Стандарты управления качеством в первую очередь связаны с определением процессов обеспечения качества, которые гарантировали бы надлежащее качество изделий. Эти процессы отличаются от механизмов качества, связанных с управлением качеством в изделиях. Соответственно, *гарантия качества* (раздел 5.3.3) отличается от *контроля качества* (раздел 5.3.2).

5.3.1. Показатели качества программного обеспечения

Различные пользователи (и различные разработчики) имеют разные представления относительно того, что составляет **показатели качество ПО**. Имеется много желаемых показателей качества, и в зависимости от проекта ПО некоторые из них могут быть более важными, чем другие. Показатели качества ПО должны быть идентифицированы, определены и классифицированы.

Основная классификация делит показатели качества ПО на **показатели качества изделий** и **показатели качества процессов**. Обычно нельзя произвести качественное изделие без качественного процесса, который определяет производственные операции. Цель процесса — изделие. Следовательно, хотя показатели качества могут быть приписаны процессу как таковому, все они могут быть непосредственно связаны с качеством изделия. Обратное также верно.

С точки зрения управления качеством качество изделия должно быть гарантировано и управляться не только для **конечного изделия**, поставляемого клиенту, но также и (и прежде всего) для всех промежуточных **рабочих изделий** (которые известны все вместе под термином **продукты**). Управление качеством в рабочих изделиях требует поддержки инструментальных средств управления конфигурацией, способных хранить и управлять многочисленными продуктами, их версиями и отношениями между ними.

Гецци и другие [34] предлагают превосходный набор типичных показателей качества программных продуктов и процессов. Эти показатели следующие:

- корректность;
- надежность;
- ошибкоустойчивость;
- выполнение функций;
- применимость;
- понятность;
- удобство сопровождения (ремонтопригодность);
- масштабируемость (способность к развитию);

- возможность многократного использования;
- мобильность;
- способность к взаимодействию;
- производительность;
- своевременность;
- видимость.

Корректность (correctness) — ряд формальных свойств ПО, которые устанавливают непосредственное соответствие между программным продуктом и его функциональными спецификациями. Существуют две проблемы с показателем корректности. Во-первых, имеются мощные аргументы в пользу известного высказывания, что невозможно доказать корректность программы, потому что невозможно гарантировать отсутствие ошибок в таких сложных изделиях, как ПО. Во-вторых, функциональные спецификации редко определяются с достаточной точностью и стабильностью, чтобы служить исчерпывающим ориентиром в определении показателя правильности. Поэтому было бы лучше иметь правильное ПО при неправильных требованиях.

Надежность (reliability), также называемая **функциональной надежностью** (dependability), — свойство ПО, обеспечивающее к нему доверие пользователей, потому что оно ведет себя правильно и таким образом, как ожидают пользователи. В идеале понятие надежности ПО должно включать понятие корректности. В то время как это истинно в традиционных технических дисциплинах, в разработке ПО надежные программы могут содержать «известные ошибки». Даже если приходится столкнуться с новыми ошибками, ПО можно все еще считать (достаточно) надежным.

Ошибкоустойчивость (robustness) подразумевает, что ПО вряд ли подведет или испортится, или, по крайней мере, будет безвозвратно разрушено. Ошибкоустойчивость дополняет понятия корректности и надежности. Вместе эти три качества определяют, выполняет ли ПО свои функции так, как ожидалось. Кроме того, все три свойства могут характеризовать качество как изделия, так и процесса.

Выполнение функций (performance) означает работать удовлетворительно в соответствии с заданными целями. Обычной целью является время отклика системы ПО, которое определяет, как долго пользователь готов ждать, пока система ответит на его запрос. В отличие от многих других качеств выполнение функций — черно-белое качество: или система обладает им, или нет.

Применимость (usability) касается дружелюбия к пользователю ПО; насколько легко пользователю работать с ним. Это — очень субъективное свойство. Что является дружественным и легким для одного человека, может быть недружелюбным и сложным для другого. Качество применимости — основная характеристика при проектировании пользовательского интерфейса для системы. Как и в традиционных технических дисциплинах, чем более стандартный и типовой пользовательский интерфейс, тем он более пригоден. Новаторские проекты пользовательских интерфейсов могут выглядеть прелестно, но быть ужасными в употреблении, особенно для новичка, однако грамотного в компьютерном отношении пользователя.

Понятность (understandability) означает легкость, с которой могут быть проанализированы и поняты внутренняя структура и поведение ПО эксплуатационным персоналом. Понятность является условием для удобства сопровождения.

Удобство сопровождения (maintainability) ПО отличается от удобства сопровождения других технических изделий. Части ПО не ломаются из-за длительного использования, скачков напряжения и других подобных причин. Удобство сопровождения при разработке ПО является всеобъемлющей концепцией, означающей решение каждой из следующих трех задач: 1) исправление ошибок и недостатков ПО, 2) приспособление ПО к новым требованиям и 3) совершенствование ПО для получения новых качеств. В узком смысле слова, принятом здесь, удобство сопровождения ограничено только первым пунктом. Это по Гецци [34] называется **ремонтопригодностью** (repairability).

Масштабируемость (scalability) или **способность к развитию** (evolvability) означает легкость, с которой ПО может быть увеличено или развито в ответ на повышение требований к его функциональным возможностям. Масштабируемость может быть достигнута лишь в системах с ясным, понятным структурным проектированием. Как только структура системы ухудшается в результате сопровождения, масштабируемость уменьшается. Понятность, удобство сопровождения и масштабируемость известны под объединенным названием **возможности сопровождения** (supportability).

Возможность многократного использования (reusability) ПО определяет уровень, на котором компоненты ПО могут использоваться для конструирования других изделий. ПО может многократно использоваться двумя способами. Оно может использоваться как **компонент** другого изделия или может использоваться как **шаблон**, который следует настроить, чтобы создать другое изделие. Возможность многократного использования применяется также и к процессам создания ПО.

Мобильность (portability) означает, что ПО может выполняться на различных платформах аппаратных средств (программного обеспечения) без какой-либо модификации или после выполнения незначительной настройки или параметризации. Мобильность имеет экономическое значение. Это качество существенно для системного ПО и в меньшей степени для прикладного ПО.

Способность к взаимодействию (interoperability) определяет способность ПО сосуществовать или работать вместе с другим ПО, возможно, даже с будущим ПО, которое еще не существует. Подобно мобильности способность к взаимодействию существенна для системного ПО. Это качество обеспечивает такое ПО, которое известно как **открытые системы**.

Производительность (productivity) — показатель качества процесса. Производительность определяет скорость, с которой процесс позволяет создать ПО, имея некоторое количество ресурсов. Производительность — мера эффективности и выполнения функций процесса.

Своевременность (timeliness) — другой показатель качества процесса, который определяет способность процесса создать ПО вовремя. «Вовремя» обычно означает согласно плану исходных данных (пересмотренный план,

согласно которому изделие было поставлено, вряд ли будет своевременным). В случае коммерческих изделий своевременность может означать «вовремя для рынка».

Наконец, **видимость** (visibility) — также показатель качества процесса. Видимый процесс — прозрачный процесс — процесс с ясно определенными и задокументированными стадиями и операциями. Видимый процесс является условием хорошего проекта и управления рисками. Он является также условием для организации, чтобы получить CMM-свидетельство или регистрацию ISO.

5.3.2. Контроль качества

Контроль качества главным образом обеспечивается тестированием качества изделия, в противоположность **гарантии качества**, которая характеризуется непосредственным обеспечением качества изделия. Контроль качества имеет реактивный характер. Это в большой степени — упреждающее действие. Контроль качества представляет собой в основном оперативные и тактические действия. Гарантия качества имеет сугубо стратегический аспект. Иногда контроль качества рассматривается как часть гарантии качества.

Прессман [81] приравнивает контроль качества к **управлению вариациями**. Управление вариациями — концепция, используемая в производстве, где производятся многократные копии одного и того же изделия, и они должны быть идентичны. Копия, которая имеет вариации по сравнению с моделью (шаблоном) копии, не удовлетворяет проверке качества.

В разработке ПО многократные копии программного продукта обычно не бывают. Большинство прикладных программных продуктов существует в единственной копии. Когда все же производятся многократные копии, типа коммерческих систем ПО, дублирование является простым вопросом создания компакт-дисков или других носителей данных. Часто коммерческое распределение ПО даже не требует производства многократных копий — ПО можно просто скачать с Web-сайта продавца ПО.

Тестирование ПО

Однако имеется параллель между контролем качества ПО и управлением вариациями. Контроль качества — нечто вроде обнаружения вариаций в работе или конечном изделии по сравнению с данной спецификацией. Спецификация может включать выполнение функциональных требований и/или удовлетворение показателей качества, имеющихся в списке раздела 5.3.1. Цель контроля качества ПО состоит в том, чтобы подвергать ПО частым испытаниям для устранения проблем как можно раньше. Во многих случаях контроль качества ПО синонимичен **тестированию программного обеспечения**.

Как было сказано в разделе 1.2 и в других местах книги, тестирование — это не отдельная стадия жизненного цикла разработки. Подобно управлению качеством, тестирование — всеобъемлющая операция, которая применяется ко всем стадиям жизненного цикла. Естественно, тестирование наиболее интенсивно, когда доступен код. Однако, как это ни парадоксально, тестирова-

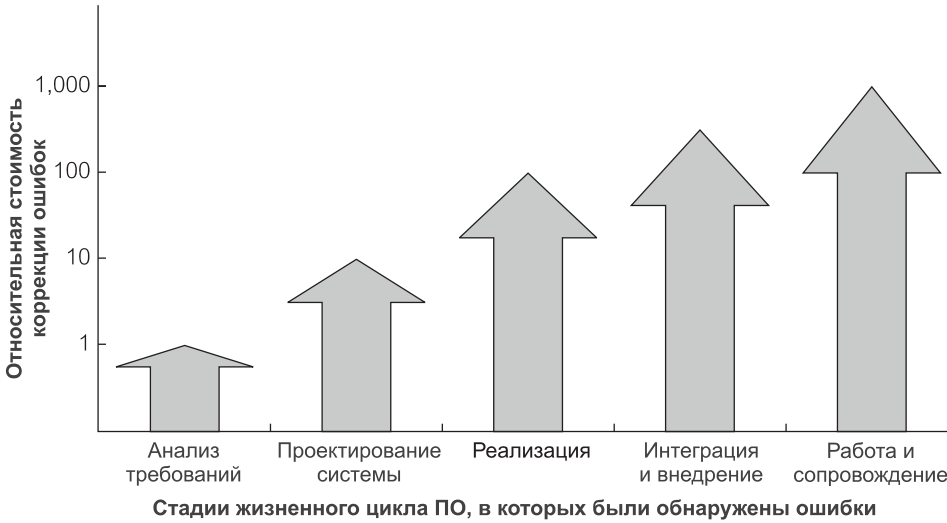


Рис. 5.6. Относительная стоимость коррекции ошибок

ние кода — не обязательно наиболее ценный вид тестирования в терминах плана и бюджета проектных работ.

Как говорилось в старых пособиях IBM, относительная стоимость исправления ошибки растет почти по экспоненте в зависимости от стадии жизненного цикла, в которой эта ошибка была обнаружена. Исследование IBM установило факт, что стоимость исправления ошибки может изменяться в десятки раз на разных стадиях жизненного цикла (рис. 5.6). Следовательно, принимая, что исправление ошибки, найденной на стадии анализа требований, стоит 1 доллар, исправление той же самой ошибки стоило бы целых 10 долларов на стадии проектирования системы, 100 долларов, если она обнаружена на стадии реализации, и целых 1000 долларов, если она обнаружена после того, как система будет развернута пользователями.

Эта повсеместно принятая истина об относительной стоимости исправления ошибок дает дополнительный стимул и определяет важность контроля качества и тестирования операций. Тестирование имеет различные формы в зависимости от стадии разработки. На стадии анализа требований тестирование старается установить полноту требований, чтобы устранить несовместимости и противоречия, согласовать с пользователями все необходимые опытные образцы ПО и т. д. Во время проектирования системы тестирование всесторонне проверяет различные модели проекта, анализирует по спецификациям продукты проекта, гарантирует, что все требования учтены, и т. д. На стадии реализации и позже ведущее положение занимает тестирование кода.

Имеется интересная зависимость между продуктами разработки ПО и тестированием кода. Зависимость состоит в том, что более поздние стадии тестирования концентрируются на проверке более ранних продуктов разработки. Это наблюдение изображено на рис. 5.7 [78]. Например, тестирование интеграции проводится со спецификациями структурного и детального проекта.

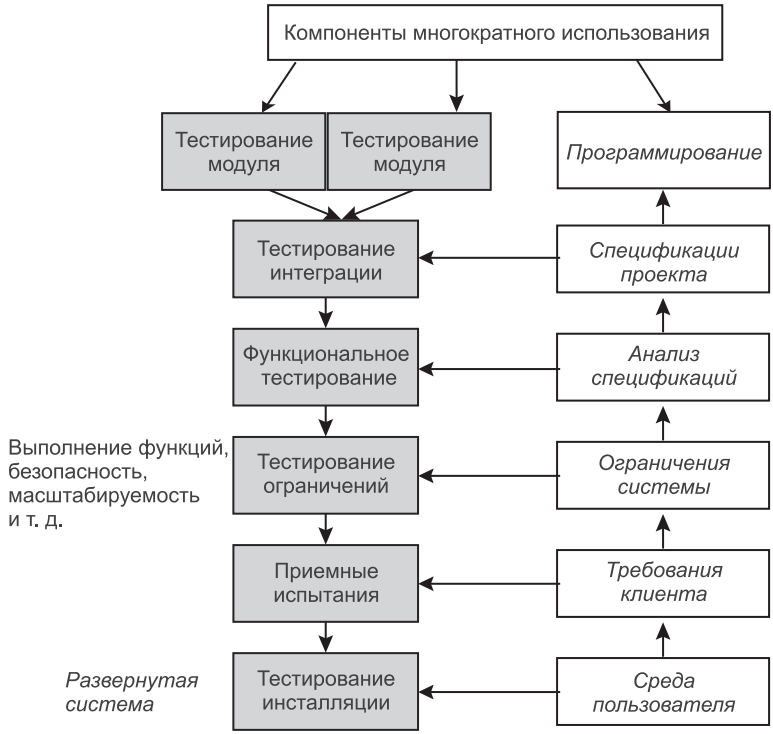


Рис. 5.7. Зависимости между типами тестирования и стадиями разработки

Однако приемочные испытания подчеркивают текущие требования клиента, которые, будем надеяться, не изменились по сравнению с тем, как они были задокументированы для целей проектирования и анализа. Если они все же изменились, приемочные испытания могут дать отрицательную обратную связь от клиента, и возникшие проблемы должны быть решены между клиентами и менеджерами проекта.

Технологии тестирования

Имеется много различных **технологий тестирования**, которые можно комбинировать, чтобы получить лучший возможный охват и результаты тестирования. Независимо от того, насколько обширно тестирование, оно не может быть исчерпывающим и гарантировать корректность программы или системы. Невозможно проверить все возможные входные данные или все пути выполнения кода.

Технологии (стратегии) тестирования могут классифицироваться по разным критериям [109]. Дальнейшее обсуждение основано на пяти критериях: 1) видимость, 2) автоматизация, 3) разделение, 4) охват и 5) использование сценариев (рис. 5.8). Критерии не являются независимыми. Например, две технологии разделения применяются главным образом к тестированию «черного ящика», но не применимы к тестированию «белого ящика». Технологии

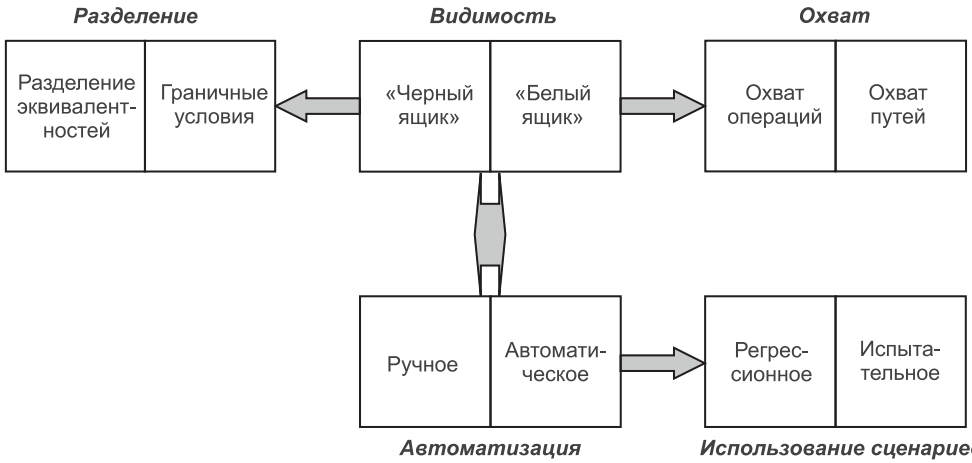


Рис. 5.8. Технологии тестирования

использования сценариев (регрессионные тесты и испытательные тесты) являются автоматическими испытаниями.

Тестирование «черного ящика» (или **тестирование на основе технических требований**) предполагает, что испытатель при тестировании не знает или игнорирует внутреннюю работу программной единицы. Испытуемая единица рассматривается как «черный ящик», который имеет некоторый вход и формирует некоторый результат. Тестирование выполняют, подавая входные данные на испытуемую единицу и проверяя, что получен ожидаемый результат. Поскольку никаких знаний устройства не требуется, тестирование «черного ящика» может выполняться пользователями. Соответственно, оно является главной технологией приемочных испытаний.

Технология «черного ящика» проверяет функциональные возможности системы или функциональные выходы системы, если быть точными. Она также применима для тестирования ограничений, типа выполнения функций или безопасности системы. Особая цель тестирования «черного ящика» — проверка **нарушений функциональных возможностей**, то есть, когда ожидаемые функциональные требования не были реализованы в ПО. В этом случае испытательная единица получает данные, вызывающие такие функциональные возможности, и выдает ошибку, потому что нет необходимого кода, выполняющего требуемые действия.

Тестирование «белого ящика» (или **тестирование на основе кода**) — противоположность тестированию «черного ящика». Испытуемая единица рассматривается как белый ящик или, точнее, прозрачный **стеклянный ящик**, который показывает свое содержание. В этом подходе испытатель изучает код и выбирает входные данные, которые могут «реализовать» выбранные пути выполнения кода. Тестирование «черного ящика» должно обычно сопровождаться тестированием «белого ящика», чтобы точно определить ошибки, обнаруженные при тестировании на основе технических требований.

В отличие от тестирования «черного ящика», тестирование «белого ящика» не ограничивается тестированием кода. Оно одинаково подходит и для других продуктов разработки, типа моделей проекта и документов спецификации. Тестирование этого вида использует технологии просмотра, типа сквозного контроля и инспекции (раздел 5.3.3). Интересным результатом, который можно получить с помощью «белого ящика», но не с «черным ящиком», является выявление существования **«мертвого кода»**. Мертвый код — это операторы программы, которые нельзя выполнить, то есть, они никогда не будут использоваться независимо от того, каким будет вход.

Технологии разделения эквивалентностей и граничных условий применяются главным образом к тестированию «черного ящика», хотя и при тестировании «белого ящика» также можно получить положительный результат. Это определенные подходы к технологии «черного ящика», чтобы противодействовать невозможности проведения исчерпывающих испытаний.

Разделение эквивалентностей группирует входные данные (и, неявно, выходные данные) в области, обеспечивающие однородные цели испытаний. Предполагается, что тестирование с любым членом такой области эквивалентно. Поэтому достаточно использовать только один член этой области, а другими пренебречь. Выбор однородных областей — главная трудность. Области должны быть выбраны с хорошим знанием данных и требований приложения к ним. В этом смысле такое разделение по существу не является в полной мере «черным ящиком» (это даже и не технология тестирования). Технология тестирования — это «черный ящик», который поддерживает разделение эквивалентностей.

Граничные условия — просто дополнительная технология анализа данных, чтобы помочь в разделении эквивалентностей и, следовательно, помочь в тестировании «черного ящика». Граничные условия — предельные случаи в рамках разделения эквивалентностей. Например, если область — множество значений целого числа от 1 до 100, анализ граничных условий будет рекомендовать выполнить испытания с величинами, находящимися на границах областей, то есть для –1, 0, +1, а также для 99, 100 и 101. Естественно, ожидаемый результат для тестирования при –1 и 101 является условием ошибки.

Технологии охвата определяют, сколько кода должно быть выполнено при испытаниях «белого ящика». **Охват операций**, называемый также **охватом методов**, гарантирует, что каждая операция в коде выполнена при испытании «белого ящика», по крайней мере, однажды. Охват операций — современная объектно-ориентированная замена охвата операторов и ветвей, которые применяются в процедурных языках программирования.

Охват путей стремится пересчитать возможные пути выполнения в программе и выполнить их один за другим. Ясно, что число таких путей в больших программах определить не удастся. Испытатель способен определить только наиболее критические и наиболее часто используемые пути, и только эти пути будут проверены.

Тестирование может быть ручное или автоматическое [109]. Человек-испытатель, который взаимодействует с приложением в процессе тестирования, выполняет ручное тестирование. Испытатель запускает приложение, систематически инициирует его функции и наблюдает результаты. Шаги выполнения

проводятся согласно заранее составленному *сценарию тестирования*. Сценарий тестирования определяет последовательность действий тестирования и ожидаемые результаты. Чтобы написать испытательные сценарии, применяются сценарии использования и другие документы технического задания.

Главная проблема с ручными испытаниями состоит в том, что в большинстве практических ситуаций они выполняются с «живыми данными». Нет никаких заранее определенных исходных данных, гарантирующих один и тот же результат для повторяющегося выполнения ручных испытаний. По этой причине в испытательных сценариях ожидаемый результат не всегда определяется точно. Часто результат даже не представляется на экране, но он проявляется в изменениях БД. Это вынуждает испытателя писать и выполнять SQL-скрипты или другие *средства тестирования*, чтобы проверить результаты испытательных действий.

Ручные испытания дороги при подготовке, выполнении и управлении. Они не способны удовлетворить требования «объема» тестирования. **Автоматическое тестирование** использует инструментальные средства тестирования ПО, чтобы выполнить большие объемы испытаний без участия человека. Инструментальные средства могут также сформировать необходимые пост-испытательные сообщения, чтобы облегчить управление результатами тестирования. Естественно, предварительные задачи, такие как решение, что проверять, программирование некоторых испытательных сценариев и установка среды тестирования, должны все еще быть выполнены экспертами-людьми.

Как изображено на рис. 5.8, автоматическое тестирование может быть разделено на регрессионное тестирование и испытательное. **Регрессионное тестирование** — популярный термин, означающий повторяющееся выполнение одних и тех же испытательных сценариев над теми же самыми базовыми данными, чтобы проверить, что полученные ранее функциональные возможности системы не были нарушены последующими изменениями кода, то есть, изменениями, по-видимому, не связанными с проверенными функциональными возможностями.

Регрессионное тестирование выполняется с помощью автоматического выполнения заранее записанных сценариев тестирования в запланированные моменты тестирования. Первоначальные сценарии тестирования записываются **инструментальным средством захвата/воспроизведения** в процессе фиксации действий человека-испытателя во время тестирования приложения. Регрессионное тестирование — операция воспроизведения сценария. Во многих случаях сценарии, захваченные инструментальным средством, изменяются (улучшаются) программистом-испытателем, чтобы обеспечить регрессионное тестирование особенностей, которые инструментальное средство не могло зафиксировать автоматически.

Испытание, названное нами так из-за отсутствия лучшего термина, является действительно автоматическим тестированием охвата. Инструментальное средство, которое реализует испытательное тестирование, производит автоматически и случайным образом различные возможные действия, которые в другой ситуации могли быть выполнены пользователем на приложении в процессе испытаний. Это можно было бы сравнить с безумным пользовате-

лем, ударяющим по всевозможным клавишам на клавиатуре, выбирающим любые возможные команды меню, нажимающим любую доступную кнопку команды и т. д.

Произведенные действия записываются (захватываются) в сценарий, иногда называемый **лучшим сценарием**, который, как кажется, может испытать приложение. Любые ошибки или отказы приложения регистрируются, при этом формируется также отдельный сценарий, создаваемый для действий, которые привели к проблемам. Он называется **сценарием дефектов**. Сценарий дефектов может быть сыгран повторно в любое время, чтобы воспроизвести ошибку и попробовать определить ее причину.

Поскольку и регрессионные, и испытательные тесты используют инструментальные средства захвата/воспроизведения, которые заканчиваются программами на некотором языке сценариев, они могут питать друг друга, таким образом создавая мощную автоматическую среду тестирования. На практике трудности с автоматическим тестированием заключаются не в их проведении, а в запуске последовательных тестов устойчивой среды тестирования с идентичной БД системы отладки, с тем же самым состоянием открытых и активных приложений на испытательном автоматизированном рабочем месте, с предсказуемой скоростью работы сети, с передачей по сети на закрытое автоматизированное рабочее место, с тем же самым отображением рабочего стола компьютера (включая разрешение экрана, цвета рабочего стола, шрифты) и т. д. Наконец, все автоматические тесты должны быть зачищены после работы, соответственно должна быть заново создана БД и прикладная система отладки, как это было до начала испытаний.

Планирование испытаний

План испытаний — часть плана управления качеством. План испытаний определяет график тестирования, бюджет, задачи (контрольные примеры) и ресурсы. План не должен быть ограничен только тестированием кода. Он должен включать тестирование других продуктов проекта. Он должен также быть связан с задачами управления изменениями, ответственными за обработку дефектов.

Часть планирования испытания — определение **испытательной среды**, как отдельной от среды разработки. План должен определить, какие задачи тестирования должны быть выполнены в испытательной среде и какие — в среде разработки или производства. Испытательная среда требует распределения людских и материальных ресурсов. Должны быть определены испытательные автоматизированные рабочие места, создана испытательная БД и установлены испытательные инструментальные средства ПО.

Рис. 5.9 — модель классов концепций тестирования, существенных для планирования испытаний. План испытаний разработан на основе **контрольных примеров** (или задач в традиционном смысле планирования). Требования представляют собой входы испытаний к контрольным примерам. Требование может быть требованием сценария использования или требованием тестирования. Идеально требования тестирования получаются от и передаются к требованиям сценариев использования. Поэтому контрольные примеры должны содержать операторы для требований тестирования.

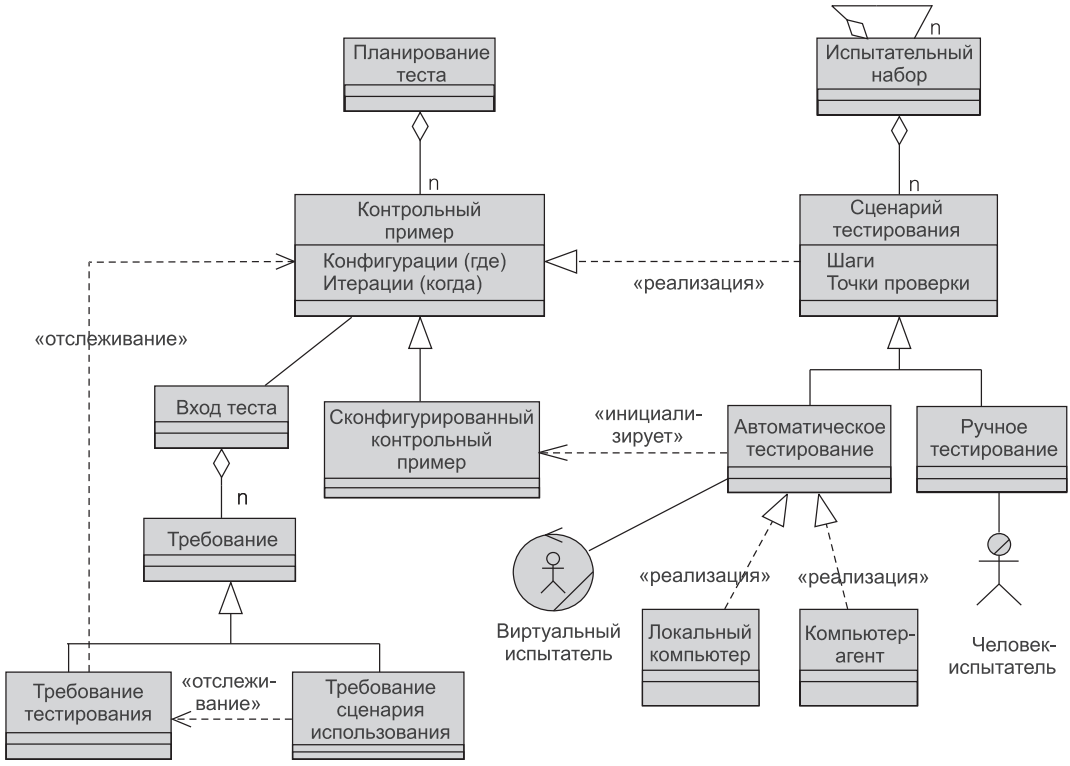


Рис 5.9. Концепции тестирования

Каждый контрольный пример определяет конфигурацию аппаратного/программного обеспечения для проведения испытаний, а также рекомендации насчет итераций, когда должны быть выполнены испытания. Сконфигурированный контрольный пример может инициализироваться для выполнения как автоматический тест.

Контрольные примеры реализуются в **сценариях тестирования**. Каждый сценарий тестирования состоит из детального перечня испытательных шагов и **точек проверок** (которые являются вопросами, проверяющими любые существенные результаты испытательных шагов). Сценарии тестирования могут быть объединены в **тестовые наборы**. Тестовые наборы могут содержать другие тестовые наборы. Автоматическое тестирование может выполнить тестовые наборы как единые сущности.

Сценарий тестирования может быть предназначен для автоматического или ручного тестирования. Человек-испытатель выполняет ручное тестирование. Виртуальный испытатель, являющийся автоматизированным рабочим местом, которое может запускать приложение для испытания и выполнять испытательную программу на этом приложении, выполняет автоматическое тестирование. Автоматические тесты могут запускаться на локальном компьютере (внутри среды разработки) или на компьютере-агенте (внутри испытательной среды). Запуск теста на компьютере-агенте не устраняет возможность просмотра и записи результатов испытаний на локальном компьютере.

5.3.3. Гарантия качества

Контроль качества и гарантия качества — две стороны одной и той же медали — управления качеством. Они включают цикл обратной связи в процесс разработки. Кроме уже упомянутых различий между контролем качества и гарантией качества, по-видимому, наибольшее различие заключается в том, что **гарантия качества** является функцией уровня управления (даже притом, что некоторые операции гарантии качества проводятся самими разработчиками).

Группа **гарантии качества программного обеспечения** (Software Quality Assurance — SQA), назначенная на уровне управления, обеспечивает эту функцию. Группа независима от разработчиков и формирует сообщения функциональному управлению (это могло бы быть на эксплуатационном, тактическом или даже стратегическом уровне в зависимости от важности проекта). Будучи функцией управления, гарантия качества контролирует не только изделия разработки и процессы, но также и планы проектов — график, бюджет, распределение и использование ресурсов.

Гарантия качества подтверждает соответствие программных продуктов и процессов принятым стандартам. Как утверждает название, гарантия качества — операция подтверждения — подтверждения всем заинтересованным сторонам проекта (владельцам, клиентам и разработчикам), что качество системы является высоким.

Из диапазона технологий гарантии качества три заслуживают более пристального взгляда. Это: 1) контрольные списки, 2) обзоры и 3) ревизии.

Контрольные списки

В повседневной жизни контрольный список — это список того, что нужно сделать. В гарантии качества **контрольный список** — список проблем и вопросов, относительно которых программный продукт или процесс проверяется на характеристики качества. Это — исходные данные мер качества, ожидаемых в ПО. Хотя контрольный список, вероятно, наиболее примитивен из всех технологий гарантии качества, его удобно использовать, и он может обеспечить раннюю оценку и предсказание качества изделия или процесса.

Контрольный список может быть просто списком относительно независимых контрольных точек или это может быть такой логической последовательностью шагов, когда каждый шаг получает информацию от предыдущего шага. В первом случае контрольный список походит на анкетный опрос. Во втором случае контрольный список может содержать ветви, чтобы направлять далее проверки по определенным путям. Элементы контрольного списка могут быть расположены по приоритетам, согласно их важности, или по времени, когда должны быть выполнены проверки.

Контрольные списки могут использоваться независимо или как часть другой задачи гарантии качества, типа обзора или ревизии. Фактически контрольный список обеспечивает жесткое управление формальным просмотром (сквозной контроль или экспертиза). Организации по стандартизации производят контрольные списки, которые организации, создающие ПО, используют, чтобы проверить, соответствуют ли их процессы стандартным требованиям. Вопросы контрольного списка должны быть направлены на то, чтобы получить CMM-сертификацию или ISO-регистрацию.

Контрольные списки — важный элемент формирования *культуры качества* в пределах организации. Это важный образовательный рычаг, в особенности, когда используется, чтобы проверить, сконструированы ли модели ПО, включая программы, в соответствии с принципами моделирования и образцами. Формирование ошибки, а затем формирование сообщения о природе ошибки является, возможно, наиболее эффективной учебной технологией. Контрольные списки касаются расположения ошибок и информирования относительно их природы.

Последний пункт проясняет, что качество самого контрольного списка — предпосылка для успешного использования технологии. Создавать контрольные списки должны наиболее квалифицированные люди в организации и люди, которые имеют опыт в разработке ПО. Плохие контрольные списки будут иметь отрицательное, причем длительное воздействие на культуру качества.

Обзоры

Обзор в гарантии качества связан с формальной встречей разработчиков и, возможно, менеджеров, на которой рассматривается рабочий продукт или процесс. Понятие охватывает диапазон предварительно организованных технических совещаний, из которых наиболее известны **сквозной контроль** и **инспекция**.

Обзоры — документы управления. Документ подготавливается заранее и становится доступным для рассмотрения участниками до встречи. Участники, как ожидается, изучат доступный материал при подготовке к встрече. Обзор ограничен небольшим числом людей: *руководителем обзора* (и координатором встречи), разработчиком, чья работа рассматривается (*производителем*), и парой коллег-разработчиков.

Во время встречи производитель объясняет («пробегает») ту часть работы, которая просматривается, и позволяет рецензентам поднять проблемы. На встрече точно определяются и документируются проблемы, но не делается попытка решить их. Выявленные проблемы регистрируются в **списке проблем обзора** [81], который вручается производителю после встречи. Список стимулирует производителя к созданию исправлений в рабочем изделии или процессе. Как только будут сделаны исправления, руководитель обзора информируется об этом и он оценивает работу и сделанные исправления. Если необходимо, собирается повторная встреча.

Основные особенности формального технического обзора следующие:

- Он может использоваться для любого рабочего изделия и процесса, который содержится в документе любого вида (это может быть UML-модель, план испытаний, руководство пользователя, часть кода, определение процесса и т. д.).
- Полные материалы для встречи распределяются и изучаются заранее (подготовка не должна требовать больше пары часов работы каждого рецензента).
- Состав встречи четко определен и обычно ограничен пятью участниками.
- Продолжительность встречи коротка (два часа или меньше).

- Задачи, распределенные между участниками, ясно определены:
  - руководитель обзора оценивает материал с точки зрения его готовности для встречи, распределяет документы среди участников, организует встречу и руководит ей;
  - производитель «пробегает» документ и отвечает на вопросы;
  - все рецензенты, включая руководителя обзора, поднимают проблемы, основанные на их предварительных изучениях (один из рецензентов собирает замечания и составляет список проблем обзора; этот человек называется регистратором или секретарем).
- На встрече точно определяются и документируются проблемы в списке проблем обзора, но не делается попытка решить их (список проблем обзора затем регистрируется как часть документации формального проекта).
- Должны быть приняты все меры, гарантирующие, что на встрече будет рассмотрено рабочее изделие или процесс, но не производитель (и никакие последствия персональной оценки не должны следовать из встречи).
- Число встреч не ограничено, но любая встреча назначается по усмотрению руководителя обзора или руководителя проекта.

Сквозной контроль и инспекции сходны. Главное различие в том, что инспекции выполняются при тесном наблюдении за управлением, менее часты и занимаются более или менее техническими проблемами. Имеется много свидетельств, что формальные обзоры работают хорошо, если они все делают правильно. Они влияют на всю культуру качества, увеличение производительности, обеспечение сроков выполнения, понимание проекта, выработку профессионализма и т. д.

Ревизии

Ревизии имеют более широкие права, чем контрольные списки или обзоры. **Ревизия** — это процесс гарантии качества информационных систем (Information Systems — IS) и информационных технологий (Information Technology — IT). Этот процесс подобен по своим возможностям, требуемым ресурсам и уровню предпринятой экспертизы традиционным ревизиям бухгалтерского учета. Ревизии наиболее формальны из всех технологий гарантии качества. Каждая ревизия начинается с обширных приготовлений и изучений ревизуемой системы и продолжается с помощью интервью и ревизий. Полученная информация проверяется в максимально возможной степени, и формируются заключения. Заключения относительно состояния процесса или изделия документируются в виде формального отчета, который также включает оценку риска при продолжении проекта.

Анхелкар [109] указывает следующие различия между ревизией и другими технологиями гарантии качества:

- Производитель ревизуемого изделия или процесса — обычно коллектив, а не человек.
- Ревизия может быть выполнена без присутствия производителя.
- Ревизии широко используют контрольные списки и интервью и в меньшей степени обзоры.

- Ревизии могут быть внешние, то есть проведенные сторонними к организации ревизорами.
- Ревизия может длиться от дня до недели и даже больше (но всегда строго ограничивается определенными возможностями и показателями).

Возможности ревизии обычно намного более широкие, чем таковые в других операциях гарантии качества. Ревизия признает стратегическую важность информационных технологий для бизнеса и сосредотачивается на управлении информационными технологиями и на бизнес-проблемах реинжиниринга. Ревизуемый проект анализируется с точки зрения всех инвестиций в информационные технологии и его соответствия бизнес-цели.

Ревизия использует целый диапазон организационных планов и финансовых документов, а не только план проекта. Она рассматривает законность привлечения средств и проблемы управления проектом. Типичные показатели — обнаружение и предотвращение мошенничеств и нарушений безопасности, а также планирование восстановления после бедствий и непредвиденных обстоятельств. Особый акцент ставится на оценке эффективности и результативности использования всех проектных ресурсов.

Имеется тесная зависимость между ревизуемым изделием и процессом с одной стороны и более широкой бизнес-политикой и методикой — с другой. Ревизия проекта обеспечивает исходный материал, чтобы гарантировать устойчивость бизнес-политики и методики. Она также может рекомендовать дополнительную политику и методику применительно к проекту.

5.4. Управление изменениями и конфигурацией

Тема управления изменениями и конфигурацией была представлена в разделе 3.4 в контексте возможностей инструментальных средств, поддерживающих эти операции. Во многих случаях сложность управления изменениями и конфигурацией — свидетельство зрелости всего процесса разработки. Как отмечено в начале этой главы, совершенствование управления изменениями и конфигурацией — основной фактор в достижении оптимального уровня зрелости в масштабе CMM (рис. 5.1).

Управление изменениями и конфигурацией — процесс управления изделиями и продуктами, а также управления показателями работы коллектива в разрабатываемой системе ПО. Процесс касается всего жизненного цикла ПО от его начала до завершения. Организации должны планировать изменения. Управление изменениями и конфигурацией уведомляет и усиливает все другие процессы управления, включая управление качеством.

Рис. 5.10 показывает главные концепции и зависимости в управлении изменениями и конфигурацией. Необходимость изменений может привести к изменениям в сценариях использования и в требованиях к ним. Некоторые изменения становятся усовершенствованиями (предложения на будущее) и не рассматриваются в текущей разработке. Имеется соответствие между требованиями к сценариям использования и требованиями тестирования. Требования тестирования можно рассматривать как контрольные точки в сценариях тестирования.

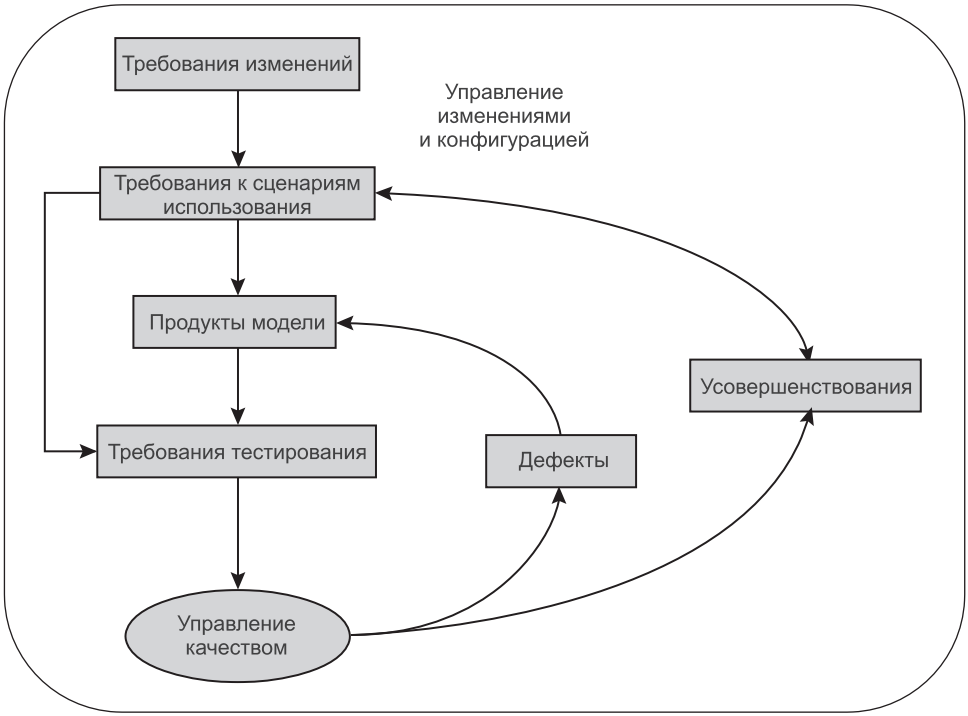


Рис. 5.10. Концепции и зависимости в управлении изменениями и конфигурацией

Моделирование системы приводит к требованиям для сценариев использования создать продукты модели, включая код. Продукты могут иметь много различных версий. Они имеют точки проверки в требованиях тестирования. Некоторые требования тестирования попадают на задачи управления качеством. Это главным образом дефекты, но некоторые могут классифицироваться и как усовершенствования. Перевод недостатков ПО в ранг усовершенствований на самом деле ограничивает возможности системы. Некоторые требования к сценариям использования должны быть отброшены из-за возможностей текущего проекта.

5.4.1. Изменения требований

Действительность проектирования ПО такова, что в течение жизненного цикла разработки ПО требования пользователя изменяются. Некоторые требования исчезают, другие изменяются, наконец, появляются новые. Изменения требований не должны приниматься слишком просто. Они должны быть одобрены в формальном процессе запроса на изменение.

Управление изменениями использует как ручной, так и автоматический процессы. Любое желание сделать изменение в требованиях должно быть формально представлено как запрос на изменение. **Форма запроса на изменение** (Change Request Form — CRF) содержит рекомендации относительно

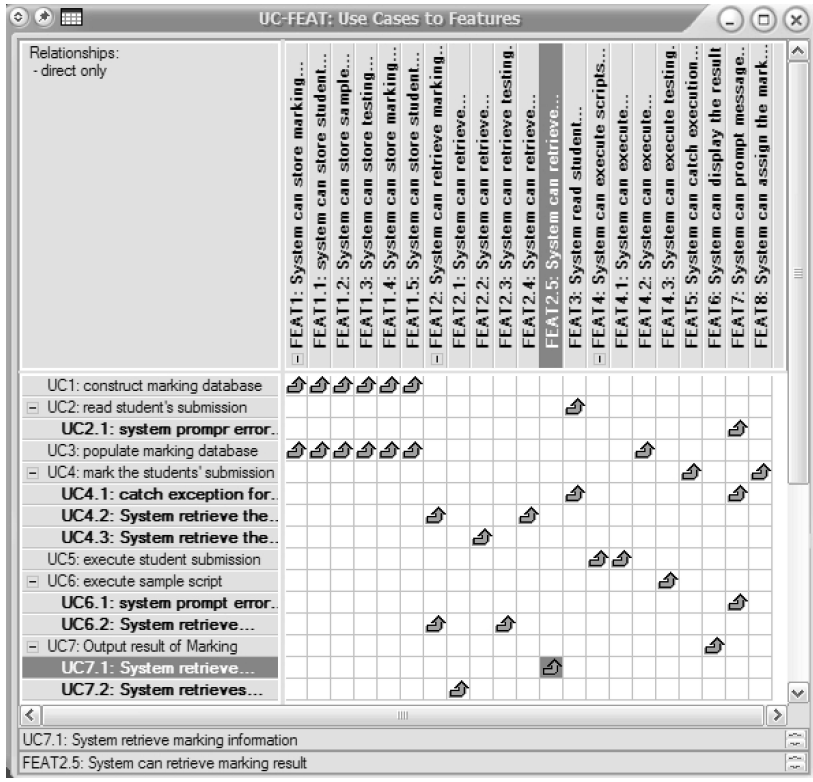


Рис. 5.11. Взаимосвязь между требованиями сценариев использования и требованиями тестирования
Источник: [61], перепечатано с разрешения Pearson Education

предложенного изменения, имя того, кто представляет изменение, дату, проектируемые затраты и другие значения. CRF может быть заполнен вручную или может быть сформирован системой управления изменениями (это произойдет, если изменение — результат предложенного усовершенствования (рис. 5.10)). Вручную заполненная форма должна быть снабжена ключом и зарегистрирована в БД изменений.

Каждый запрос на изменение подвергается тщательной оценке технических и других достоинств. Затем он передается менеджерам проекта для одобрения. В некоторых организациях человек или группа людей назначаются ответственными за принятие решений по запросам изменений. Такая группа имеет звучное имя **специалиста по управлению изменениями** (Change Control Authority — CCA) или **совета по управлению изменениями** (Change Control Board — CCB). Решение определяет стратегические, организационные и технические воздействия. Одобренные изменения завершаются *заказами на изменение разработки* (Engineering Change Orders — ECO) [81]. ECO в свою очередь определяет точные изменения, которые необходимо сделать, любые ограничения и критерии гарантии качества.

Каждое требование к сценарию использования анализируется на предмет, как оно может быть проверено, и с ним связываются одно или несколько требований тестирования (рис. 5.11). Любые изменения в требованиях необходимо связать с изменениями в требованиях тестирования. Все CRF и ECO архивируются для различных дальнейших ссылок, но обычно не бывает никакого управления версиями, связанного с ними (разделы 3.4 и 5.4.2).

5.4.2. Версии продуктов разработки

Разработка завершается производством продуктов модели, включая код. Продукты производят различные члены коллектива разработчиков. Может существовать несколько полезных версий одного и того же продукта. Хранение и управление версиями — область компетенции систем **управления конфигурацией**. Традиционно объектно-ориентированные СУБД [50, 19] являются лучшими в управлении объектами, имеющими разные версии, и многие системы управления конфигурацией основаны на технологии объектных БД.

Рис. 5.12 объясняет принципы, определяющие **управление версиями** [50]. Большинство продуктов разработки хранится как **объекты, обладающие различными версиями**, в **общем рабочем пространстве** (раздел 3.4). Каждый раз, когда разработчик помещает такой объект в **личное рабочее пространство**, создается новая версия. Может быть много версий, полученных из одной версии. Версия, полученная из другой версии, называется ее **производной**. Версии, которые не находятся на одном и том же пути к корню, иногда называются **альтернативами** (или параллельными версиями).

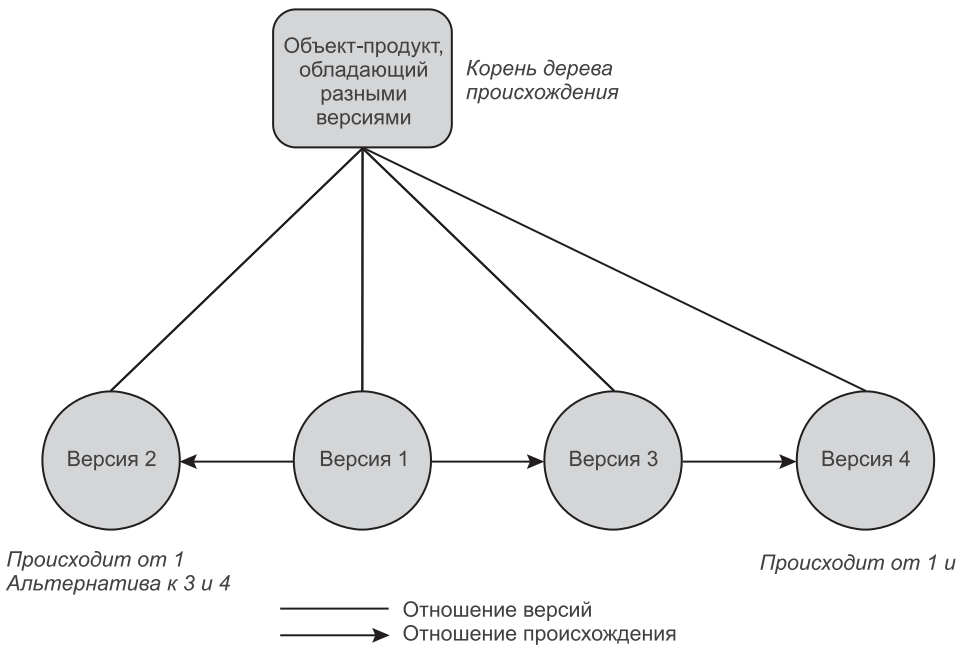


Рис. 5.12. Версии

Версии могут обращаться к объектам различной степени детализации. Продукты часто составлены из других продуктов. Например, класс содержит методы. Поэтому новая версия метода подразумевает новую версию класса. Этот процесс создания новой версии объекта в результате корректировки (а также добавления или удаления) объекта-подмножества называется **перколяцией**.[1]

Выбор некоторого множества взаимно согласованных версий, которое завершается семантически отличающимся продуктом более высокого уровня, называется **конфигурацией**. Технически конфигурация — также версия, которая порой состоит из целого диапазона версий объекта. Часто конфигурация определяет весь программный продукт.

Необходимым условием для надлежащего управления версиями является однозначная *идентификация* объектов, которые обладают разными версиями. Это означает, что в мире объектно-ориентированных БД каждый продукт, обладающий разными версиями, идентифицируется четверкой параметров: {идентификатор БД (database id), идентификатор класса (class id), идентификатор объекта (object id), идентификатор версии (version id)}. Предполагая наличие единственной БД продуктов, нотация `c5o3v7` может означать версию 7 3-го объекта (экземпляра) класса 5.

Рис. 5.13 — пример конфигурации, определенной на версиях трех объектов [19]. Конфигурация `c7o1v1` сама является версией продукта, имеющего версии. Обратите внимание, что альтернативы обычно не рассматриваются

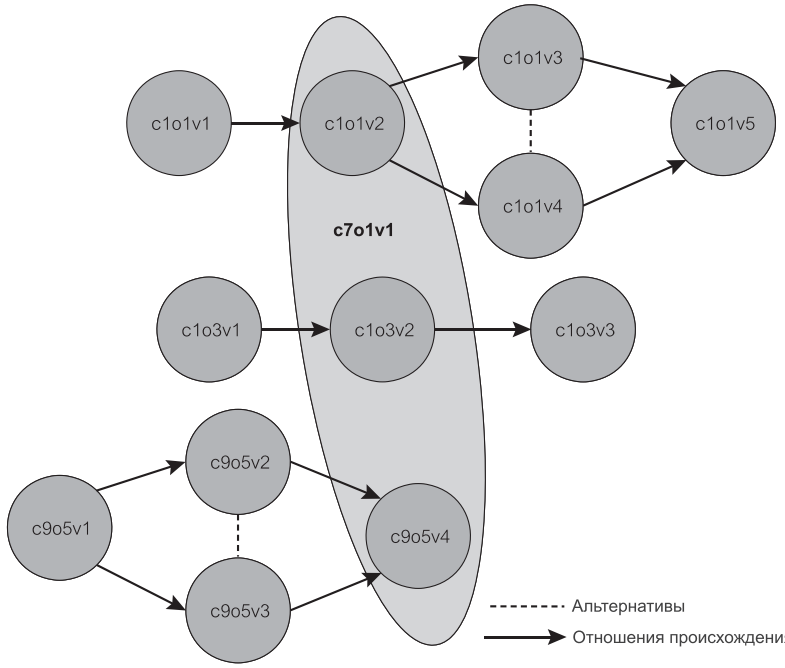


Рис. 5.13. Конфигурация

[1] Перколяция — «образование связанных объектов в неупорядоченных средах». Тарасевич Ю. Ю. Перколяция: теория, приложения, алгоритмы, Едиториал УРСС, 2002. (*Прим. перев.*)

как хорошие кандидаты на конфигурацию. Они должны быть **объединены** разработчиками (с помощью инструментального средства) в согласованную версию, прежде чем будут рассматриваться для использования в конфигурации. Возможно, что сюда могут быть включены версии различных объектов одного и того же класса. Конфигурация на рис. 5.13 имеет две версии класса `c1` (`c1o1v2` и `c1o3v2`).

Конфигурация в общем рабочем пространстве может быть зарегистрирована в качестве текущих (самых последних) **исходных данных**. Исходные данные определяют версии продукта, которые разработчики могут импортировать в личное рабочее пространство и работать с ними. Конфигурация исходных данных заморожена для изменений в общем рабочем пространстве, однако новая может быть *выдвинута* в качестве исходных данных. Более ранние конфигурации сохраняются в БД конфигураций с тем, чтобы старшие версии объектов могли использоваться в новых конфигурациях, включая новые исходные данные.

С точки зрения разработчика исходные данные, распределенные между клиентами, называются **версией** системы (или **подлежащим сдаче продуктом**). Однако с точки зрения управления версия — больше чем ПО на основе исходных данных. Версия системы — сложный логический пример с внешними отношениями, который должен учитывать разнообразие клиентов и текущие настройки (инсталляции). Например, новые версии должны быть инсталлируемы на любых предыдущих версиях или на пустом месте. Должна быть выбрана среда распространения для ПО и документации (это, вероятно, будет Интернет). Должны также быть разработаны различные способы оплаты. Оценки версий ПО могут понадобиться, чтобы сделать их доступными для загрузки.

Рассмотрев все это, можно сделать вывод, что выпуск системы включает существенное число изделий и процессов [96]:

- исполняемый код клиента;
- код сервера БД (структура БД, триггеры, хранимые процедуры);
- загруженные данные БД (если они есть) и любые другие файлы данных;
- файлы конфигурации и инструкции инсталляции;
- документация системы и диалоговая справочная система;
- условия распространения и связанная с этим информационная и бухгалтерская документация.

5.4.3. Дефекты и усовершенствования

Дефект ПО — это ошибка в ПО, обнаруженная в процессе формальных просмотров или испытаний, либо обнаруженная пользователями в эксплуатируемой системе. **Усовершенствование** — рекомендуемая особенность системы, которая может быть реализована в будущем. Усовершенствования поступают из двух главных источников. Они могут быть официально представлены заинтересованными сторонами как формальное требование изменения, не имеющего никакого отношения к какому-то дефекту. С другой стороны, они могут быть некоторыми дефектами, которые связаны с неполной или отсутствующей

реализацией требования пользователя. Такие дефекты могут иногда откладываться для более позднего решения как усовершенствование системы.

Дефекты и усовершенствования изменяют состояние системы в результате действия на нее. История изменений состояния фиксируется системой управления изменениями. Эта история заменяет потребность в управлении версиями и конфигурацией на использование дефектов и усовершенствований. Дефекты и усовершенствования — независимые объекты; они не являются композицией других дефектов или усовершенствований. Кроме разрешения доступа к истории дефектов и усовершенствований, инструментальное средство управления изменениями позволяет их сгруппировать в соответствии с различными критериями, типа владельца, приоритета, симптомов (раздел 3.4.1).

Рис. 5.14 показывает пример *определения дефекта*. Каждый дефект нумеруется (ID), он находится в конкретном состоянии (State). В рассматриваемом примере дефект — `Submitted` (подчиненное), ему дают заголовок (Headline) и детальное описание (Description), он принадлежит проекту (Suite Project и UCM Project), он имеет приоритет (Priority), степень серьезности (Severity), владельца (Owner), ключевые слова (Keywords) и симптомы (Symptoms). Для каждого состояния дефекта имеется множество возможных действий. Множество действий настраиваемо согласно стратегии управления изменениями для дефектов, имеющихся в проекте.

Рис. 5.15 показывает пример *определения усовершенствования*. Подобно дефектам каждое усовершенствование нумеруется, оно находится в определенном состоянии, в рассматриваемом примере — `Opened` (открыто), ему дают заголовок и детальное описание, оно принадлежит проекту, оно имеет

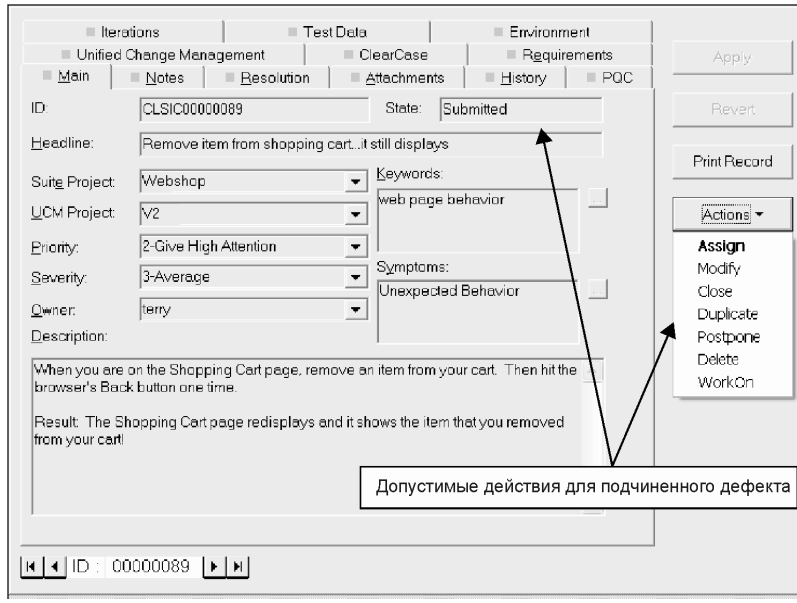


Рис. 5.14. Дефект

Источник: образ экрана *IBM Rational Suite*, перепечатано с разрешения *Rational Suite Tutorial*

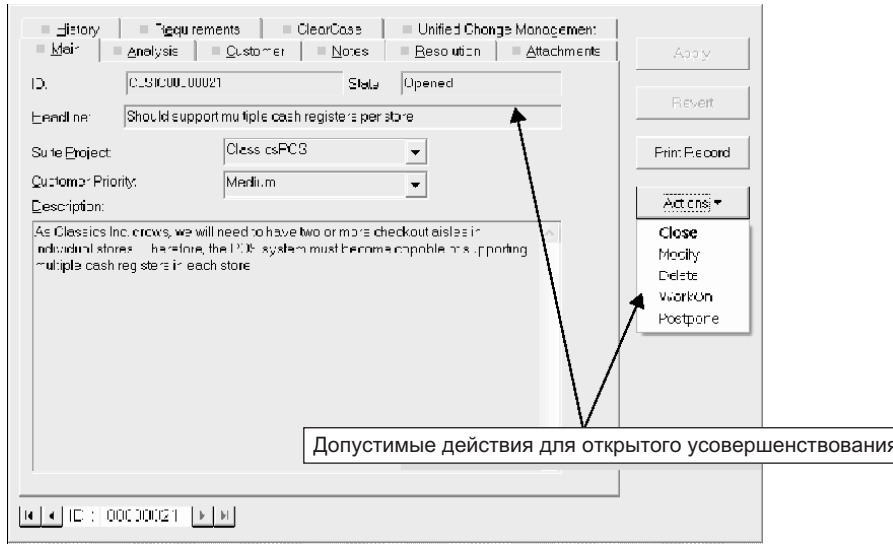


Рис. 5.15. Усовершенствование
Источник: образ экрана *IBM Rational Suite*, перепечатано с разрешения *Rational Suite Tutorial*

приоритет клиента. Для каждого состояния усовершенствования имеется множество возможных действий. Множества действий настраиваются согласно стратегии управления изменениями для усовершенствования, имеющегося в проекте.

Web-сайт программных средств с открытым кодом для GCC-системы отслеживания ошибок Bugzilla [16] рекомендует следующие *состояния для дефектов*:

- `new` (новое) — когда дефект представляет собой новый вход в инструментальное средство, и ранее о нем ничего не говорилось;
- `resolved` (решенное) — дефект был устранен, но он может все еще повторно открыться позже, если решение окажется неудовлетворительным;
- `unconfirmed` (неподтвержденное) — дефект был отмечен, но все же должно быть проверено, является ли это дефектом вообще;
- `reopened` (повторно открытое) — дефект был повторно открыт, потому что предыдущая позиция на самом деле не работала;
- `assigned` (назначенное) — дефект еще не устранен, но был назначен в соответствующую группу;
- `suspended` (приостановленное) — работа с дефектом была приостановлена, поскольку требуется времени больше, чем запланировано, или считается неэффективной по затратам;
- `waiting` (ожидающее) — требуется больше информации, чтобы должным образом идентифицировать дефект;
- `closed` (закрытое) — вопрос закрыт в результате его решения или другим организационным действием.

В Bugzilla *усовершенствование* — просто дефект с очень низкой строгостью и поэтому рассматриваемое по принципу «хорошо, если есть особенность». В целом типичные *уровни строгости для дефектов* следующие:

- `trivial` (тривиальный) — дефект может быть тривиально зафиксирован;
- `minor` (незначительный) — хорошо бы зафиксировать, иначе жить будет несколько некомфортно;
- `normal` (нормальный) — фиксация его вовремя, наличие дефекта приводит к некоторой потере функциональных возможностей;
- `major` (главный) — дефект воздействует на главные функциональные возможности, но однако не приводит к порче ПО;
- `critical` (критический) — ничего себе, как эта вещь вообще оказалась здесь? Однако никто не говорит о полном крушении;
- `blocker` (блокирующий) — ПО даже не будет запускаться или его невозможно устранить.

Когда дефект проанализирован, решение о нем может иметь много различных форм. Снова следующий список — только рекомендация; он может отличаться у разных продавцов и может быть настроен менеджерами проекта [16]:

- `invalid` (недействительный) — этот дефект не может быть воспроизведен или больше не имеет силу для текущей версии ПО, либо это даже не дефект вообще;
- `duplicate` (дубликат) — были сообщены некоторые другие дефекты, подобные этому;
- `workaround` (обходной путь) — дефект подтвержден и будет зафиксирован когда-нибудь позже, но пока есть обходной путь;
- `fixed` (зафиксированный) — проблемы с дефектом были решены, что кончается *решенным* состоянием для дефекта;
- `worksforme` (рабочая форма) — дефект не воспроизводим в текущей инсталляции, что завершается *неподтвержденным* состоянием;
- `wontfix` (не фиксируемый) — дефект не собираются устранять из-за технических или других причин.

5.4.4. Метрики

Тема **метрик** (системы показателей) была представлена в главе 4 в контексте управления проектом. Планирование проекта и составление бюджета основаны на предшествующем опыте проектирования и на различных параметрах (метриках) изделий и процессов, собранных в прошлом. Например, метрики, которые оценивают размер ПО, типа числа строк кода или числа функциональных единиц в коде, являются обязательными в оценке стоимости проекта. Сбор метрик, подобно тестированию, является операцией, имеющей отношение ко всем стадиям жизненного цикла.

Метрики ПО — большая независимая область знаний и такая, которая вызывает много противоречий. Главное противоречие касается самого характера метрик — собрание количественных показателей, которые обеспечивают

доступ к таким субъективным проблемам, как удобство сопровождения, масштабируемость, сложность или даже размер кода или размер трудозатрат. Однако даже приблизительные, неоднозначные и примерные числа лучше, чем ничего. Кроме того, очевидно, что они также хорошо обеспечивают решения в управлении и планировании.

Другое важное противоречие касается того факта, что метрики имеют побочный эффект оценки людей, а не только изделий и процессов. Как наблюдал Гецци и др. [34]: «Фактически ясно, что какое бы количество мы ни взяли для оценки какого-либо качества, проектировщик будет пытаться максимизировать (или минимизировать) измеренное количество, а не реальное качество».

Вместо того чтобы дать базовое введение в метрики, в этом разделе представлен ряд наиболее употребительных метрик для разработки объектно-ориентированного ПО. Известные как **CK-метрики** по именам их создателей Шидамбера и Кемерера (Chidamber и Kemerer) [21], приведенные здесь метрики тоже не избежали определенной степени противоречий [38]. Однако они совместимы с подходами, принятыми далее в книге, и поэтому объясняются здесь.

Имеется шесть CK-метрик [21]:

- взвешенные методы на класс (weighted methods per class — WMC);
- глубина дерева наследования (depth of inheritance tree — DIT);
- число детей класса (number of children — NOC);
- связывание классов объектов (coupling between object classes — CBO);
- отклик класса (response for a class — RFC);
- недостаток связанности в методах (lack of cohesion in methods — LCOM).

Взвешенные методы на класс (WMC) вычисляются как сумма сложностей всех методов в классе. Сложности нормализованы значениями в диапазоне от 0 до 1. Точный смысл понятия сложности метода не определен. Это на усмотрение тех, кто использует метрику, однако предполагается, что могут использоваться некоторые структурные свойства методов.

Шидамбер и Кемерер [21] предлагают следующие точки зрения на WMC-метрику:

- WMC — это *метрика-предсказатель* (в противоположность *метрике управления*). Метрика может служить как предсказатель времени и трудозатрат, необходимых, чтобы разработать и поддерживать класс.
- Большое значение WMC создает увеличенную потенциальную зависимость наследования, если класс многократно используется как надкласс.
- Большое значение WMC является вероятным индикатором того, что класс является специфическим для приложений, таким образом ограничивая возможность его многократного использования многими приложениями.

Иерархия наследования имеет корень (суперкласс) и листья (подклассы). **Глубина дерева наследования** (DIT) — максимальная длина (число классов) от узла до корня дерева. DIT применяется к классу, а не к дереву наследования или к системе. В тривиальном случае определение применяется к оди-

ночному наследованию, но может применяться и в случае множественного наследования.

Точки зрения на DIT [21]:

- Класс с большим значением DIT наследует большое количество методов, и поэтому более трудно понять его поведение.
- Для существенного числа классов большое значение DIT подразумевает большую сложность проекта в смысле общего числа классов и методов в системе.
- Класс с большим значением DIT, вероятно, будет иметь преимущество повторного использования унаследованных методов.

Подобно DIT **число детей класса** (NOC) — метрика, связанная с наследованием. NOC считает число непосредственных подклассов рассматриваемого класса. Большое значение NOC — скорее отрицательное свойство, согласно следующему списку точек зрения [21].

- Большее значение NOC означает большее количество повторных использований через наследование (что может быть и хорошо, и плохо в зависимости от принятой стратегии).
- Большое значение NOC обычно подразумевает неподходящую или расплывчатую абстракцию суперкласса и неправильное использование формирования подклассов.
- Класс с большим значением NOC, вероятно, будет делать слишком много (беря на себя слишком много ответственности в системе). Это создает несбалансированный проект и делает тестирование более трудным.

Два класса *связаны*, если они зависят друг от друга (но зависимость, а поэтому и связывание, могут иногда быть только в одном направлении). Зависимость может быть структурной (через переменные экземпляра) или поведенческой (благодаря зависимостям методов). **Связь между классами объектов** (CBO) — это число классов, соединенных с данным классом. Связь необходима для сотрудничающих классов, однако чрезмерное использование связей имеет существенные нежелательные эффекты.

Точки зрения Шидамбера и Кемерера [21] на CBO:

- Чрезмерное значение CBO вредно для качества структурного проектирования.
- Большое значение CBO для класса предотвращает его повторное использование, потому что класс зависит от других классов. (Это истинно для повторного использования наследования, но можно также считать, что сама связь *является* повторным использованием.)
- Большое значение CBO препятствует инкапсуляции класса и делает его сопровождение более неприятным.
- Проект с чрезмерным значением CBO создает дополнительные проблемы для тестирования.

Отклик класса (RFC) — число методов, чьи услуги потенциально необходимы, чтобы обработать требования сообщений, полученных объектом класса. Другими словами, RFC — число обращений к методу для обеспечения обслуживания. Методы могут быть внутренние к классу и/или внешние.

Слово «потенциально» подразумевает возможность самого плохого случая, то есть, максимальное число методов, вовлеченных в выполнение работы (практически для любого отдельного требования некоторые методы в цепи вызовов могут быть не нужны и не будут использоваться).

Шидамбер и Кемерер [21] предлагают следующие точки зрения на RFC-метрику:

- Тестирование и отладка класса более сложны при большей величине RFC.
- Большое значение RFC подразумевает сложный класс.
- Будучи величиной самого плохого случая, RFC может быть хорошим индикатором времени, необходимого, чтобы исполнить тестирование.

Недостаток связанности методов (LCOM) — заключительная CK-метрика. Большая величина связанности методов говорит, что методы класса не походят друг на друга (не подобны), и поэтому они, вероятно, не принадлежат общей абстракции (единому классу). Подобие может быть измерено множеством методов, обращающихся к тем же самым переменным экземпляра. Чем больше число подобных методов, тем более связный класс. Полное отсутствие связанности означает, что методы полностью независимы.

Точки зрения Шидамбера и Кемерера [21] на LCOM:

- Существование связанности методов улучшает инкапсуляцию и поэтому желательно.
- Недостаток связанности означает, что методы независимы, и класс следует разбить на два или более класса.
- Низкое значение LCOM увеличивает сложность проекта и в потенциале приводит к ошибкам.

Метрики ПО предложены как количественные меры оценки качества программных продуктов и процессов. Несмотря на их важность, реальность такова, что метриками часто пренебрегают в проектах ПО. Такие проекты автоматически находятся на низком уровне зрелости, но это не утешение.

Резюме

1. *Модель развития функциональных возможностей* (Capability Maturity Model — CMM) определяет пять уровней зрелости процесса: 1) начальный, 2) однотипный, 3) определяемый, 4) управляемый и 5) оптимизирующий. Эта глава сконцентрирована на основных факторах, определяющих вышеприведенные уровни зрелости процесса, а именно: 1) управление людьми, 2) управление рисками, 3) управление качеством и 4) управление изменениями и конфигурацией.
2. Люди в *сетевой организации* имеют уникальные представления о мотивации, лояльности, организационной силе, лидерстве, достижениях, удовлетворении, вознаграждении и т. д. Сетевые организации требуют новых подходов к управлению людьми, в особенности в области привлечения и мотивации людей.
3. Привлечение и мотивация людей — сущность процесса *построения коллектива*. Имеется много *теорий мотивации*, которые пробуют выявить

то, что побуждает людей в их действиях, и какие факторы влияют на профессиональное поведение людей.

4. Для успешного выполнения проекта должны быть установлены надлежащие *формы и линии связи*. И слишком большая, и слишком малая связь может повредить проекту. Кроме информационных и организационных аспектов связь является первичным средством разрешения конфликтов в коллективах.
5. *Коллективы* проходят четыре стадии *создания*: 1) формирование, 2) волнение, 3) нормализация и 4) выполнение.
6. *Управление рисками* включает три стадии: 1) идентификацию рисков, 2) оценку рисков и 3) обработку рисков. Стратегии управления рисками находятся в диапазоне от реактивных до упреждающих. *Реактивные стратегии рисков* задерживают обработку рисков, пока они не станут критическими факторами в проекте. *Упреждающие стратегии рисков* поощряют воздействие на риски, как только они возникают или даже в ожидании их возникновения.
7. *Управление качеством ПО* — всеобъемлющая операция, которая переплетается с большинством других операций управления и охватывает весь жизненный цикл разработки и ее процесс. Имеется много желаемых показателей качества ПО, и в зависимости от проекта ПО некоторые из них могут быть более важны, чем другие. Показатели качества ПО должны быть идентифицированы, определены и классифицированы.
8. Управление качеством состоит из гарантии качества и контроля качества. *Контроль качества* главным образом связан с тестированием качества изделия в противоположность *гарантии качества*, которая занимается обеспечением качества изделия. Контроль качества имеет реактивный характер. Гарантия качества — принципиально упреждающее действие.
9. *Тестирование ПО* — доминирующая операция контроля качества. Имеется много возможных *технологий тестирования*, которые можно комбинировать, чтобы обеспечить лучший объем тестирования и результаты. Независимо от того, насколько обширно тестирование, оно не может быть исчерпывающим и гарантировать корректность программы/системы. *План испытаний* определяет график тестирования, бюджет, задачи (контрольные примеры) и ресурсы. План не должен быть ограничен только тестированием кода. Он должен включать тестирование других продуктов проекта.
10. Группа управления-разработки *гарантий качества ПО* (Software Quality Assurance — SQA) обеспечивает гарантию качества. Гарантия качества включает: 1) контрольные списки, 2) обзоры и 3) ревизии.
11. *Управление изменениями и конфигурацией* — процесс управления изделиями и продуктами процесса и управления операциями коллективной работы в системе создания ПО. Процесс относится ко всему жизненному циклу ПО, от его начала до завершения.
12. *Управление изменениями* включает ручные и автоматические процессы. Предложенные изменения формируются как *запросы на изменение*. За-

просы на изменение могут завершиться изменениями в сценариях использования и требованиях сценариев использования. Некоторые изменения станут усовершенствованиями и не рассматриваются в текущей разработке. Имеется соответствие между требованиями сценария использования и требованиями тестирования.

13. Разработка завершается созданием продуктов модели, включая код. Может быть много полезных версий одного и того же продукта. Хранение и управление версиями — область систем *управления конфигурацией*. Выбор некоторого множества взаимно согласованных версий, которое завершается семантически отличающимся продуктом более высокого уровня, называется *конфигурацией*.
14. *Дефект* ПО — ошибка в ПО, обнаруженная во время формальных обзоров или испытаний, либо обнаруженная пользователями в эксплуатируемой системе. *Усовершенствование* — рекомендуемая особенность системы, которая может быть реализована в будущем. Дефекты и усовершенствования могут менять свои состояния в результате воздействий на них.
15. Набор *метрик*, подобно тестированию, является операцией, имеющей отношение ко всем стадиям жизненного цикла. СК-метрики — наиболее известные метрики для разработки объектно-ориентированного ПО.

Ключевые термины

Обзорные вопросы

1. Объясните уровни зрелости процесса CMM. Каковы основные факторы для улучшения уровней зрелости?
2. Что такое сетевая организация? Каковы главные проблемы управления людьми в пределах сетевой организации? Свяжите ваши объяснения с процессами создания коллектива и с подходящими мотивационными теориями.
3. Обсудите взаимодействие между формами связи и линиями связи.
4. Обсудите взаимодействие между формами связи и разрешением конфликтов.
5. Насколько полезны реактивные стратегии в управлении рисками? Объясните.
6. Как подсчитывается подверженность риску? Что такое контрольная точка риска?
7. Каковы стратегии обработки рисков в упреждающем управлении рисками? Как они применяются? Приведите пример.
8. Объясните различия между гарантией качества и контролем качества. Какие главные технологии управления качеством используются в контроле качеством и в гарантии качества?
9. Объясните качество возможности сопровождения.
10. Считают, что тестирование «черного ящика» и «белого ящика» являются дополнением друг другу. Объясните.
11. Каково отношение между регрессионными тестами и испытательными тестами? Могут ли они использоваться совместно? Объясните.
12. Объясните концепции контрольного примера, сценария тестирования и тестового набора. Как эти концепции связаны?
13. Как вы понимаете культуру качества? Сравните, как операции контроля качества и гарантии качества вносят вклад в культуру качества.
14. Обратимся к рис. 5.10. Говорят, что изображение стоит тысячи слов. Объясните изображение своими словами (никакая тысяча слов не потребуется).

15. Версии продуктов моделирования должны быть объединены. Что это означает? Как это делается? В какие моменты выполнения проекта необходимо выполнять слияние? Являются ли процессы слияния автоматическими или необходимо ручное вмешательство?
16. Обратимся к рис. 5.14. Предположим, что дефект не в состоянии `Submitted` (подчиненное), а в состоянии `Resolved` (решенное). Какие действия вы позволили бы в выпадающем списке для дефекта в состоянии `Resolved`? Объясните.
17. Обратимся к рис. 5.15. Предположим, что усовершенствование не в состоянии `Opened` (открытое), а в состоянии `Postponed` (отложенное). Какие действия вы позволили бы в выпадающем списке для усовершенствования в состоянии `Postponed`? Объясните.
18. Обратимся к CK-метрикам. Какие из этих метрик имеют особую важность для определения качества возможности сопровождения системы? Объясните.

Часть 2

От требований через структурное проектирование к готовому программному обеспечению

Современные процессы создания ПО *итеративны*. Производство ПО выполняется от итерации к итерации. *Итерация* — цикл разработки ПО со своим собственным анализом требований, проектированием системы, реализацией, тестированием и поставкой пользователям. Итерации являются *пошаговыми* — последовательные итерации поставляют все более качественное изделие, расширяя и улучшая предыдущую итерацию. Каждая итерация производит выполнимую, проверенную и интегрированную *версию* ПО. Итерации имеют короткие продолжительности (несколько недель) для того, чтобы поддерживать доверие заинтересованных сторон к разработке ПО и более оперативно получать обратную связь от пользователя для последующих итераций.

Эта книга представляет принципы разработки информационных систем, предлагая при этом итерации, которые являются пошаговыми версиями рассмотрения учебного примера. Предложены три итерации, которым посвящены части 2, 3 и 4 книги. Каждая итерация имеет свой собственный акцент и требует соответствующих знаний и набора навыков. Получение основ знаний обеспечивается таким образом, что оно переплетается с рассмотрением итеративных задач и решений.

Учебная сторона книги состоит в том, чтобы связать объяснение с учебным примером и с небольшими проектами. Если, однако, требуемые основы знаний достаточно сложны, даются более короткие примеры (не обязательно полученные из учебного примера) для объяснения основных концепций. Учебный пример, решения задач, небольшие проекты и большинство примеров используют компанию по имени AEM, которая специализируется в *оценке расходов на рекламу* (Advertising Expenditure Measurement).

Часть 2 книги определяет модель бизнес-объектов для AEM (глава 6). Модель выделяет главные области AEM. Предметная область, рассматриваемая в книге, называется *управлением деловыми партнерами* (Contact Management — CM). Объектная модель предметной области для CM дается в главе 7. Остальные главы части 2 относятся к подсистеме CM, называемой *управлением электронной почтой* (Email Management — EM), которая является предметом учебного примера книги.

EM-подсистема помогает управлять документами электронной почты. Учебный пример сосредоточен на составлении и передаче сообщений по электронной почте (называемых исходящими сообщениями), размещенных предварительно в БД и намеченных для рассылки деловым партнерам (клиентам, поставщикам, служащим организаций и т. д.). Исходящие сообщения — это вопросы к деловым партнерам, просьбы о встречах и т. д. — вводятся в БД в процессе проведения ежедневных бизнес-транзакций. Окончательный EM-продукт может включать сбор, индексацию, классификацию и исправление всей деловой корреспонденции электронной почты. И исходящие, и входящие сообщения могут размещаться и управляться в БД EM.

Для управления содержанием сообщений, которые, по сути, являются неструктурированными документами электронной почты, им следует придать соответствующую структуру. На начальных итерациях учебного примера структура весьма проста — она задает тему исходящего сообщения, текст сообщения, дату сообщения, отправителя и получателя. Текст исходящего сообщения — недифференцированный поток байтов, причем ничего не известно относительно его внутренней структуры (это отражает ограничения реляционных БД по отношению к обработке типов данных мультимедиа).

Учебный пример рассматривает возможность системы посылать сообщения по электронной почте и корректировки БД, чтобы указать, какие исходящие сообщения и прилагаемые документы были отосланы. Основные акторы ЕМ (используя язык UML — см. раздел 2.2.2) — служащие организации. В учебном примере отправляются только формальные деловые сообщения АЕМ-партнерам, сведения о которых находятся в памяти. Неофициальные (частные) сообщения могут использовать те же почтовые серверы, но они не интересны для учебного примера.

Итерация 1 (рассмотренная в этой части) основана на простом консольном Java-приложении, которое обеспечивает доступ к БД через JDBC — см. раздел 12.1.3. В ответ на регистрационное имя, посланное к БД электронной почты, приложение может извлечь из БД и показать список исходящих сообщений, которые могут быть отправлены деловым партнерам. После этого пользователь (служащий) может запросить автоматическую отправку выбранного исходящего сообщения. Когда сообщение будет успешно отправлено, БД соответствующим образом обновляется.

Последующие итерации учебного примера адресованы к более сложным средствам управления сообщениями электронной почты, содержащими текст и добавленные к сообщению неструктурированные (мультимедиа) документы, которые должны быть посланы. *Итерация 2* (рассмотренная в части 3) позволяет вводить новые исходящие сообщения в БД через GUI или другой допустимый в Web-технологии интерфейс. Она поддерживает также рассмотрение исходящих сообщений, основанное на различных фильтрах и критериях поиска. *Итерация 3* (рассмотренная в части 4) перемещает основной акцент логики приложений на БД и задает дополнительные бизнес-правила в БД, которым должно подчиняться приложение. Примеры задач в конце каждой главы расширяют учебный пример. Рассылка электронных сообщений — связанная с основной работой операция, нацеленная на деловых партнеров АЕМ (клиентов, поставщиков и т. д.). Это дает непосредственную возможность отслеживать события, связанные с деловыми партнерами, и управлять временными протоколами служащих.

Основные учебные цели части 2 и итерации 1 учебного примера — получение знаний в следующих областях:

- размещение проекта информационной системы внутри модели бизнес-объектов и внутри объектной модели предметной области;
- разработка ПО, которая вытекает из документации сценариев использования, концептуальной модели классов и дополнительных спецификаций;
- значение структурной основы в итеративной и пошаговой разработке;
- проектирование и программирование БД для обеспечения информационных потребностей прикладных программ;
- проектирование классов и проектирование взаимодействий между объектами;
- доступ к БД и кэширование объектов сущностей в Java;
- взаимозависимость между программированием и тестированием, разработка через тестирование и приемочные испытания;
- разработка понятного, ремонтопригодного и расширяемого ПО, которое является строгим и задокументированным, но все же сдержанным с точки зрения традиций и бюрократизма.

Глава

6 Модель бизнес-объектов

Следуя **унифицированному процессу** (Unified Process — UP) [53, 85], мы будем различать **модель бизнес-объектов** (Business Object Model — BOM), и **объектную модель предметной области** (Domain Object Model — DOM) — называемые в RUP моделью предметной области. Обе являются объектно-ориентированными моделями системы высокого уровня (то есть высоко абстрагированными), но BOM относится ко всему бизнесу, в то время как DOM относится к рассматриваемой предметной области. BOM — широкая модель предприятия. DOM — концептуальная модель интересующей предметной области. В этом учебнике BOM описывает основы и контекст для изучения примера и обсуждается только в этой главе. Остальные главы учебного примера в частях со второй по четвертую определяют DOM или используют ее.

BOM состоит из модели бизнес-сценариев использования и модели бизнес-классов. **Модель бизнес-сценариев использования** представляет взаимодействия *бизнес-акторов* с бизнес-сценариями использования и показывает основные отношения между бизнес-сценариями использования. Бизнес-сценарии использования — бизнес-процессы большой степени детализации — главные бизнес-функции предприятия. **Модель бизнес-классов** представляет *бизнес-сущности* и отношения между ними. Бизнес-сущности — наиболее фундаментальные классы («бизнес-объекты»), необходимые для реализации бизнес-сценариев использования (*класс* используется здесь в обычном объектно-ориентированном значении как абстракция, которая описывает множество объектов с одинаковыми атрибутами, операциями, отношениями и семантическими ограничениями).

Упрощенный вариант модели бизнес-сценариев использования, называемый **диаграммой бизнес-контекста**, применяется, чтобы изобразить возможности основного бизнеса предприятия. Контекстная диаграмма — старая технология, используемая в структурном моделировании, чтобы очертить возможности системы. В структурном моделировании контекстные диаграммы были составными частями когда-то очень популярной и успешной практики моделирования, называемой диаграммами потока данных (Data Flow Diagram — DFD) (раздел 2.1.1). Хотя DFD и не поддерживается в UML, контекстные диаграммы остаются популярным вариантом моделирования в объектно-ориентированном анализе (например, [55]).

Модели и диаграммы — примеры продуктов разработки ПО. **Продукт** — любая задокументированная информация, произведенная разработчиком сис-

темы и используемая в жизненном цикле разработки. Один из наиболее важных продуктов разработки информационной системы (Information System — IS), работа над которым должна начаться как можно раньше, — это *глоссарий* терминов и определений. Главная цель глоссария состоит в том, чтобы обеспечить ясность в отношениях между разработчиками и другими заинтересованными сторонами проекта. Без глоссария следующее анонимное высказывание часто будет циркулировать в проекте: «Я знаю, что Вы полагаете, что Вы поняли то, что Вы думаете, что я сказал, но я не уверен, что Вы сделали то, что Вы слышали, и не то, что я подразумевал».

Подобно всем операциям моделирования систем, BOM использует диаграммы, чтобы визуализировать основные концепции. Хотя визуальное представление и обязательно для рассуждения относительно моделей, спецификации систем ПО состоят в основном из текстовых документов. Диаграммы обслуживают скорее цели объединения — они особенно полезны при категоризации и классификации концепций моделирования и выражают отношения и зависимости между концепциями моделирования. BOM устанавливает структуру для стратегии разработки информационной системы организации и представляет начальную терминологию для связи клиента и разработчика.

6.1. Advertising Expenditure Measurement, ее бизнес

AEM (**Advertising Expenditure Measurement** — оценка расходов на рекламу) — *компания исследования рынка*. AEM не производит товары в обычном смысле слова. Ее основной бизнес — получение, обработка и продажа информации относительно рекламных объявлений, которые появились в различных средствах информации. Основа клиентов AEM состоит из организаций и индивидуумов, которые покупают информацию, касающуюся исследования рынка относительно рекламных расходов.

Процесс определения рекламных объявлений и записи содержания рекламы частично является автоматическим и частично — ручным. *Автоматический сбор данных* включает современные средства просмотра и представление в мультимедиа аппаратного и программного обеспечения. *Ручной сбор данных* — утомительный процесс наблюдения рекламных TV объявлений, чтения газет и журналов и т. д. Рекламные объявления вводятся в БД AEM как новые (не рекламируемые прежде), или они соответствуют рекламе, уже размещенной в БД. Соответствие поступающих рекламных объявлений тем, которые уже введены в БД, проверяется во время сбора данных.

Содержание всех рекламных объявлений размещается в БД (оно включает рекламируемые изделия, продолжительность — для TV и радио, размер — для прессы, денежные величины и т. д.). Процесс идентификации рекламного содержания подвержен ошибкам и требует частой внешней и внутренней проверки. *Внешняя проверка* требует контакта с организациями-распространителями рекламы, рекламными агентствами, индивидуальными рекламодателями и т. д. *Внутренняя проверка* — механизм контроля качества, который должен гарантировать, что сбор данных AEM-системой и служащими правилен. AEM имеет договорные соглашения со многими информационными

агентствами, чтобы регулярно получать от них *электронные регистрационные файлы* с информацией, имеющей отношение к рекламному содержанию этих материалов, что очень облегчает сбор данных и процессы проверки.

Рекламные данные обеспечивают две области сообщений AEM-клиентам. Клиент может запросить сообщение, появились ли, как предполагается, рекламные объявления, которые он оплатил (это называется *мониторингом компании*). Клиент может также запросить сообщение, обрисовывающее конкурентоспособное рекламное положение в конкретной области промышленности (это называется *сообщением о расходах*). Сообщения о расходах охватывают расходы, которые тратит рекламодатель или рекламируемое изделие в соответствии с различными критериями (время, географическое положение, средства информации и т. д.).

Любой AEM-клиент (не только рекламный клиент) может купить *сообщения о расходах* — в форме заказного программного продукта или твердой копии. Основные клиенты AEM включают индивидуальных рекламодателей, рекламные агентства, информационные компании, консультантов по покупке средств информации, а также специалистов по продажам и маркетингу, проектировщиков средств информации, покупателей и т. д.

6.2. Диаграмма бизнес-контекста

Описания основного бизнеса AEM в разделе 1.1 достаточно, чтобы разработать диаграмму бизнес-контекста для AEM. **Диаграмма бизнес-контекста** — версия модели бизнес-сценария использования, настроенной на задачи. **Бизнес-акторы** представляют **внешние сущности**, то есть людей, организации, информационные системы и т. д., которые являются внешними по отношению к AEM. Внешние сущности действуют как поставщики информации в AEM или получатели информации от AEM, либо являются и тем, и другим. Внешние сущности сгруппированы, чтобы идентифицировать главные категории поставщиков/получателей информации, а не индивидуальных участников системы.

В диаграмме бизнес-контекста имеется только один **бизнес-сценарий использования**. Он изображает все «поведение системы», представляющее основную бизнес-деятельность, и отражает возможности системы. Бизнес-сценарий использования получает информацию от внешних сущностей, обрабатывает ее некоторым образом и посылает выходную информацию внешним сущностям. Цель бизнес-сценария использования — преобразовать входную информацию в выходную. Все остальное — вне его целей. Бизнес-сценарий использования не будет обрабатывать незапрашиваемую входную информацию и не будет производить ненужную выходную информацию.

Взаимодействия между внешними сущностями и бизнес-сценарием использования отличаются от взаимодействий в обычной диаграмме сценариев использования. Они представляют поток входной/выходной информации (**потоки данных**) между внешними сущностями и бизнес-сценарием использования.

Рис. 6.1 — диаграмма бизнес-контекста, которая изображает возможности AEM. Имеются две внешние сущности: `Media Outlet` (информационное

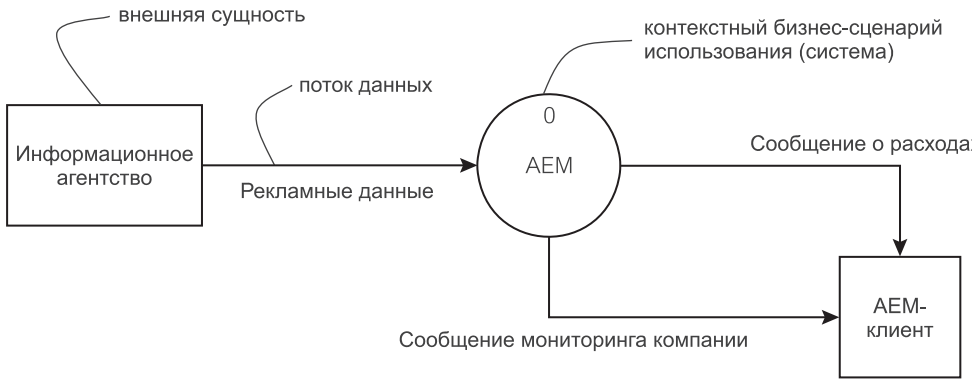


Рис. 6.1. Диаграмма бизнес-контекста

агентство) и `AEM Client` (AEM-клиент). Информационные агентства поставляют рекламные материалы для `AEM`. AEM-клиенты получают сообщения мониторинга компании и сообщения о расходах от `AEM`.

Преобразование рекламных данных в сообщения — внутренняя функция `AEM`. Проверка рекламы, ручной ввод и другие функции `AEM` формируют и снабжают данными, но они рассматриваются как внутренние потоки данных в пределах бизнес-сценария использования `AEM`. Например, поставка газет, журналов, видеокассет и т. д. для ручного ввода рассматривается как внутренняя функция в пределах системы.

6.3. Модель бизнес-сценария использования

Модель бизнес-сценария использования — *декомпозиция* (заимствование термина из диаграмм потоков данных) диаграммы бизнес-контекста. Бизнес-сценарий использования диаграммы бизнес-контекста расчленяется на небольшое число составляющих бизнес-сценариев использования. Взятые вместе, бизнес-сценарии использования, составляющие модель бизнес-сценариев использования, определяют те же самые функциональные возможности, что и бизнес-сценарий использования диаграммы бизнес-контекста.

6.3.1. Бизнес-сценарий использования и бизнес-акторы

Бизнес-сценарий использования является сценарием использования высокого уровня (случай бизнес-использования) в пределах рассматриваемой системы. Это — участок высокого уровня бизнес-функциональности с четкими границами. Бизнес-сценарии использования — первая попытка определения основных функциональных модулей (основных бизнес-операций) в предприятии.

Бизнес-акторы в модели бизнес-сценариев использования представляют собой конкретную версию **акторов** обычной диаграммы сценариев использования (глава 2). Актор — это роль, которую играет человек, система, устрой-

ство и т. д., когда он использует или, другими словами, связывается с услугами сценария использования.

Бизнес-актор может быть внешней сущностью (то есть вне возможностей системы) или он может быть ролью, которая является внутренней по отношению к системе (типа служащего за терминалом компьютера). Внутренний бизнес-актор — обычно **первичный актор** (он взаимодействует непосредственно с системой). Внешняя сущность обычно представляет **вторичный актор** (роль, которая общается с системой через посредника — обычно первичного актора). Рассмотрим банковское приложение, когда клиент подходит к кассиру (человеку или банкомату) и запрашивает получение денег со счета банка. Клиент — вторичный актор (и внешняя сущность). Первичный актор — кассир. Сценарий использования может называться `Withdraw Money` (выдача денег) или каким-то подобным образом.

Классификаторы (элементы модели) в диаграмме бизнес-сценариев использования могут быть связаны различными видами отношений [61], но отношения должны использоваться с умом. Бизнес-сценарии использования могут быть связаны отношением **ассоциации** (другие виды UML-отношений сценария использования — см. главу 2 — обычно не приветствуются в бизнес-сценариях использования).

Связь между актором и сценарием использования выражается в UML стереотипным отношением **«communicate»** (связывание). Важным отношением с широкими возможностями моделирования и программирования является **обобщение**. Это отношение находится между более общим классификатором и более определенным классификатором. Более определенный классификатор является «видом» базового классификатора. Обобщающие классификаторы включают бизнес-сценарии использования и бизнес-акторы в моделях бизнес-сценариев использования и бизнес-сущности в моделях бизнес-классов.

6.3.2. Модель бизнес-сценариев использования для AEM

Рис. 6.2 представляет диаграмму бизнес-сценариев использования для AEM. Модель использует преимущество небольшого визуального изменения в нотации UML, которое делает различия между бизнес-сценарием использования и регулярным (системным) сценарием использования (раздел 2.2.2). Бизнес-сценарий использования имеет наклонную черту внутри овала сценария использования. Точно так же бизнес-актор имеет наклонную черту в круге, представляющем голову человечка.

Модель бизнес-сценариев использования делит AEM-систему на четыре главных бизнес-сценария использования: `Data Collection` (сбор данных), `Data Verification` (проверка данных), `Valorization` (установление стоимости) и `Reporting` (формирование сообщений). Бизнес-сценарии использования связаны **однонаправленными ассоциациями** (то есть ассоциациями, представленными линиями со стрелками), чтобы показать основной бизнес-поток AEM — от сбора данных до проверки, оценки стоимости и формирования сообщений.

`Data Collection` связывается с `Media Outlet` (информационное агентство), чтобы получить рекламную зарегистрированную информацию (это обычно делается автоматически через соединение компьютер–компьютер).

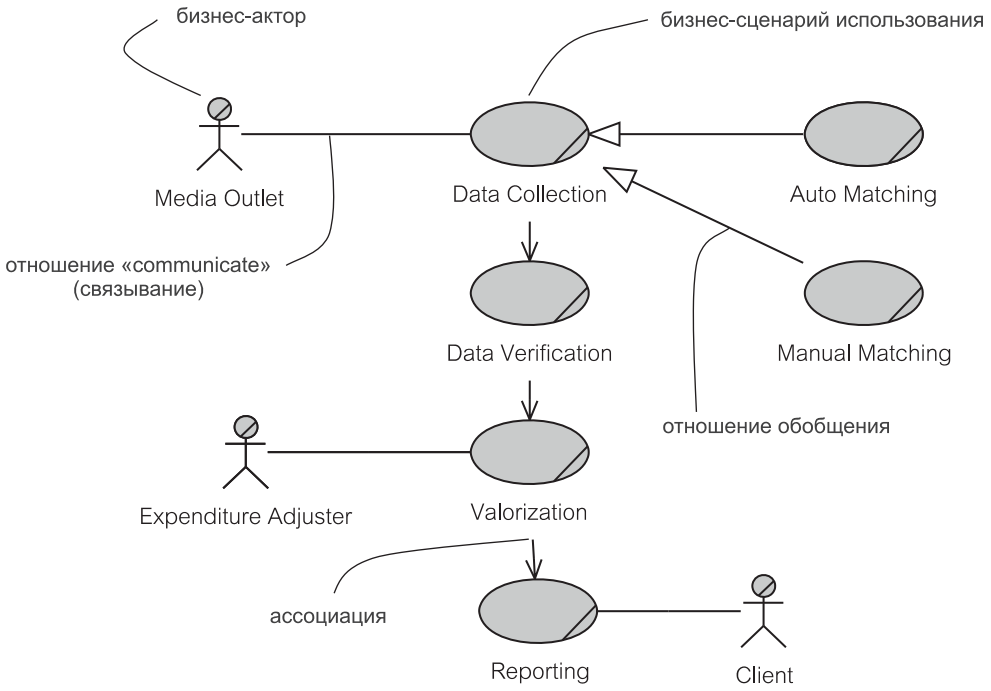


Рис. 6.2. Диаграмма бизнес-сценариев использования для АЕМ

Регистрация используется для сбора данных и проверки их соответствия рекламным объявлениям, уже размещенным в БД АЕМ — частично автоматической и частично ручной операции. Это показано на рис. 6.2 *отношениями обобщения*. `Auto Matching` (автоматическое согласование) является «видом» `Data Collection`, и `Manual Matching` (ручное согласование) также является «видом» `Data Collection`.

Рекламные данные вводятся в БД и затем подвергаются тщательной `Data Verification` (проверка данных). Задача проверки состоит в том, чтобы подтвердить, что все собранные части рекламы являются достоверными и логичны в контексте окружающей информации. Сюда включаются различные проверки на допустимость, типа объявления телевизионной передачи неразумной продолжительности или со временем, накладывающимся на программу телепередач (вместо намеченного перерыва для демонстрации рекламы).

После того, как рекламные объявления введены и проверены, они подвергаются `Data Valorization` (установление стоимости данных) — процессу назначения стоимости рекламных объявлений. `Expenditure Adjuster` (оценщик расходов) — бизнес-актор — обычно внешний консультант со знанием рыночных цен рекламы в различных средствах информации и при различных дополнительных условиях (типа покупательной способности рекламных агентств, конкурентоспособного положения различных рекламных средств информации, времени, когда показываются рекламные объявления — в случае телепередач и т. д.).

Бизнес AEM — продажа рекламной информации своим клиентам. Client (клиент) — бизнес-актор (внешняя сущность), готовый оплатить информацию из последнего бизнес-сценария использования в диаграмме — Reporting. Сообщения настраиваются на индивидуальных клиентов. Они могут быть поставлены клиентам на периодической (подписной) основе (с помощью электронной почты, на компакт-дисках, в бумажной форме и т. д.). Используется также формирование специальных сообщений для спорадических клиентов.

6.3.3. Альтернативная модель бизнес-сценариев использования для AEM

Модель бизнес-сценариев использования на рис. 6.2 отображает рабочий поток операций AEM (от сбора данных до формирования сообщений). Все бизнес-акторы в этой модели являются вторичными акторами — внешними вкладчиками в систему и получателями информации из AEM. Однако это только одно из многих возможных представлений высокого уровня функциональной структуры BOM. Рис. 6.3 представляет альтернативный взгляд.

Модель на рис. 6.3 имеет имя Quality Control (контроль качества) для бизнес-сценария использования, предварительно имевшего имя Data Verification (проверка данных). Это совместимо с названием отдела, ответственного за качество собранных данных о рекламе. Бизнес-сценарий использования по имени Valorization (установление стоимости) на рис. 6.2 теперь включен в Reporting (формирование сообщений) и не показан как отдельный бизнес-сценарий использования. Этот факт обозначен линией связи между бизнес-актором Rates Consultant (консультант по тарифам) и Reporting.

Рис. 6.3 представляет два дополнительных бизнес-сценария использования: Contact Management (управление деловыми партнерами) и Data Maintenance (сопровождение данных). Contact Management имеет классическую обязанность привлекать новых клиентов к бизнесу и обеспечивать потребности существующих клиентов. Это называется Contact Management (что лучше, чем Customer Management — управление клиентами), чтобы указать обязанность бизнес-сценария использования оставаться также в контакте с AEM-поставщиками (типа информационных агентств), консультантами (типа Rates Consultant) и поддерживать соединение с другими внешними сущностями (типа государственных информационных исследовательских центров).

Data Maintenance — удивительно сложный бизнес-сценарий использования. Его главная обязанность состоит в том, чтобы гарантировать корректность «стабильных данных» в БД AEM. Под «стабильными данными» мы подразумеваем описательную метаинформацию относительно рекламных объявлений (таких, как типы рекламных объявлений), средств информации (типа категорий средств информации), списков данных рынка сбыта (типа возможных статусов списка). Другие «стабильные данные» — это все то, что не подвергается постоянным изменениям в основном рабочем потоке AEM между сбором данных и формированием сообщений (наподобие адреса рынка сбыта или названия агентства).

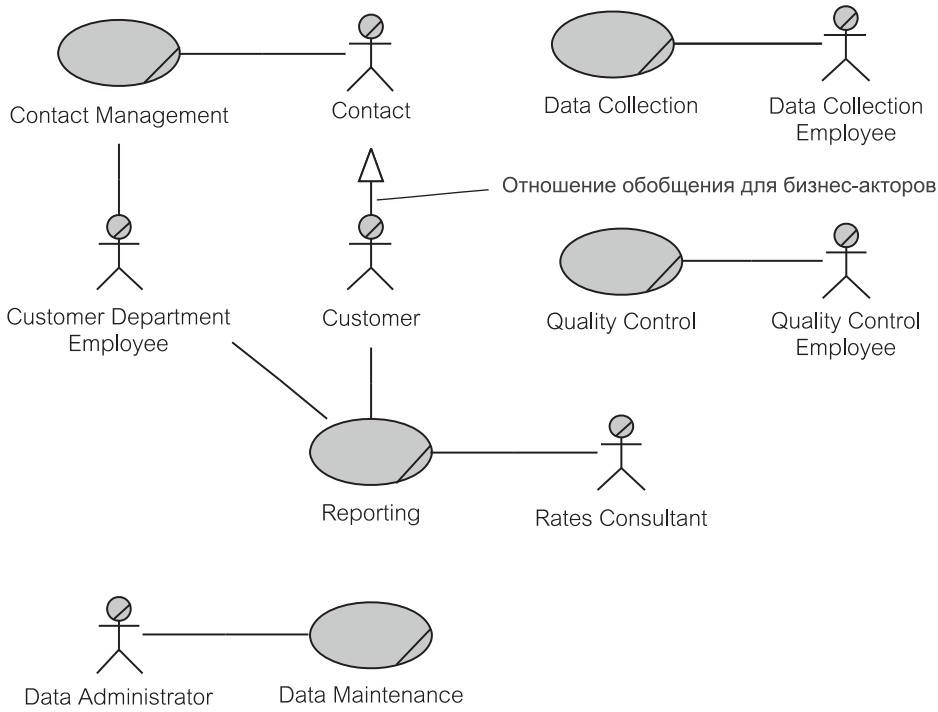


Рис. 6.3. Альтернативная диаграмма бизнес-сценариев использования для AEM

Рис. 6.3 представляет наиболее важные бизнес-акторы и наиболее важные линии связи с бизнес-сценариями использования. «Наиболее важные» означает наиболее уместные по отношению к тому, что мы хотим подчеркнуть в модели. Это звучит достаточно произвольно, но это — природа моделирования. Для модели любой разумной сложности никакие два разработчика моделей не придумают точно те же самые модели. Это не обязательно означает, что одна модель лучше другой — обе из них могут быть одинаково хороши или одинаково плохи.

Можно было бы поспорить, например, утверждая, что `Contact` (деловой партнер) связывается со всеми пятью бизнес-сценариями использования. Однако мы должны провести водораздел между тем, что является важным, и тем, что не так важно для конкретной точки зрения, представленной моделью. В этом случае мы предполагали, что основной бизнес-сценарий использования, с которым связан `Contact`, — `Contact Management`. Менее важные отношения «`communicate`» (связывание) для `Contact` не показаны, за исключением того, что подкласс `Contact`, называемый `Customer` (клиент), связан с `Reporting`.

Заметьте, что отношение обобщения может быть установлено для бизнес-акторов, а не только для бизнес-сценариев использования, как на рис. 6.2. Актор `Customer` является «видом» актора `Contact`. Это означает, что актор `Contact`, который одновременно является и актором `Customer`, связан

с `Reporting`. Обратите внимание на выразительную силу обобщения. Обобщения для бизнес-сценариев использования и для бизнес-акторов могут упростить графическое представление и все же передавать ту же самую семантическую информацию.

При сравнении моделей на рисунках 6.2 и 6.3 мы обращаем внимание, что те же самые элементы модели иногда называются по-другому. `Data Verification` на рис. 6.2 называется на рис. 6.3 `Quality Control`, `Expenditure Adjuster` (оценщик расходов) называется `Rates Consultant` (консультант по расценкам) и `Client` (клиент) называется `Contact`. Это создает потенциал для двусмысленности при установлении связи между различными заинтересованными сторонами проекта. Хотя практически только одна из этих двух моделей бизнес-сценариев использования закончится *продуктом* проекта, важно, чтобы все терминологические двусмысленности были размещены и решены в *глоссарии* системы (см. следующий раздел).

Таблица 6.1. Бизнес-глоссарий для AEM

Термин	Определения и объяснения
ad (advertisement) — реклама	Отдельная часть творческого материала, которая может быть радиопередачей, экранизацией, печатным изданием или иначе представленной и демонстрируемой любое число раз. Каждая ее демонстрация информационным агентством известна в AEM-системе как экземпляр рекламы
ad instance — экземпляр рекламы	Отдельная демонстрация рекламы, то есть каждый случай радиопередачи, экранизации или публикации
product — изделие	Единица изделия или услуги, которая может рекламироваться. Изделия могут быть категоризированы (то есть изделие может принадлежать к категории изделий). Категории — классификации изделий, как предусмотрено в AEM. AEM-система поддерживает иерархическую группировку категорий с неограниченным числом уровней в иерархии. Изделия могут быть категоризированы только на самом низком уровне иерархии категорий
category — категория	Классификация изделий, как предусмотрено в AEM, для цели формирования сообщений. Категории могут быть организованы в иерархические структуры категорий и подкатегорий. Изделия «категоризированы» на нижнем уровне иерархической структуры
organization — организация	Бизнес-сущность, с которой AEM имеет дело. Имеются различные типы организаций, включая рекламодателей, агентства, рынки сбыта, группы рекламодателей, группы агентств и поставщиков данных. Организация может быть одного, нескольких этих типов или не входить ни в один из них
agency — агентство	Организация, которая занимается планированием рекламы и закупкой средств информации для рекламодателя. Цель агентства состоит в том, чтобы оптимизировать расходы на рекламу
outlet — рынок сбыта	Организация, которая выставляет рекламу. В телевидении и на радио рынок определяется как станция, которая транслирует объявления. В кино и наружных средствах информации рынок определяется как компания, которая организует демонстрацию рекламы. В области печати рынок определяется как публикация, в которой напечатана реклама

advertiser — рекламодатель	Организация, которая использует средства информации, чтобы выставить свою творческую продукцию (организовать рекламу), продвинуть изделие (товар или услугу). Большая часть рекламодателей известна в AEM-системе как «прямые рекламодатели» (Direct Advertiser). Это рекламодатели, которые обходят агентства и места рекламирования в книгах непосредственно с помощью информационного агентства
advertiser group — группа рекламодателей	Организация, которая непосредственно или косвенно управляет одной или большим количеством других компаний и заинтересована в рассмотрении накопленных рекламных операций и расходов объединенных компаний. Группы рекламодателей не оплачивают какую-либо рекламу; сами рекламодатели внутри группы делают это. Группы рекламодателей также известны как холдинговые компании (Holding Companies)
agency group — группа агентств	Компания, которая непосредственно или косвенно управляет одним или большим количеством других агентств и заинтересована в рассмотрении накопленных планов и закупок средств информации для агентства или группы агентств. Группы агентств не выполняют никакого планирования или закупки средств информации; агентства внутри группы делают это сами
provider (auxiliary supplier) — поставщик (вспомогательный поставщик)	Организация, которая обеспечивает вспомогательные данные для AEM-системы. Эта информация включает демографические отчеты (разделение людей на группы на основе конкретных атрибутов типа возраста, пола и социально-экономических факторов), обзоры (оцененные и проектируемые средства просмотра, прослушивания и чтения для различных демографических категорий и для различных рынков сбыта), нормы (цены на рекламу различных рынков сбыта, чтобы определить стоимость объявлений), скидки (оцененные снижения рекламных цен из-за покупательной способности агентств при заказе объявлений через указанную среду)
adlink — рекламная цепочка	Объединение рекламы, изделий, которые она рекламирует, рекламодателей, кто платит за демонстрацию рекламы, и агентств, ответственных за заказ таких демонстраций. Рынки сбыта, которые выставили объявление, могут быть определены из описаний экземпляров рекламы

6.4. Бизнес-глоссарий

Работа над глоссарием [55] должна начаться на ранних стадиях разработки информационной системы и это должна быть непрерывная операция. **Глоссарий** определяет платформу связи между всеми заинтересованными сторонами проекта. Он включает все термины и определения, которые имеют любую — даже отдаленную — возможность быть неясными или неоднозначными.

Глоссарий может быть написан как простой текстовый документ в формате таблицы. Как минимум, таблица должна состоять из двух колонок: термин и определение. Когда глоссарий вырастет до существенных размеров, термины должны быть перечислены в алфавитном порядке.

6.4.1. Бизнес-глоссарий для AEM

Таблица 6.1 представляет начальный AEM-глоссарий. Он содержит термины, необходимые на начальных стадиях проекта, включая те, которые нужны при разработке модели бизнес-классов, являющейся нашей следующей задачей

(раздел 6.5). Термины в алфавитном порядке не сортируются — они объясняются в порядке, который облегчает последовательность обучения.

6.5. Модель бизнес-классов

Модель бизнес-классов показывает наиболее фундаментальные бизнес-сущности («бизнес-объекты») и основные отношения между ними. Модель разрабатывается одновременно с моделью бизнес-сценария использования. *Бизнес-сущность* — классификатор, который характеризует и участвует в выполнении основных бизнес-операций организации. Часто бизнес-сущность представляет моделирование информационной системы в терминах, определенных в глоссарии (таблица 6.1).

6.5.1. Бизнес-сущности

Бизнес-сущность представляет собой класс, но отличающийся от других UML-классов, необходимых в информационных системах, типа классов, ответственных за работу графического пользовательского интерфейса (GUI), логику программы или доступ к БД. UML обеспечивает графические средства, чтобы отличить бизнес-сущности от обычных классов системы. Бизнес-сущность представляется как круг с наклонной линией и горизонтальной линией под этим кругом.

Будучи классом, бизнес-сущность может иметь атрибуты и операции. Однако, являясь высоко абстрактным представлением бизнес-объекта, атрибуты и операции, по-видимому, не будут определены.

Три формы UML-отношений могут использоваться с бизнес-сущностями: **ассоциация**, **обобщение** и **агрегирование**. Основное назначение ассоциации и обобщения таково, как это объясняется в моделях бизнес-сценариев использования. Агрегирование — более строгая форма ассоциации со своим содержимым и своими свойствами.

Множественности можно показать на подходящих отношениях, то есть на ассоциациях и агрегированиях. Множественности могут быть перечислены в основной форме как «один к одному», «многие ко многим» или «один ко многим», они могут также содержать определение **участия** (или **необязательности**). Экземпляр сущности может и не участвовать в отношении. Это показывается значением 0 перед индикатором множественности (например, 0..1 означает необязательное (возможно, равное 0) участие для множественности «один»; 1..n означает обязательное (по крайней мере, равное 1) участие для множественности «многие»).

6.5.2. Модель бизнес-классов для AEM

Рис. 6.4 показывает диаграмму бизнес-классов для AEM. Модель содержит шесть бизнес-сущностей и первичные отношения между ними. Используются два вида UML-отношений: *ассоциации* и *агрегирование*. Показаны также основные *множественности* отношений (все они являются или «многие ко многим» или «один ко многим»).

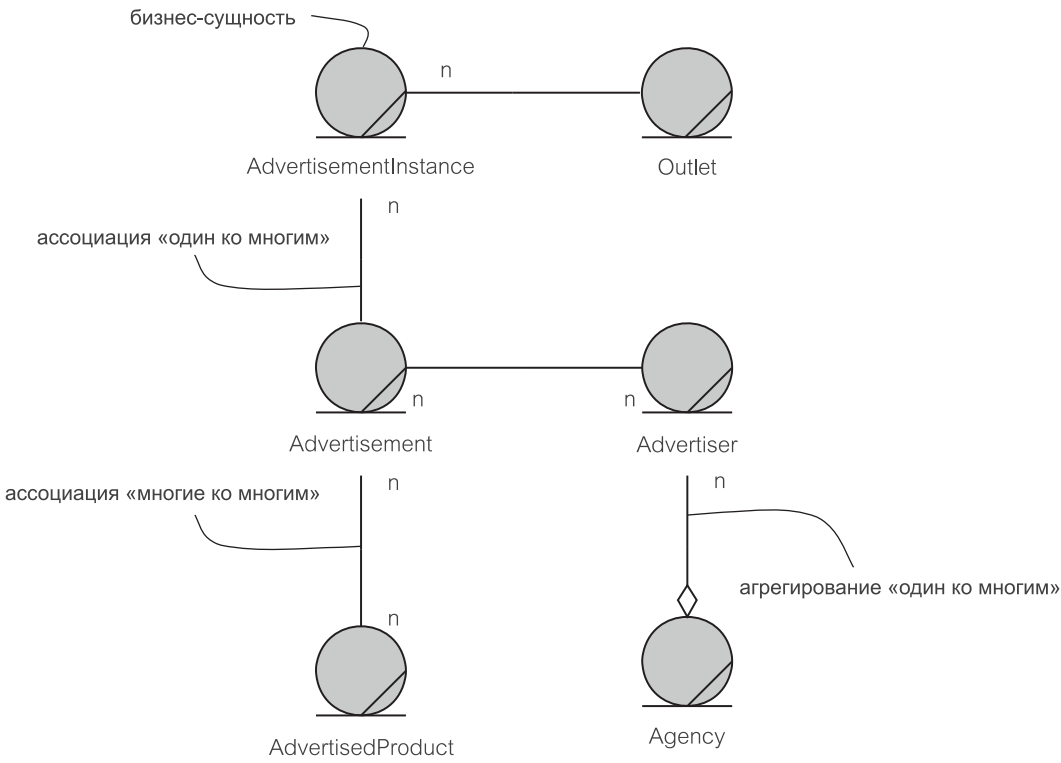


Рис. 6.4. Диаграмма бизнес-классов для AEM

`AdvertisementInstance` (экземпляр рекламы) реализует индивидуальное создание рекламы. Реклама — конкретная часть творческого материала, которая может быть радиопередачей, экранизацией, печатным изданием или иначе представленной и демонстрируемой любое число раз. `AdvertisedProduct` (рекламируемое изделие) охватывает изделие, которое является целью рекламы.

Имеются также три организационных класса. `Outlet` (рынок сбыта) хранит информацию относительно компаний, специализирующихся на средствах информации. `Advertiser` (рекламодатель) — организация, которая использует средства информации, чтобы рекламировать изделия. `Agency` (агентство) — организация, которая занимается планированием рекламы и закупкой средств информации для рекламодателя.

Диаграмма также показывает основные отношения между бизнес-классами. Отношение между `Agency` и `Advertiser` — агрегирование. Другие отношения — ассоциации. Каждый `Advertisementinstance` связан с единственным `Outlet` и единственной `Advertisement` (реклама). Одна и та же `Advertisement` может использоваться более чем одним `Advertiser` и может рекламировать больше чем один `AdvertisedProduct`. `Agency` представляет много `Advertiser`, но — согласно модели — `Advertiser` может быть представлен только единственным `Agency` (если дело обстоит не так, агрегирование не должно использоваться).

6.5.3. Альтернативная модель бизнес-классов для AEM

Модель бизнес-классов на рис. 6.5 — «шаг» усовершенствования модели из рис. 6.4. Новая модель добавляет два бизнес-класса: `Organization` (организация) и `Category` (категория). `Organization` — лишь базовый класс для `Outlet` (рынок сбыта) и `Customer` (клиент). Введение `Category` означает, что `Product` (изделие) может быть теперь категоризирован (то есть, связан с `Category`).

Модель увеличивает точность определений ассоциаций, добавляя свойства *участия* (*необязательного*) к определениям множественности отношений. Определения участия добавляют значительное количество вариантов к модели.

Мы знаем теперь, что информация относительно `Outlet` может быть размещена в БД AEM, даже если для него нет никакого `AdInstance` (экземпляр рекламы). Точно так же `AdInstance` и `Product` могут быть размещены, даже если мы не знаем (пока еще) описания их `Ad` (рекламы). `Advertiser` (рекламодатель) не обязательно должен рекламировать, используя `Agency` (агентство), но он может рекламировать и через несколько `Agency` (если дело обстоит так, то использование агрегирования на рис. 6.4 ошибочно).

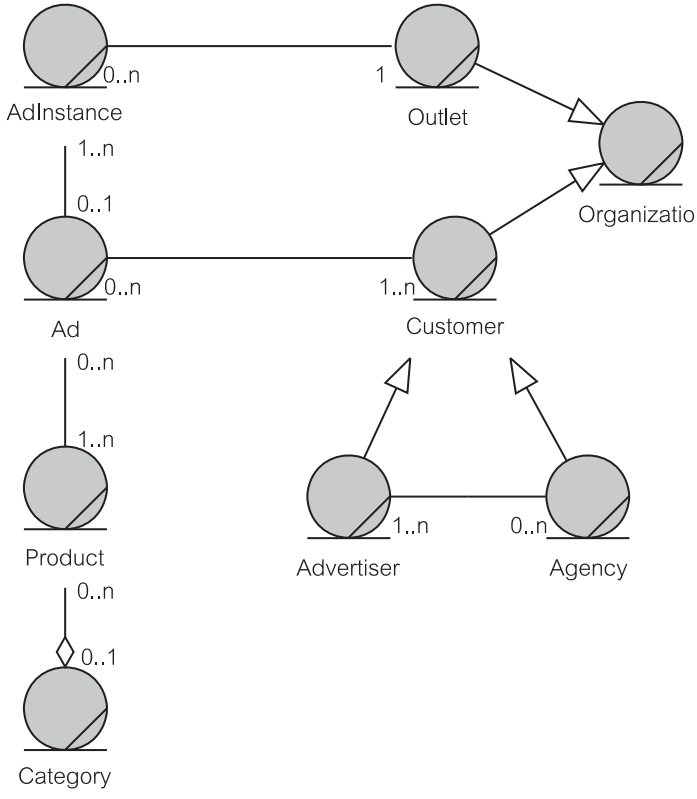


Рис. 6.5. Альтернативная диаграмма бизнес-классов для AEM

С другой стороны, `AdInstance` не может существовать в БД, если он не связан с `Outlet`. `Ad` не может существовать, если она не связана, по крайней мере, с одним `AdInstance`, одним `Product` и одним `Customer`. `Product` может быть категоризирован, самое большее, одной `Category`. Одна и та же `Category` может быть связана со многими `Product`, но `Category` может также существовать и сама по себе.

Резюме

1. *Модель бизнес-объектов* (Business Object Model — BOM) — объектно-ориентированная модель, которая описывает основной бизнес предприятия. Описание определяет границы предприятия (диаграмма бизнес-контекста), его бизнес-обязанности (модель бизнес-сценариев использования), и его информационные сущности (модель бизнес-классов).
2. Моделирование бизнес-объекта — разработка ПО, а также производство *продуктов* — документов, моделей, описаний, ПО и т. д.
3. Продукт модели содержит *классификаторы* — элементы модели, которые описывают поведенческие и структурные особенности предприятия или системы.
4. Предприятие, рассматриваемое в книге, — компания исследования рынка по имени AEM (Advertising Expenditure Measurement — оценка расходов на рекламу). Ее основной бизнес — запись, обработка и продажа информации количественной оценки относительно рекламных объявлений во всех основных аудиовизуальных и печатных средствах информации.
5. *Диаграмма бизнес-контекста* состоит из одного бизнес-сценария использования, представляющего предприятие и потоки данных от/к внешних(м) сущностей(ям) (другие организации или системы).
6. *Модель бизнес-сценариев использования* определяет бизнес-сценарии использования, бизнес-акторы и отношения между ними. *Бизнес-сценарий использования* — главный функциональный модуль, создаваемый с целью выполнения бизнес-задачи уровня предприятия. *Бизнес-актор* представляет нечто, взаимодействующее с бизнес-сценарием использования (через отношение *«communicate»* — связывание). Бизнес-сценарии использования могут быть связаны *ассоциациями* и могут быть классифицированы с помощью отношения *обобщения*.
7. Бизнес-*глоссарий* — список бизнес-терминов и определений. Важно начать строить глоссарий в процессе разработки как можно раньше.
8. Существование глоссария облегчает разработку модели бизнес-класса, которая определяет бизнес-сущности и отношения между ними. *Бизнес-сущность* — классификатор для бизнес-объектов — того, что может содержать данные и может обладать поведением. Отношения включают ассоциации, агрегирования и обобщения. Множественность и необязательность отношения (ассоциации или агрегирования) определяют бизнес-ограничения и правила, управляющие отношением.
9. Хотя желательно, чтобы, в конечном счете, была определена единственная BOM для предприятия (и все ключевые заинтересованные стороны

договариваются об этом), возможны альтернативные модели. Альтернативные модели служат полезной цели, обеспечивая различные точки зрения относительно моделирования на предприятии.

Ключевые термины

Обзорные вопросы

Вопросы для обсуждения

1. Каковы цели системы? Как это можно определить?
2. Каково различие между бизнес-актором и внешней сущностью?
3. Каково различие между бизнес-актором и актором?
4. Каково различие между первичным актором и вторичным актором?
5. Каково различие между потоком данных и отношением «`communicate`» (связывание)?
6. Каково различие между ассоциацией и отношением?
7. Какие отношения обобщения полезны в диаграммах бизнес-сценариев использования и в диаграммах бизнес-классов?
8. Почему глоссарий полезен в модели бизнес-классов?
9. Обсудите важность определения множественностей отношений в диаграмме бизнес-классов. Является ли жизненно важным определить свойства участия (необязательности) отношений в диаграмме бизнес-классов?
10. Обсудите пользу отношений агрегирования в моделировании бизнес-классов.
11. Каковы главные продукты BOM?

Вопросы учебного примера

1. Что такое сбор и согласование данных? Каково различие между ручным и автоматическим согласованием? Как эти операции связаны со сбором данных и проверкой данных? Можно ли ручное согласование рассматривать как вид или часть проверки данных?
2. Что такое рекламная цепочка? Как она связана или может быть связана с согласованием данных? Является ли задание рекламной цепочки функцией сбора данных или проверки данных? Обсудите вашу точку зрения.
3. Что такое установление стоимости? Что такое участие радиослушателей (телезрителей) или читателей? Как вы думаете, связаны ли установление стоимости и радиослушатели (телезрители)/читатели? Должно ли быть установление стоимости самостоятельным бизнес-сценарием использования или частью бизнес-сценария использования `Reporting` (формирование сообщений)?
4. Имеются две главные категории AEM-сообщения. Каковы они? Отметьте важность обеих категорий сообщений для AEM-бизнеса.
5. Что такое сопровождение данных? Считаете ли вы его допустимым бизнес-сценарием использования?

Примеры задач

Упражнения учебного примера

1. Прочитайте заново раздел 6.1 и рассмотрите модели бизнес-сценариев использования в разделе 6.3. Постройте диаграмму бизнес-сценариев использования, которая объединяет диаграммы на рисунках 6.2 и 6.3, а также учитывает возможные расширения, основанные на информации из раздела 6.1.

   «Комбинирование» не должно означать непосредственное объединение структур на рисунках 6.2 и 6.3 (в любом случае такое объединение будет невозможно из-за несоответствий в терминологии). Удачное использование отношения обобщения обеспечивает хороший механизм для объединения и расширения моделей.
2. Расширьте глоссарий в таблице 6.1 раздела 6.4, чтобы отразить термины, рассмотренные в моделях бизнес-сценариев использования раздела 6.3 и не включенные в глоссарий.
3. Разработайте модель бизнес-классов, которая использует термины глоссария в таблице 6.1 раздела 6.4 как бизнес-классы. Для отношений покажите множественности со свойствами участия.

Небольшой проект — оценка расходов на рекламу

Рассмотрим следующую «формулировку задачи» относительно AEM (Advertising Expenditure Measurement — оценка расходов на рекламу). Формулировка дополняет описание основного бизнеса AEM в разделе 6.1, но она рассматривается здесь как отдельное представление.

AEM-система должна удовлетворять двум множествам пользователей: внутренним пользователям (AEM-служащие) и внешним пользователям (AEM-клиенты и другие деловые партнеры). Мы хотим позволить нашим внешним пользователям иметь доступ к нашим данным с возможно минимальной задержкой между сбором данных о рекламных операциях и готовностью нужной информации. Это имеет отношение как к напечатанным (твердая копия), так и к программным копиям сообщений, а также к файлам данных, которые поставляются некоторым клиентам, кто купил наше ПО AEM*Reporter*, формирующее сообщения.

Наш отдел работы с клиентами (Customer Department) занимается поиском новых клиентов, исследуя те области, где наши данные имеют потенциальный рынок. Коммерческий штат будет затем использовать эту информацию, чтобы установить контакт с предполагаемыми клиентами для заключения бизнес-контракта на поставку сообщений AEM. Как только мы выиграем клиента, нашей основной целью должно быть обеспечение гарантий, что он будет полностью удовлетворен обслуживанием и данными, которые он получит.

Управление деловыми партнерами касается не только наших клиентов. Оно охватывает также наших вспомогательных поставщиков. Это организации, которые обеспечивают AEM данными (главным образом, информационные агентства, но также и организации, которые снабжают нас вспомогательными данными — прайс-листы рекламы, радио- телезрители и читатели, а также другие подобные источники информации). Существуют контракты с некоторыми поставщиками о поставке вспомогательных данных.

Большинство клиентов AEM находятся на ежегодных контрактах. Контракт определяет его период, ограничения в терминах доступа к нашей информации, частоту поставки информации, цель рекламных рынков (включая информационные агентства и географические области) и любые другие условия по предоставлению данных. Эта информация поступает в нашу группу ожидаемых поступлений (Accounts Receivable) с целью составления счетов и отслеживания платежей. Постоянное требование — составление сообщений с анализом возможного сбыта. Это очень помогает в перезаключении контрактов.

AEM собирает информацию из различных источников. Основная форма сбора данных — электронная передача отчета о рекламной деятельности от информационных агентств в списки операций (то есть, компьютерные файлы). Это обычно производят в середине ночи, устанавливая соединение «компьютер — компьютер» с информационными агентствами и получая данные об операциях за предыдущие 24 часа. Рекламные данные, которые не могут быть собраны с помощью электроники, вводятся вручную из рекламных источников. В случае использования прессы информация берется из газет и журналов. Для кино и наружной рекламы мы ожидаем, что соответствующие информационные агентства пошлют нам для использования рекламные сообщения.

AEM собирает приблизительно полмиллиона экземпляров рекламы каждую неделю. Чтобы справиться с таким объемом данных наиболее эффективным образом, мы пытаемся во время ввода данных автоматически связывать

собранные данные с рекламными объявлениями, уже находящимися в нашей БД. Используемый принцип здесь такой: если мы видели детали рекламы прежде, мы можем использовать те же самые решения для отчета и в дальнейшем. Не распознанный материал помечается, чтобы обратить внимание работника нашего штата контроля качества.

Штат контроля качества выполняет проверку всех автоматически собранных и помеченных данных, а также всех данных, собранных вручную (то есть введенных штатом сбора данных из источников рекламы, когда автоматическое распознавание рекламы невозможно). В процессе сбора и проверки данных стараются определить так называемые рекламные цепочки. Рекламная цепочка — связь рекламы с рекламируемым изделием, рекламодателем и агентством, используемым рекламодателем.

После того, как данные собраны и проверены, они подвергаются процессу объединения, чтобы создать итоговые сообщения клиентам. Процесс объединения требует, чтобы собранные и проверенные данные были сначала скопированы из нашей рабочей БД в хранилище данных (то есть БД, которая содержит всю историю операций с рекламными данными и которая предназначена непосредственно для анализа данных).

Чтобы сделать это, данные, собранные в отдельные группы по информационным агентствам (например, канал TV), а также дни объединяются в группы по типам средств информации (например, столичное телевидение) и неделям. И общие, и более специализированные сообщения требуют обновления рекламной информации с учетом тарифов на рекламу (расходов) и сведений о прослушивании/чтении. Тарифы должны отражать максимально возможные реальные цены, которые могут быть заплачены за посредничество, в зависимости от ряда параметров (например, дня и времени рекламы, покупательной способности агентства, нашей собственной сообразительности и наблюдательности и т. д.). Сведения о прослушивании/чтении необходимы, чтобы измерить эффективность воздействия нашей рекламы на прослушивание/чтение информации нашими клиентами.

AEM продает информацию о рекламных операциях своим клиентам. Мы можем распределять эту информацию разными способами. Печатные сообщения — только один из них. Поскольку мы развиваемся во времени, значение предоставления информации на бумаге будет уменьшаться, пока вообще не перестанет пользоваться спросом. Сообщения в виде программных копий, посланных электронным способом нашим клиентам, будут требоваться в течение обозримого будущего. Вместо сообщений мы можем обеспечивать наших клиентов файлами данных, которые клиент может использовать для формирования своих собственных сообщений, используя наше ПО AEM*Reporter*, формирующее сообщения. Наконец, нашим лучшим клиентам можно также позволить соединяться с нашим хранилищем данных и разрешить им доступ к данным, на которые они заключили контракт.

Упражнения

1. Постройте модель бизнес-сценариев использования, отражающую формулировку задачи для рассмотренного небольшого проекта. Постарайтесь не быть под влиянием моделей бизнес-сценариев использования для AEM,

представленных в книге. Практической стартовой точкой должно стать чтение формулировки задачи и выделение кандидатов на бизнес-сценарии использования и кандидатов на бизнес-акторы (желательно использовать выделение в тексте различными цветами).

2. Постройте модель бизнес-классов, отражающую формулировку задачи для небольшого проекта. Попытайтесь не быть под влиянием моделей бизнес-классов для АЕМ, представленных в книге. Не учитывайте пока бизнес-сущности, требуемые для поддержания хранилища данных (объединение данных и распределение информации о рекламных операциях между клиентами). Для отношений покажите множественности со свойствами участия. Практическая стартовая точка здесь — чтение формулировки задачи и выделение кандидатов в бизнес-классы. Обратите также внимание на то, что бизнес-акторы модели бизнес-сценариев использования являются кандидатами в бизнес-классы в модели бизнес-классов.

Глава 7

Объектная модель предметной области

Модель бизнес-объектов (Business Object Model — BOM), рассмотренная в главе 6, является моделью предприятия. **Объектная модель предметной области** (Domain Object Model — DOM) является моделью подмножества BOM. Предприятие имеет одну BOM и много DOM. В этой книге BOM представляет AEM — предприятие, оценивающее расходы на рекламу. С любым основным бизнес-сценарием использования BOM можно обращаться как с DOM.

Как отмечается в этой главе, границами проекта информационной системы может быть предметная область. **Предметная область** — это прикладная область (подсистема) информационной системы предприятия. Она определяет границы **разрабатываемого приложения**. Однако при итеративном и пошаговом подходе к производству ПО предметная область может быть слишком большой, чтобы представлять границы для итерации разработки. Более вероятно, что один или пара сценариев использования, определенных в DOM, зададут границы итерации.

DOM, рассмотренная в этой главе, — управление деловыми партнерами (Contact Management — CM) (см. также [61]). Управление деловыми партнерами представляет главный учебный пример книги — управление электронной почтой (Email Management — EM). EM — *разрабатываемое приложение*. Рис. 7.1 представляет диаграмму Венна, показывающую зависимости между BOM, DOM и EM учебного примера. Обратите внимание, что в этом контексте некоторые небольшие проекты, представленные в конце глав, принадлежат другим DOM, то есть, не CM. Однако все небольшие проекты находятся в пределах BOM книги.

DOM — больше чем определение унифицированного процесса (Unified Process — UP) модели предметной области. В UP модель предметной области — продукт бизнес-моделирования и представление концептуальных классов реального мира в интересующей предметной области. Рискуя упрощением, можно сказать, что модель предметной области UM является моделью класса, напоминающей семантические модели сущностей-отношений, которые популяризируют структурные методы анализа. В частности, модель предметной области UP визуализирует классы, их атрибуты и отношения, но не операции.

DOM — относительно полная UML-модель анализа, которая может включать широкий диапазон методов и технологий UML. Однако полная DOM не будет создана, если границы *разрабатываемого приложения* являются под-

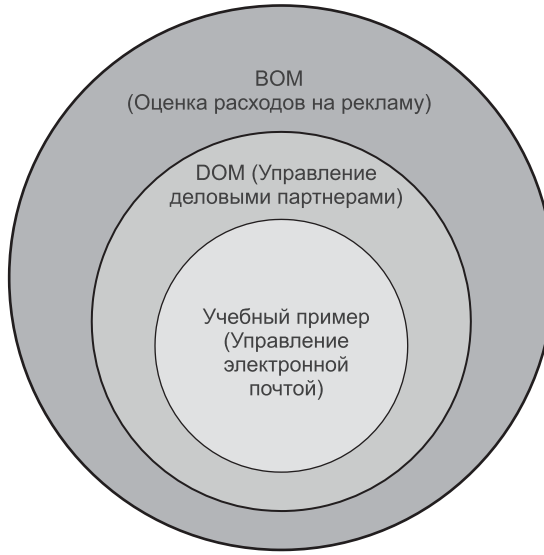


Рис. 7.1. Зависимости между BOM, DOM и учебным примером

множеством DOM. В этой книге DOM — только «связь» между BOM и учебным примером EM (Email Management — управление электронной почтой). BOM и DOM вместе обеспечивают знание исходных данных и окружающей среды для разрабатываемого приложения.

7.1. Управление деловыми партнерами — предметная область

CM (Contact Management — **управление деловыми партнерами**) — бизнес-функция AEM. CM находится на «боковой дороге» главной цепочки технологического процесса AEM, состоящей из сбора данных, установления стоимости и формирования сообщений (раздел 6.3). Однако CM — главный вкладчик в цепочке вычислений AEM, использующий концепцию, которую популяризирует Майкл Портер ([85], см. также [61]).

CM ответственно за связь с деловыми партнерами AEM, в преобладающем числе — клиентами и поставщиками. Оно включает планирование, инициализацию, регистрацию результатов связи, а также обработку информационных деталей деловых партнеров. CM можно также рассматривать как функцию поддержки для цепочки технологического процесса всякий раз, когда возникает потребность связи в пределах этой цепочки, но передается к CM для обработки. Примеры включают запросы к информационным агентствам относительно погрешностей в электронных зарегистрированных данных (раздел 6.1), запросы, разрешающие неизвестные рекламные цепочки (раздел 6.4), запросы к агентствам и рекламодателям, чтобы проверить точность цен рекламных объявлений и т. д.

СМ — нечто вроде связи «человек — человек», являющейся очень важным вкладчиком в организационные знания, которые должны быть «задокументированы». Документирование связи «человек — человек» является попыткой управлять **неструктурированной информацией** — письменная корреспонденция, факсимильные обмены, электронная почта, запись голоса, деловые презентации, Web-страницы и т. д. С точки зрения БД это все примеры **мультимедийных данных**.

В современной практике мультимедийные данные не структурированы — они хранятся как недифференцированные потоки байтов без знаний о структуре их информационного содержания. **Описательные атрибуты** (типа того, кто создавал данные, предметная область, ключевые слова и т. д.) хранятся вместе с объектом мультимедиа, чтобы «восстанавливать» его информационное содержание. Описательные атрибуты (которые сами по себе являются **структурированной информацией**) размещаются в БД и используются для доступа к объектам мультимедиа (в той же самой или другой БД или в файловой системе).

СМ — система регистрации описательной информации о связи «человек — человек» и, если требуется и это возможно, обеспечивает доступ к неструктурированным источникам этой описательной информации. СМ хранит описания переговоров с новыми клиентами, характеристики деловых партнеров, последующие действия, требования клиентов, поставки сообщений, вопросы производства, вытекающие из «технологического процесса» и т. д.

7.2. Модель сценариев использования предметной области

В терминах концепций моделирования **модель сценариев использования предметной области** напоминает модель бизнес-сценариев использования. Обе являются представлениями бизнес-функций, отношений между ними и отношений со средой. Различие в том, что модель бизнес-сценариев использования является ориентируемой на предприятие и обычно не стремящейся к непосредственной реализации, в то время как модель сценариев использования предметной области более «приземленная», по возможности ориентируемая на проект и может служить как контракт между разработчиками и заинтересованными сторонами проекта.

Модель сценариев использования предметной области создается из многих источников. Отправная точка — BOM и модель бизнес-сценариев использования в частности, но также и глоссарий, и модель бизнес-классов. Другие источники, «вдохновляющие моделирование», включают бизнес-модели, организационные структуры и процессы, требования посредников, любые дополнительные спецификации для системы и т. д.

7.2.1. Сценарии использования и акторы

Сценарий использования представляет главную часть функциональных возможностей системы. **Актор** — роль, которую кто-то или что-то играет в отношении сценария использования. Актор связывается со сценарием исполь-

зования (через отношение «communicate» — связывание) и ожидает от него некоторую обратную связь — величину или заметный результат.

Иногда желательно рисовать многократные диаграммы сценариев использования для одной и той же предметной области, при этом каждая из диаграмм подчеркивает свой аспект модели или ее части. Две типичных точки зрения (кроме глобальной модели) следующие:

- никаких акторов — только сценарии использования и их отношения;
- точка зрения единственного актора — все сценарии использования для каждого важного актора.

Диаграммы сценариев использования (раздел 2.2.2 в главе 2) — UML-технология для визуализации сценариев использования, акторов и их отношений. Визуализация весьма распространена и часто не очень полезна. *Спецификации сценариев использования*, которые являются текстовыми документами, размещенными CASE-средством, являются главным достоинством и привлекательным моментом моделирования сценария использования.

7.2.2. Отношения сценариев использования

Акторы в модели могут быть категоризированы посредством отношений обобщения. Никакие другие отношения UML не встречаются между акторами. Сценарии использования могут быть связаны отношениями ассоциации и обобщения и двумя стереотипными отношениями, называемыми «include» (включение) и «extend» (расширение). Акторы и сценарии использования связаны специальным видом ассоциации, называемой **отношением «communicate»** (связывание).

Рис. 7.2 (дублированный здесь из рис. 2.10) показывает, как классификаторы (элементы модели) могут быть связаны в модели сценариев использования. Сценарии использования связаны, потому что для решения задач системы они обычно должны взаимодействовать. Однако рекомендуется ограничивать отношения между ними. Показ всех отношений затеняет смысл модели. При показе только отобранных отношений возникает вопрос, почему эти отношения более важны, чем другие. Здесь нужно помнить, что, в конечном счете, диаграммы сценариев использования играют вторичную роль по отношению к текстовым документам, определяющим каждый из них.

Принцип благоразумного использования отношений применяется также к двум отношениям-стереотипам в UML, встречающимся при моделировании сценариев использования, — отношениям «include» и «extend» (раздел 2.2.2). **Отношение «include»** означает, что выполнение одного сценария использования охватывает (всегда) функциональные возможности включенного сценария использования. **Отношение «extend»** означает, что выполнение одного сценария использования может потребовать расширения (иногда) функциональными возможностями второго сценария использования.

Отношения «include» и «extend» всегда **однонаправлены**, чтобы обозначить направление включения или расширения. Отношения «communicate» и ассоциации могут иметь, а могут и не иметь стрелки, означающие, что передвижение возможно лишь в одном направлении. Отсутствие стрелки означает, что или **навигация** возможна в обоих направлениях,

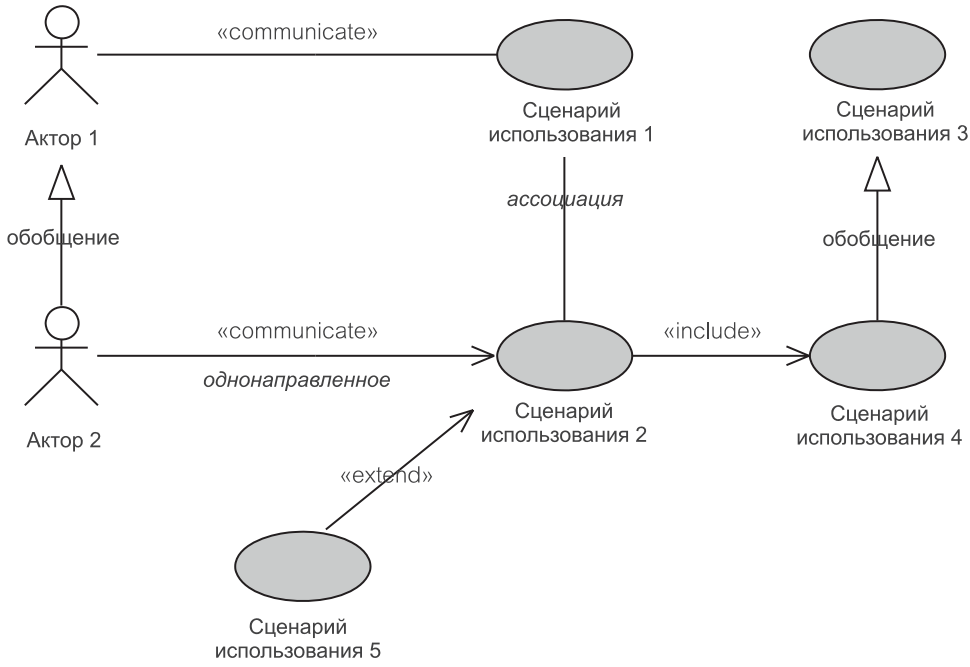


Рис. 7.2. Отношения между классификаторами модели сценариев использования

или направление навигации не определено (то есть, возможно, что это будет определено в будущем, но пока нас не волнует). Обратите внимание, что навигация в обоих направлениях представляется одной прямой линией, а не линией с двумя стрелками.

7.2.3. Модель сценариев использования для управления деловыми партнерами

Управление деловыми партнерами (Contact Management — CM) — сложная предметная область. Бизнес большинства организаций развивается вокруг удовлетворения клиентов и надежности поставщиков. AEM не является исключением. Однако в этой книге CM — просто дорожка к учебному примеру — управлению электронной почтой. По этой причине предложенное представление CM несколько упрощено.

Рис. 7.3 — возможная модель сценариев использования для CM. Она состоит из пяти сценариев использования: `Negotiate New Customer` (переговоры с новым клиентом), `Record Customer Requirements` (запись требований клиента), `Maintain Contact Information` (поддержание информации о деловом партнере), `Track Contact Events` (отслеживание событий делового партнера), `Handle Production Queries` (управление производственными запросами). Следуя нашей рекомендации в разделе 7.2.1, никакие отношения между сценариями использования не моделируются.

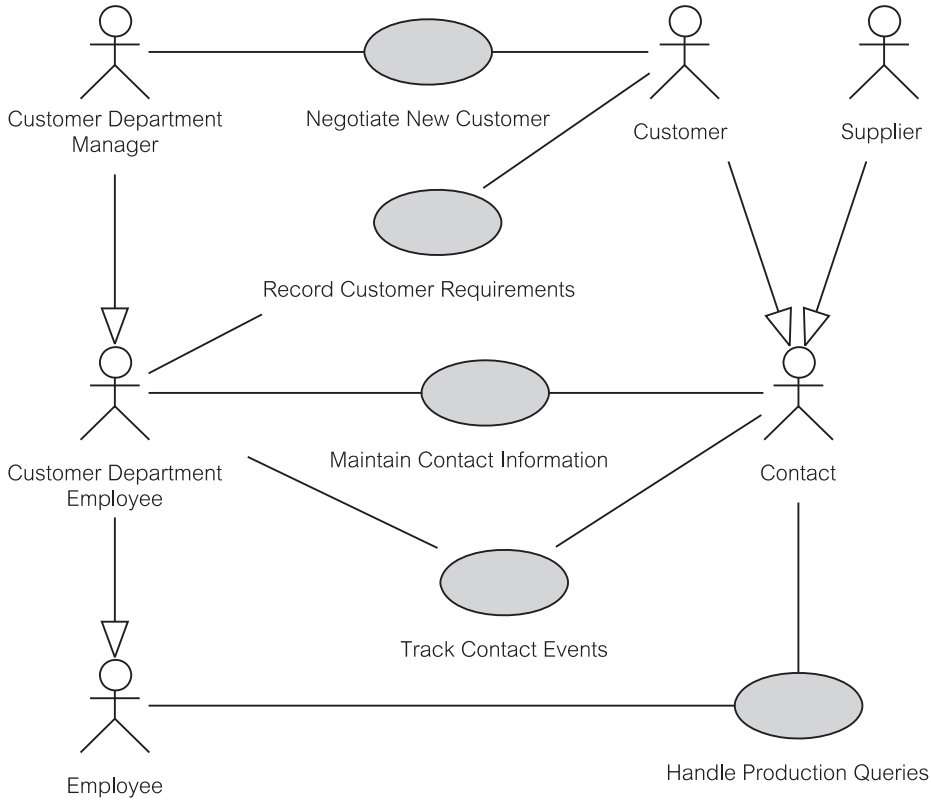


Рис 7.3. Диаграмма сценариев использования для CM

Отношения обобщения между акторами упрощают модель. Например, мы знаем, что `Customer Department Manager` (менеджер отдела работы с клиентами) — единственный актор, который связан с `Negotiate New Customer`. Но `Customer Department Manager` может также быть связан с `Handle Production Queries`. Это потому, что `Customer Department Manager` является «разновидностью» `Customer Department Employee` (служащий отдела работы с клиентами), а `Customer Department Employee` является, в свою очередь, «разновидностью» `Employee` (служащий).

Жизненно важной операцией CM является поиск новых клиентов. Это делается путем исследования тех возможных деловых партнеров, для которых информация AEM о рекламных операциях имеет потенциальный интерес. Эти партнеры являются предполагаемыми клиентами, и с ними проводится работа в рамках `Negotiate New Customer`. `Customer Department Manager` отвечает за привлечение этих перспективных клиентов и сообщение деталей AEM-услуг. Цель состоит в том, чтобы заключить контракт с перспективным клиентом. Если от клиента не добились никакого интереса, переговоры регистрируются в системе с датой, когда следует возобновить контакт с этим перспективным клиентом.

`Record Customer Requirements` отвечает за управление спецификациями потребностей AEM-клиентов в информации о рекламных операциях. Спецификации включают категорию и частоту предоставления информации о рекламных операциях, в которых заинтересован клиент (включая диапазон рекламируемых изделий или категорий, интересующие информационные агентства, географическое расположение рынков, промежутки времени, для которых разыскиваются сообщения, рекламодателей и агентства, интересующие клиента).

`Maintain Contact Information` — обычная функция поддержания сведений о клиентах и поставщиках, включающих адреса, телефоны и характеристики факса, тип поставляемых рекламных данных, людей для контакта. Этот сценарий использования дополнен внутренними сведениями типа относительной важности клиента для AEM-бизнеса, надежности поставщика, изменений характера контракта (например, когда агентство с текущим AEM-контрактом потеряет рекламодателя как своего клиента).

Все операции с деловыми партнерами и сеансы связи между служащими отдела работы с клиентами и деловыми партнерами AEM регистрируются внутри системы. Отсюда исключаются отдельные операции, которые не кончаются значимыми результатами для AEM. `Track Contact Events` содержит отслеживание операций, которые приводят или могут в потенциале привести к значимым для AEM бизнес-результатам. `Track Contact Events` может объединить операции, предпринятые одним или несколькими служащими за определенный период времени так, чтобы их можно было рассматривать как единое целое. Такое объединение с четкой целью называется задачей.

`Handle Production Queries` управляет всеми запросами деловых партнеров, возникающими в пределах реализации «цепочки технологического процесса». Запросы могут быть сформированы на любом шаге цепочки — в процессе сбора данных, в течение проверки данных, в течение установления стоимости рекламных объявлений и при производстве и поставке сообщений о рекламных операциях.

7.2.4. Альтернативная модель сценариев использования для управления деловыми партнерами

Управление деловыми партнерами можно рассматривать как функцию управления документами, передаваемыми к/от партнерам(ов), планируя операции служащих, связанные с деловыми партнерами, записывая уже выполненные операции и, конечно же, рассматривая информацию о партнерах.

Управление документами включает неструктурированные документы (раздел 7.1). Планирование операций может быть частью более широкой задачи управления электронными календарями служащих (типа функции Calendar программы Microsoft Outlook). Все записи по отношению к деловым партнерам могут быть связаны и прослежены с помощью календарей. Записи в календарь, с которыми приходилось иметь дело, могут быть зарегистрированы в электронных временных списках служащих.

Вышеупомянутые высказывания могут привести к альтернативной модели сценария использования, которая подчеркивает базовые (менее настроенные)

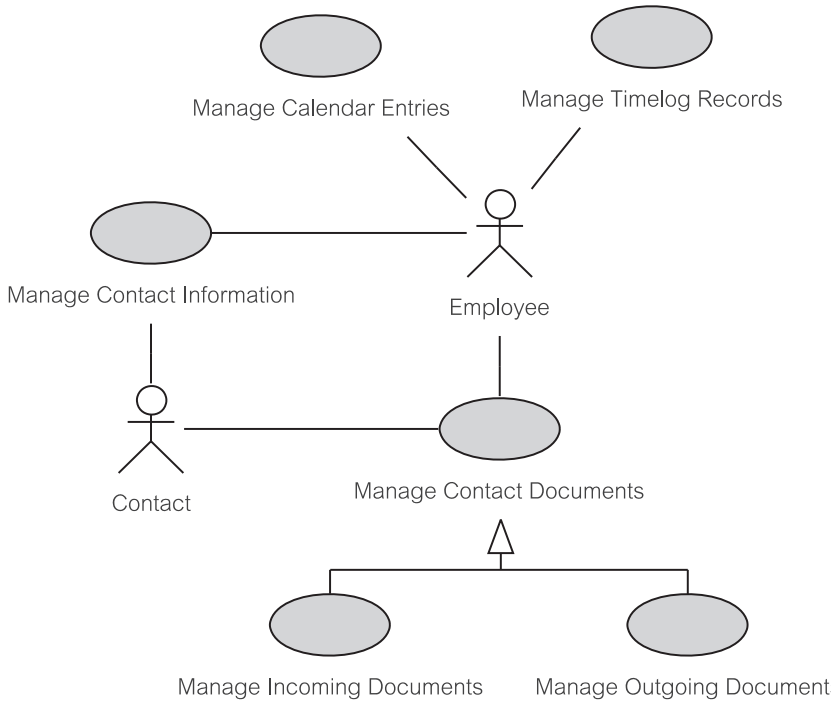


Рис. 7.4. Альтернативная диаграмма сценариев использования для СМ

функции типичного СМ-приложения и скрывает специфичные СМ-операции в базовых сценариях использования. Рис. 7.4 представляет альтернативную модель.

Модель на рис. 7.4 делит СМ на четыре главных сценария использования: `Manage Calendar Entries` (управление записями календаря), `Manage Timelog Records` (управление записью временного протокола), `Manage Contact Information` (управление информацией о деловых партнерах) и `Manage Contact Documents` (управление документами деловых партнеров). Имеются только два базовых актора, идентифицированные в модели: `Employee` (служащий) и `Contact` (деловой партнер). `Contact` не связан с `Manage Calendar Entries` и `Manage Timelog Records`, поскольку эти сценарии использования рассматриваются как внутренние по отношению к АЕМ-служащими.

`Manage Calendar Entries` отвечает за планирование и отслеживание встреч, собраний и других взаимных связей между АЕМ-служащими и деловыми партнерами. Каждый служащий имеет свой собственный электронный календарь. Записи в календарь могут быть введены вручную служащим, могут быть запланированы для служащего другими служащими (диспетчерами) или могут быть автоматически введены другими функциями АЕМ-системы (типа обратиться с какими-либо запросами к деловым партнерам во время сбора данных или их проверки).

Хотя это явно и не прослеживается на рис. 7.4, имеются два главных вида записей в календарь: временные и невременные. Временные записи определяют дату и время операции: условленная встреча, собрание, телефонный звонок и т. д. Невременные записи — операции и события, которые или не имеют определенных даты/времени или длятся в течение одного или большего количества дней (типа «нахождения» в командировке).

Предполагается, что `Manage Calendar Entries` включает функциональные возможности отслеживания событий, связанных с деловыми партнерами, и планирования производственных запросов (явно смоделированных на рис. 7.3). Однако обратите внимание, что отслеживание результатов событий, связанных с деловыми партнерами, и результатов производственных запросов включает два других сценария использования: `Manage Timelog Records` и `Manage Contact Documents`. Последний будет использован для формирования документов, относящихся к событиям, связанным с деловыми партнерами и производственными запросами (включая содержание сообщений электронной почты).

`Manage Contact Information` должен быть функционально связан с `Maintain Contact Information` (поддержание информации о деловом партнере) и `Record Customer Requirements` (запись требований клиента), приведенными на рис. 7.3. Он имеет дело с характеристиками и требованиями людей и организаций, которые являются предметами интересов CM.

`Manage Contact Documents` позволяет размещать и извлекать любые документы — и структурированные, и неструктурированные, включая сообщения электронной почты, связанные с деловыми партнерами. Документы классифицируются входящими, если они поступают от деловых партнеров, и исходящими, если они отправляются к ним.

7.3. Глоссарий предметной области

Бизнес-глоссарий в таблице 6.1 раздела 6.4 определял основные условия, связанные с оценкой расходов на рекламу. Название *бизнес-глоссарий* показывает, что он содержит словарь, связанный с BOM. Так как BOM определяет DOM, бизнес-глоссарий должен быть расширен, чтобы включить термины предметной области.

Практически разработка информационной системы касается одной *предметной области* единовременно (и имеет дело только с одним приложением единовременно, то есть, с разрабатываемым приложением). Поэтому благоразумно говорить о **глоссарии предметной области**, как о списке терминов в пределах возможностей предметной области.

7.3.1. Глоссарий предметной области для управления деловыми партнерами

Предметная область для управления деловыми партнерами определяет возможности ее глоссария — см. таблицу 7.1. Глоссарий содержит все термины бизнес-глоссария (таблица 6.1) и добавляет термины, связанные с управлением деловыми партнерами. Поскольку глоссарии быстро растут в размере,

глоссарий в таблице 7.1 сортируется. В столбце «Определения и объяснения» ссылки на термины, уже определенные в глоссарии, напечатаны курсивом.

Таблица 7.1. Глоссарий предметной области управления деловыми партнерами

Термин	Определения и объяснения
ad (advertisement) — реклама	Отдельная часть творческого материала, которая может быть радиопередачей, экранизацией, печатным изданием или иначе представленной и демонстрируемой любое число раз. Каждая ее демонстрация информационным агентством известна в AEM-системе как *экземпляр рекламы*
ad instance —экземпляр объявления	Отдельная демонстрация *рекламы*, то есть каждый случай радиопередачи, экранизации или публикации
adlink — рекламная цепочка	Объединение *рекламы*, *изделий*, которые она рекламирует, *рекламодателей*, кто платит за демонстрацию рекламы, и *агентств*, ответственных за заказ таких демонстраций. Рынки сбыта, которые выставили объявление, могут быть определены из описаний *экземпляров рекламы*
advertiser — рекламодатель	*Организация*, которая использует средства информации, чтобы выставить свою творческую продукцию (*организовать рекламу*), продвинуть *изделие* (товар или услугу). Большая часть рекламодателей известна в AEM-системе как «прямые рекламодатели» (Direct Advertiser). Это рекламодатели, которые обходят агентства и места рекламирования в книгах непосредственно с помощью информационного *агентства*
advertiser group — группа рекламодателей	Компания, которая непосредственно или косвенно управляет одной или большим количеством других компаний и заинтересована в рассмотрении накопленных рекламных операций и расходов объединенных компаний. Группы рекламодателей не оплачивают какую-либо рекламу; сами рекламодатели внутри группы делают это. Группы рекламодателей также известны как холдинговые компании (Holding Companies)
agency — агентство	*Организация*, которая занимается планированием рекламы и закупкой средств информации для *рекламодателя*. Цель агентства состоит в том, чтобы оптимизировать расходы на рекламу
agency group — группа агентств	*Организация*, которая непосредственно или косвенно управляет одним или большим количеством других *агентств* и заинтересована в рассмотрении накопленных планов и закупок средств информации для *агентства* или группы агентств. Группы агентств не выполняют никакого планирования или закупки средств информации; агентства внутри группы делают это сами
calendar entry — запись в календарь	Назначение, встреча, событие или другая операция, запланированная в электронном календаре служащего, чтобы ее выполнить. Запись в календарь может быть временной (когда она размещена по дате/времени) или невременной (она представляет собой запись «операций, которые нужно сделать» без указания даты/времени выполнения)
category — категория	Классификация *изделий*, как предусмотрено в AEM, для цели формирования сообщений. Категории могут быть организованы в иерархические структуры категорий и подкатегорий. *Изделия* «категоризированы» на нижнем уровне иерархической структуры
contact — деловой партнер	Человек или *организация*, которые взаимодействуют с AEM или с которыми делают бизнес

customer — клиент	*Деловой партнер*, который получает данные или услуги от AEM и обычно платит за это
employee — служащий	Человек, который нанялся в AEM или имеет другое соглашение для работы с AEM с оплатой или без нее
organization — организация	Бизнес-сущность, с которой AEM имеет дело. Имеются различные типы организаций, включая *рекламодателей*, *агентства*, *рынки сбыта*, *группы рекламодателей*, *группы агентств* и *поставщиков* данных. Организация может быть одного, нескольких этих типов или не входить ни в один из них
outlet — рынок сбыта	*Организация*, которая выставляет *рекламу*. В телевидении и на радио рынок определяется как станция, которая транслирует *объявления*. В кино и наружных средствах информации рынок определяется как компания, которая организует демонстрацию *рекламы*. В области печати рынок определяется как публикация, в которой напечатана *реклама*
product — изделие	Единица изделия или услуги, которая может рекламироваться. Изделия могут быть категоризированы (то есть изделие может принадлежать к категории изделий). *Категории* — классификации изделий, как предусмотрено в AEM. AEM-система поддерживает иерархическую группировку *категорий* с неограниченным числом уровней в иерархии. Изделия могут быть категоризированы только на самом низком уровне иерархии *категорий*
production query — запрос на получение продукции	Просьба представить информацию или сама информация, сформированная производственным штатом для *делового партнера* AEM. Производственный штат включает всех *служащих*, включенных в сбор и обработку данных о рекламных операциях
prospect (prospective customer) — перспективный клиент (предполагаемый клиент)	*Деловой партнер*, который рассматривается коммерческим штатом AEM как потенциальный *клиент*
provider (auxiliary supplier) — поставщик (вспомогательный поставщик)	Организация, которая обеспечивает вспомогательные данные для AEM-системы. Эта информация включает демографические отчеты (разделение людей на группы на основе конкретных атрибутов типа возраста, пола и социально-экономических факторов), обзоры (оцененные и проектируемые средства просмотра, прослушивания и чтения для различных демографических категорий и для различных рынков сбыта), нормы (цены на рекламу у различных рынков сбыта, чтобы определить стоимость объявлений), скидки (оцененные снижения рекламных цен из-за покупательной способности *агентств* при заказе объявлений через указанную среду)
query (to contact) (outmessage) — запрос (для делового партнера) (исходящее сообщение)	Просьба представить информацию, или сама информация, сформированная *служащим* AEM для *делового партнера* AEM. Включает *запросы на получение продукции*

7.4. Модель классов предметной области

Модель классов предметной области определяет классы и отношения между ними. Определение визуально и описательно. Диаграмма классов обеспечивает визуальное представление. Описательное представление состоит из спецификаций различных свойств, поскольку они могут характеризовать

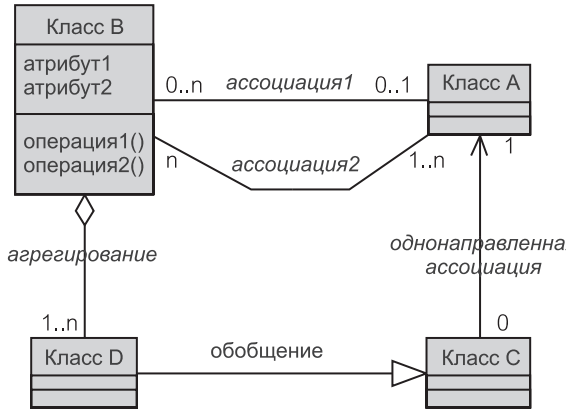


Рис. 7.5. Классы и отношения в модели классов предметной области

классы и отношения. Некоторые из этих свойств не могут эффективно визуализироваться в диаграмме классов.

Рис. 7.5 (то же самое, что и рис. 2.6) — абстрактный пример диаграммы классов. Три вида **отношений** (ассоциации, агрегирования и обобщения), используемые в модели бизнес-классов (раздел 6.5), могут также использоваться и для модели классов предметной области.

Свойства модели, которые нельзя показать визуально в диаграмме классов, но поддерживаемые UML, могут быть записаны в хранилище UML-совместимого средства визуального моделирования. **Хранилище** — это БД инструментального средства визуального моделирования, которое хранит все спецификации модели, включая и диаграммы.

Невизуальные свойства модели могут быть введены через диалоговое окно классификатора (см. раздел 6.3.1). Рис. 7.6 показывает диалоговое окно для класса. Диалоговое окно имеет много закладок. Активная закладка, имеющая название `General` (основная), позволяет описать класс в поле `Documentation` (документация). Содержание поля `Documentation` будет размещено в хранилище как текст, который не будет видим на диаграмме классов.

7.4.1. Классы и атрибуты

Класс представляет множество объектов. Классы в модели классов предметной области отражают *бизнес-концепции*. Модель классов предметной области не определяет *концепции ПО* типа классов для графического пользовательского интерфейса или классов, необходимых в прикладной программе для связи с БД. В этом смысле **классы предметной области** походят на бизнес-сущности модели бизнес-классов (раздел 6.5).

Большинство классов предметной области являются определениями для многократных **экземпляров объекта**. Прикладная программа должна будет *инициализировать* ряд объектов класса предметной области. Например, класс по имени `Product` (изделие) является определяющим шаблоном, по которому будут созданы многие изделия. Это типично для классов, представляющих

Рис. 7.6. Диалоговое окно хранилища

бизнес-объекты. В противоположность этому многие классы, представляющие концепции ПО, являются одноэлементными классами. **Одноэлементный класс** означает, что только один его экземпляр может существовать в пределах программы. Например, класс по имени `DatabaseReader` (устройство чтения БД), вероятно, будет одноэлементным классом.

Ожидается, что модель классов предметной области более детализирована (то есть, это более низкий уровень абстракции), чем модель бизнес-классов. Например, могут быть определены важные атрибуты класса предметной области (но было бы слишком рано определять операции на этой стадии). Обратите внимание, что в этом контексте графическое представление бизнес-сущностей (раздел 6.5.1) даже не имеет отдельных областей для операций и атрибутов.

Атрибут представляет значение данных. Например, `product_name` (имя изделия) — атрибут в классе `Product`. Атрибут имеет имя (типа `product_name`) и **тип данных** (например, *строка* символов). Объектно-ориентированное мышление подразумевает, что типы данных атрибутов — простые типы. **Простой тип данных** означает, что в нем нельзя выделить никакую внутреннюю структуру. Например, `product_name` — простой тип, но `person_name` (имя человека) — нет, если мы хотим различать фамилию, имя и отчество.

Точка зрения объектно-ориентированного подхода и UML заключается в том, что **непростой тип данных** является классом (даже несмотря на тот факт, что класс может содержать операции в дополнение к атрибутам). Это создает некоторую трудность, когда выбираются классы моделирования предметной области. Без квалифицированного использования абстракции разработчик моделей может создать большое количество несущественных классов типа `PersonName` (имя человека).

7.4.2. Отношения классов

Отношение — ясно выраженная связь между классификаторами (то есть, классами в случае модели классов предметной области). При моделировании классов предметной области могут использоваться отношения ассоциации, агрегирования и обобщения (рис. 7.5). Однако рекомендуемый подход состоит в том, чтобы использовать ассоциации свободно, дважды подумать перед использованием агрегирований и всеми силами сопротивляться использованию обобщений.

Хотя **ассоциация** определена на классах (типах), истинный смысл ее заключается в том, что она является отношением между экземплярами этих классов (типов). Следовательно, множественность сторон ассоциации существенна. Рис. 7.5 демонстрирует возможное использование множественностей. **Множественность** определяет, сколько экземпляров одного класса относится к одному экземпляру другого класса. Множественность должна быть определена на обеих сторонах ассоциации. Присутствие нуля в множественности (`0..n` или `0..1`) указывает, что **участие** экземпляра класса в ассоциации необязательно.

Агрегирование в UML — специальный вид ассоциации, где экземпляр одного класса (целое; **класс-супермножество**) содержит экземпляры другого класса (часть; **класс-подмножество**). В большинстве случаев модель классов предметной области будет достаточно выразительна и без использования агрегирования. Решение использовать агрегирование в моделировании предметной области может быть принято в случае исключительных обстоятельств, когда требуется такая специализированная форма ассоциации в UML.

Обобщение — отношение классификаторов, то есть, классов (типов) в модели классов предметной области. Каждый экземпляр более определенного класса (**подкласса**) является также экземпляром более общего класса (**суперкласса**). Следовательно, это не отношение экземпляров данных классов. Согласно определению обобщения, множественность к нему не применяется.

Использование обобщения в моделировании классов предметной области представляет риск неверного истолкования реальной природы отношений в модели (то есть, как связаны экземпляры объектов). Обобщение должно использоваться только тогда, когда окончательное визуальное упрощение графической модели перевешивает риск беспорядка при интерпретации модели. (Умное использование обобщения может радикально уменьшить число других отношений в модели.)

Рис. 7.5 показывает возможность (весьма необычную), где множественность для `Класса C` в ассоциации с `Классом A` — `0` (это соответствует `0..0`, то есть, минимум 0 и максимум 0). Это возможно, потому что ассоциация однонаправлена. Переход допускается только от `Класса C` к `Классу A`, но не наоборот. Следовательно, `Класс A` не связан с `Классом C`.

Множественность агрегирования обычно определяется только на одной стороне агрегирования. Целое (суперкласс) содержит часть (подкласс). Это означает, что экземпляр целого может содержать нуль, один или несколько экземпляров части. По смыслу, экземпляр части обычно содержится в одном экземпляре целого (пожалуй, он может не содержаться вообще, и в этом случае множественность была бы `0..1`).

7.4.3. Модель классов для управления деловыми партнерами

Рекомендуется разрабатывать модель сценариев использования и модель классов параллельно. К сожалению, учебник не может показать процесс создания вещей — он может только описывать изделия процесса. Два главных изделия DOM — модель сценариев использования предметной области и модель классов предметной области. Так как модель сценариев использования предметной области была описана первой (раздел 7.2), мы можем использовать ее, чтобы получить соответствующую модель классов предметной области.

Чтобы прочувствовать этот процесс, читателю рекомендуется заново прочитать раздел 7.2.3 и попытаться сделать набросок диаграммы классов, которая удовлетворяла бы информационным потребностям сценариев использования, описанных в этом разделе. Без сомнения, эксперимент, подобный этому, откроет ящик Пандоры, полный вопросов, требующих разъяснения на обеих сторонах спектра моделирования — в модели сценариев использования и в модели классов. Именно по этой причине эти две модели должны быть созданы итеративно и синхронно.

Рис. 7.7 — диаграмма классов предметной области, соответствующая информационным потребностям модели сценариев использования предметной области в разделе 7.2.3. Диаграмма показывает главные отношения и основные атрибуты классов. Все отношения — ассоциации. Типы данных атрибутов не определены, но некоторые атрибуты имеют явно непримитивные типы данных. Эти атрибуты станут классами по праву (раздел 7.4.1), но эта возможность игнорируется ради простоты.

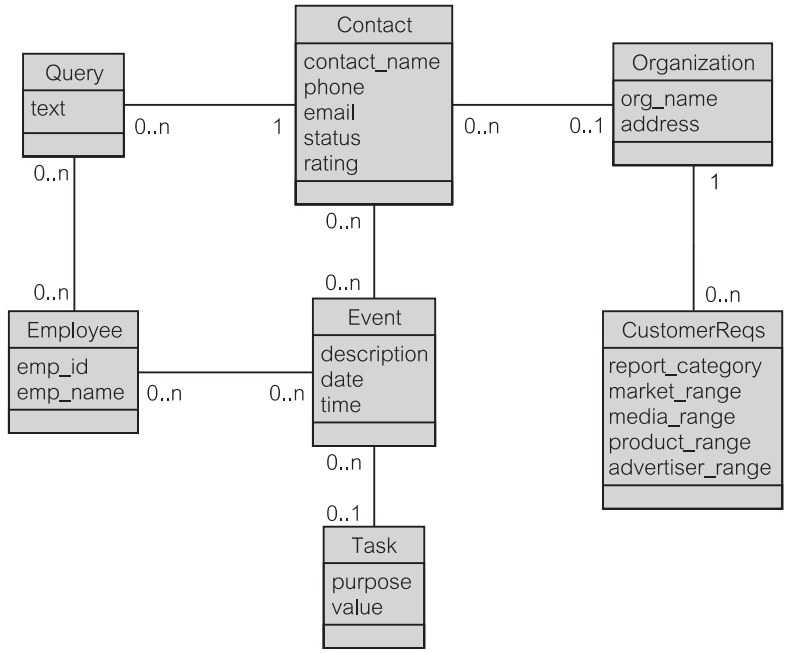


Рис. 7.7. Диаграмма классов для СМ

Contact (деловой партнер) — это человек, который представляет Organization (организацию) или действует как индивидуум с деловыми связями к AEM (следовательно, множественность — 0..1). Атрибут status (состояние) отражает состояние переговоров с клиентом, типа того, что деловой партнер является перспективным (раздел 7.3.1). Атрибут rating (рейтинг) — сравнительная величина ранжирования, которую AEM в настоящее время устанавливает деловому партнеру (она управляется сценарием использования Maintain Contact Information — поддержание информации о деловом партнере).

Модель не трансформирует Contact в подклассы, но дает понимание, что Contact представляет любого делового партнера AEM — клиента, рынок сбыта, поставщика, перспективного клиента и т. д. (раздел 7.3.1). Изображение подклассов (отношение обобщения) для Contact существенно не улучшит (если вообще улучшит) семантику модели, но сделает модель более сложной для исследования.

Contact связан со многими Event — событиями (сравните со сценарием использования Track Contact Events — отслеживание событий делового партнера) и многими Query — запросами (сравните со сценарием использования Handle Production Queries — управление производственными запросами), а также с Employee (служащий). Таким образом, деловые партнеры и служащие косвенно связаны через события и запросы.

Event могут быть объединены в Task (задача), но это не следует делать. Event может быть автономным, возможно, обеспечивающим «однократное» взаимодействие между Employee и Contact. Ассоциация между Event и Task поддерживает сценарий использования Track Contact Events.

CustomerReqs (требования клиента) поддерживает сценарий использования Record Customer Requirements (запись требований клиента), и это питает в конечном счете функцию управления контрактами клиентов — функцию, которая является внешней по отношению к управлению деловыми партнерами DOM. Хотя переговоры о требованиях клиента проводятся с деловыми партнерами, CustomerReqs связан с Organization, которую представляет Contact.

Атрибуты CustomerReqs — непростых типов данных. Они станут классами в уточненной модели классов. Создание отдельных классов в настоящий момент изменило бы полный «баланс» модели в пользу одного слишком увеличенного сценария использования Record Customer Requirements (один из пяти сценариев использования на рис. 7.3).

7.4.4. Альтернативная модель классов для управления деловыми партнерами

Рис. 7.8 представляет модель классов, которая соответствует альтернативной модели сценариев использования, приведенной в разделе 7.2.4. По сравнению с моделью на рис. 7.7 альтернативная модель менее конкретная, поскольку связана с управлением деловыми партнерами в AEM. Модель практически может служить структурой для любой CM-системы, а не только для AEM. Эта структура должна быть специализирована и расширена в следующих итерациях проекта, отображая специфические особенности AEM.

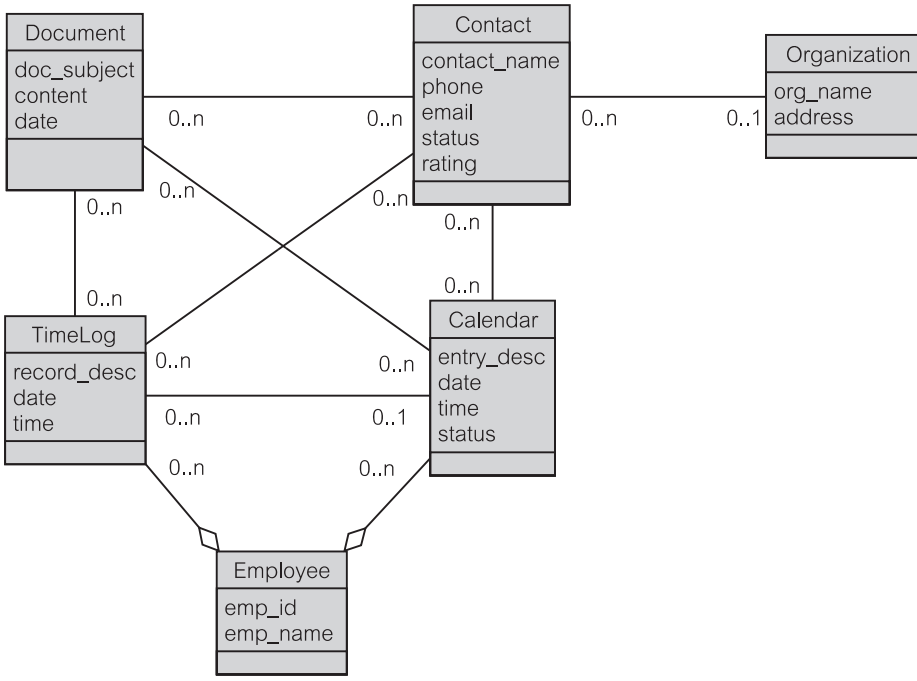


Рис. 7.8. Альтернативная диаграмма классов для CM

Модель выделяет персональную информацию о служащих (в классе `Employee` — служащий) из планов и записей рабочих операций служащих (в классах `Calendar` — календарь и `TimeLog` — временной протокол). `Employee` содержит несколько `Calendar` и `TimeLog` (смоделированных отношениями агрегирования). Причина, почему здесь используется «несколько», очевидна из рассмотрения атрибутов `Calendar` и `TimeLog`. `Calendar` содержит тексты и детали записей календаря (`entry_desc`). `TimeLog` хранит описания и детали записей временного протокола (`record_desc`).

Запись `TimeLog` может быть результатом действия служащего в соответствии с записью в `Calendar`. Следовательно, на одной стороне отношения между `TimeLog` и `Calendar` должна быть множественность `0..1`. Запись `TimeLog` и запись `Calendar` могут быть связаны множественностью `0..n` с `Contact` — деловой партнер (некоторые служебные обязанности служащих, типа участия во встречах, могут включать группу деловых партнеров).

Размещение и обработка документов играют центральную роль в модели. `Document` (документ) может поступить от (или быть передан к) одного(му) или большего(му) количества(у) `Contact`, он может быть адресован одной или большему количеству записей в `Calendar` и может быть упомянут в одной или большем количестве записей `TimeLog`.

`Contact` может быть, а может и не быть связан с `Organization`. `Organization` может существовать, даже если ее `Contact` не известны в системе. Важный вывод (и предположение одновременно) заключается в том,

что документы, записи временных протоколов и записи календарей могут быть связаны (косвенно) с `Organization`, если они связаны, по крайней мере, с одним `Contact`.

Резюме

1. *Объектная модель предметной области* (DOM) — модель одной прикладной области (предметной области) предприятия. Это одна из многих возможных моделей в рамках *модели бизнес-объектов* (BOM). Обычно DOM является UML-моделью анализа для одного бизнес-сценария использования BOM.
2. *Предметная область* определяет цели *разрабатываемого приложения*. Как правило, *итерация* разработки формирует один или пару сценариев использования, определенных в рамках DOM.
3. Предметная область, взятая в качестве сценария использования книги — CM (Contact Management — управление деловыми партнерами). Разрабатываемое приложение, EM (Email Management — управление электронной почтой), — является сценарием использования с DOM для CM.
4. CM представляет систему регистрации описательной *структурированной информации* относительно обменов данными между предприятием и его деловыми партнерами (клиентами, поставщиками и т. д.), а также хранения и управления неструктурированными документами, связанными с этими обменами связью.
5. *Модель сценариев использования предметной области* — UML-диаграмма сценариев использования и связанных с ней документов сценария использования, определенных для предметной области.
6. *Диаграмма сценариев использования* — графическое описание акторов, сценариев использования и отношений между ними.
7. *Актор* — роль, которую кто-то или что-то играет по отношению к сценарию использования. *Сценарий использования* представляет конкретную часть функциональных возможностей.
8. Актор связывается со сценарием использования через отношение «`communicate`» (связывание). Сценарии использования могут быть связаны ассоциацией и отношениями обобщения, а также двумя стереотипными отношениями, называемыми «`include`» (включение) и «`extend`» (расширение).
9. *Глоссарий предметной области* расширяет бизнес-глоссарий, добавляя к нему термины и определения, специфичные для предметной области.
10. *Модель классов предметной области* определяет классы в рамках предметной области и отношения между этими классами. Определение является и визуальным (диаграмма классов предметной области), и описательным (текстовые спецификации для классов).
11. Классы предметной области определяют *бизнес-объекты* (в противоположность *объектам ПО*). Как правило, класс предметной области представляет несколько экземпляров объекта. *Одноэлементные классы*, кото-

рые могут инициализировать только один объект, встречаются редко среди классов предметной области (но весьма часты для других классов ПО).

12. *Атрибут* представляет значения данных. *Тип атрибута* может быть простым или непростым. Непростой тип определяет класс.
13. Имеются три главных вида *отношений*: ассоциации, агрегирования и обобщения. *Множественность* и *участие* — важные свойства ассоциаций. Эти свойства применяются также и к агрегированию, но не к обобщению.

Ключевые термины

Обзорные вопросы

Вопросы для обсуждения

1. Объясните, как BOM и DOM относятся друг к другу.
2. Что такое типичная цель разрабатываемого приложения по отношению к DOM?
3. Что такое структурированная и неструктурированная информация?
4. Каково значение отношения «`communicate`» (связывание)?
5. Объясните различия между дополнительным сценарием использования и сценарием использования расширения.
6. Что такое хранилище?

7. Что считается бизнес-объектами и объектами ПО? Как эти две концепции относятся к моделированию объектов предметной области?
8. Что такое класс предметной области?
9. Что такое одноэлементный класс? Можно ли считать класс предметной области одноэлементным классом?
10. Объясните роль непростых типов в моделировании объектов предметной области.
11. Как UML выражает множественность и участие? Являются ли эти две концепции семантически зависимыми или независимыми? Является ли UML способом выражения множественности и участия, связанным с семантикой этих двух концепций?
12. Сравните полезность ассоциации и агрегирования в объектном моделировании предметной области.
13. Что такое обобщение? Как оно должно использоваться в объектном моделировании предметной области?

Вопросы учебного примера

1. Рассмотрим диаграмму сценариев использования на рис. 7.3. Предположим, что вы хотели бы сделать модель более явной, показав отношения между сценариями использования. Попытайтесь сделать это. Каково ваше заключение? Имеет ли смысл включать моделирование отношений в эту диаграмму?
2. Что такое перспективный клиент в CM-терминологии?
3. Проанализируйте модель классов в разделе 7.4.3 и диаграмму на рис. 7.7. Можно ли согласно модели AEM войти в договорные отношения с частным лицом (в противоположность организации)?

Примеры задач

Упражнения учебного примера

1. Рассмотрим модель сценариев использования для CM из раздела 7.2.3 и соответствующую диаграмму сценариев использования на рис. 7.3. Предположим, что текущая итерация разработки подчеркивает следующие аспекты CM:
   а) Имеется необходимость в отдельном отслеживании событий, касающихся договоров, которые AEM имеет с существующими клиентами. Многие события, связанные с контрактами, поступают от клиентов, сообщающих AEM о своих новых требованиях относительно рекламной информации, или от AEM-маркетинга новых услуг клиентам согласно существующим контрактам. Они также могут быть результатом обычной или специальной связи с деловыми партнером.
   б) Глоссарий в таблице 7.1 утверждает, что деловой партнер — человек или организация, с которыми контактирует AEM или с которыми делают бизнес. Необходимо разъяснить корреляцию между `PersonContact` (парт-

нер-личность) и `OrganizationContact` (партнер-организация). Предположим, что анализ природы этой корреляции показал:

1) Часто имеется много индивидуумов (`PersonContacts`), которых АЕМ использует для связи с одной организацией (`OrganizationContact`).
2) Некоторые связи с `OrganizationContact` осуществляются через индивидуумов, которые неизвестны для АЕМ. Это может быть, например, один из многих секретарей, и информация о них не зарегистрирована в БД АЕМ. Однако вероятным результатом связи с неизвестным индивидуумом может быть последующий контакт с `PersonContact` в пределах той же организации (например, когда секретарь договаривается о встрече АЕМ с финансовым чиновником).

Разработайте вариант диаграммы сценариев использования на рис. 7.3, который подчеркивает два вышеупомянутых аспекта СМ. Сохраните существующие сценарии использования. Не показывайте никаких акторов. Добавьте к модели только новые сценарии использования и установите необходимые отношения.

2. Рассмотрите модель классов для СМ в разделе 7.4.3 и соответствующую диаграмму классов на рис. 7.7. Учтите дополнительные и специфические требования, рассмотренные выше в упражнении 1. Независимо от того, пытались ли вы решить упражнение 1, создайте вариант диаграммы классов на рис. 7.7, который моделирует новые требования.

Небольшой проект — временной протокол

Управление деловыми партнерами — это не только непосредственное управление деловыми партнерами. Оно также связано с управлением служащими, участвующими в управлении деловыми партнерами. Диаграмма сценария использования на рис. 7.4 обеспечивает этот момент включением двух сценариев использования, связанных только с актором `Employee` (служащий). Эти сценарии использования: `Manage Calendar Entries` (управление записями календаря) и `Manage Timelog Records` (управление записями временного протокола). Данный небольшой проект касается последнего ([61] содержит учебный пример, который исследует типичные функциональные возможности управления электронными календарями служащих.)

Временной протокол (Time logging — TL) представляет собой средство, которое позволяет служащим записывать время работы с различными задачами в течение указанного периода времени. В случае СМ протоколирование времени включает время, потраченное на обеспечение управления деловыми партнерами. Оно включает обмен сообщениями по телефону, факсом, электронной почтой, курьером и стандартной почтой. Сюда же включаются встречи и конференции. Протоколирование включает также потраченное время на подготовку исходящих документов, писем, записок, контрактов и т. д., а также потраченное время на чтение и анализ любых поступающих материалов. Наконец, оно включает время, связанное с поддержанием характеристик делового партнера типа изменений адреса, отслеживания ответной корреспонденции и т. д.

Фундаментальное требование применимости функции протоколирования времени заключается в том, что она должна быть с минимумом ограничений.

Другими словами, функция должна быть прозрачной, совпадающей и параллельной выполнению задач управления деловыми партнерами. Без этого «непринужденного использования» протоколирование времени не будет работать — или служащие не будут стараться использовать временную информацию, или они не будут делать это в достаточной степени точно и детально.

Протоколирование времени в СМ не представляет собой «старшего брата», наблюдающего, что делают служащие, или оценивающего выполнение функций служащими. Протоколирование времени — что-то вроде определения времени и трудозатрат, потраченных на работу с различными деловыми партнерами, оценки влияния этих партнеров на АЕМ-бизнес, оценки деловых партнеров, отношений с клиентами, отслеживания событий, ведущих к успешным контрактам, своевременного ответа на жалобы и т. д. Короче говоря, протоколирование времени — это вроде бизнес-анализа, тактических решений и стратегического планирования в отношении функции управления деловыми партнерами.

Протоколирование времени не уникально для АЕМ. Любая организация может извлекать выгоду из приложения, поддерживающего эту функцию. Действительно, имеются коммерческие инструментальные средства ПО для протоколирования времени. Одно такое инструментальное средство поставляется фирмой Responsive Software [87]. Инструментальное средство может быть настроено, чтобы обеспечить легкий ввод временной информации для конкретных проектов и операций. Альтернативно это и подобные инструментальные средства могут служить как источник идей для разработки пользовательского интерфейса проекта, целью которого является разработка собственной функции протоколирования времени.

Для целей этого небольшого проекта функция протоколирования времени является сценарием использования предметной области СМ (рис. 7.4). Сценарий использования позволяет служащему работать с записями времени относительно *задач* и *операций* (называемых также *событиями*), выполненных в отношении деловых партнеров. Сценарий использования определяет из БД СМ списки деловых партнеров и предопределенных задач. Все служащие, использующие записи времени, могут обратиться к этим спискам. Индивидуально каждый служащий может создавать в БД СМ свой собственный список предопределенных *операций*. Каждый раз запись может содержать все более длинное описание текста относительно характера задачи/операции.

Запись времени всегда имеет отношение к одному служащему и одному деловому партнеру. При этом условии запись времени может быть введена для:

- предопределенной задачи и предопределенной операции в пределах этой задачи;
- предопределенной задачи без зарегистрированной операции;
- определенной служащим операции без зарегистрированной задачи (приложение позволяет создать новую определенную служащим операцию, если операция не была определена прежде и поэтому не доступна из выпадающего списка).

Обращение к записи времени включает начальную дату/время и конечную дату/время задачи/операции. Сценарий использования автоматически вычис-

ляет продолжительность задачи/операции. Продолжительность может быть меньше, чем разность между конечным и начальным временем, если служащий брал некоторые перерывы (паузы) между начальным и конечным временем. Служащий может явно вводить продолжительность перерывов в пределах записи времени.

Предыдущие временные протоколы можно просмотреть и распечатать. Они могут также сортироваться в соответствии с различными критериями, в частности, в соответствии с датой/временем, деловым партнером и задачей. Возможно настроенное отображение, типа того, когда некоторые записи фильтруются. Критерии фильтрации такие же, как и критерии сортировки. Существующие записи могут быть изменены или удалены в соответствии с некоторыми бизнес-правилами, типа таких, что изменения не допускаются для записей, используемых в предыдущих составлениях счетов, бухгалтерском учете или отчетном периоде.

Упражнения

1. На основе спецификаций небольшого проекта разработайте модель сценариев использования, которая выделяет подсценарии использования сценария использования `Manage Timelog Records` (управление записями временных протоколов) из рис. 7.4. Поскольку `Employee` (служащий) — единственный актор, нет никакой необходимости показывать его в диаграмме.
2. На основе спецификаций небольшого проекта расширьте диаграмму классов предметной области на рис. 7.8. Добавьте новые классы предметной области и свяжите их с существующими классами.

Глава

8 Итерация 1. Требования и объектная модель

Эта книга использует итеративный подход учебного примера к разработке информационных систем. Основной учебный пример — **управление электронной почтой** (Email Management — EM). EM — часть управления деловыми партнерами (Contact Management — CM), которое в свою очередь является частью оценки расходов на рекламу (Advertising Expenditure Measurement — AEM) (рис. 7.1). Соответственно, EM можно рассматривать как сценарий использования CM.

Рис. 8.1 — расширение модели сценариев использования, представленной на рис. 7.4. `Manage Email` (управление электронной почтой) — *сценарий использования расширения* (раздел 7.2.2) для трех базовых сценариев использования: `Manage Calendar Entries` (управление записями календаря),

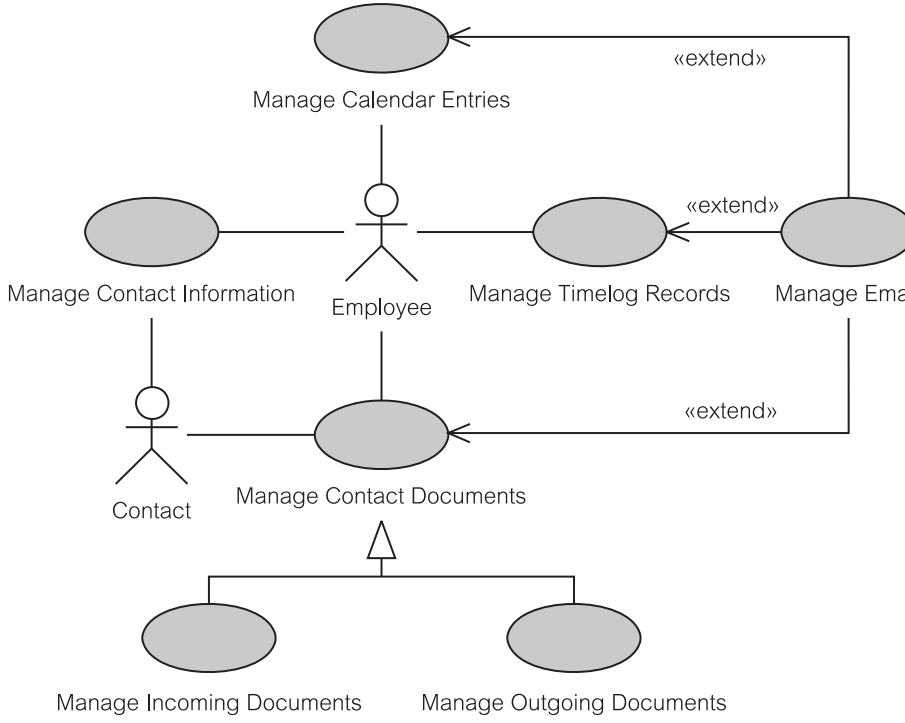


Рис. 8.1. Управление электронной почтой как CM-расширение сценариев использования

Manage Timelog Records (управление записями временного протокола) и Manage Contact Documents (управление документами делового партнера).

Сама электронная почта представляет документ, отправленный деловому партнеру или полученный от него. Дополнительно электронная почта может содержать приложения к документам. Следовательно, Manage Email расширяет Manage Contact Documents (управление документами деловых партнеров). Посылка сообщения электронной почтой может быть запланирована как операция работы служащего, и поэтому помещена как запись в календарь под управлением Manage Calendar Entries (управление записями в календарь). Факт посылки и получения ответа по электронной почте влечет за собой запись работы, которая должна быть помещена во временной протокол служащего. Поэтому Manage Email расширяет Manage Timelog Records (управление записями временного протокола).

Хотя это и выходит за пределы учебного примера книги, управление электронной почтой может иметь более широкое применение как система для **маркетинга с помощью электронной почты** (Email Marketing). Маркетинг с помощью электронной почты — нечто вроде предоставления по электронной почте деловым партнерам (потенциальным клиентам) маркетинговой информации относительно изделий или услуг. Как и в учебном примере EM, маркетинг электронной почты хранит информацию в БД относительно деловых партнеров и их электронных адресов, форматирует исходящие сообщения, основанные на информации, также хранящейся в БД, автоматически посылает по электронной почте эти сообщения и соответственно корректирует БД.

8.1. Модель сценариев использования

По педагогическим соображениям эта и последующие итерации учебного примера используют предположения и упрощения. Итерация 1 предполагает, что сообщения к деловым партнерам уже размещены в БД EM. Итерация 1 системы предназначена для создания и передачи по электронной почте сообщений и для обновления БД данных после успешной передачи.

Рис. 8.2 представляет модель сценариев использования, предназначенную для представления цели, намеченных функциональных возможностей, предположений и упрощений итерации 1. Сценарий использования Store Messages to Contacts (размещение сообщений для деловых партнеров) — вне возможностей итерации 1.

Цель Store Messages to Contacts (размещение сообщений для деловых партнеров) — разместить в Production Database (БД производства) тему и текст сообщений по электронной почте к деловым партнерам и поручить отправку этих сообщений соответствующим Customer Department Employee (служащие отдела работы с клиентами). Обычно этот сценарий использования активизируется во время сбора и проверки данных о рекламных операциях и при подготовке сообщений об операциях для клиентов. Итерация 1 предполагает, что такие сообщения уже существуют в Production Database.

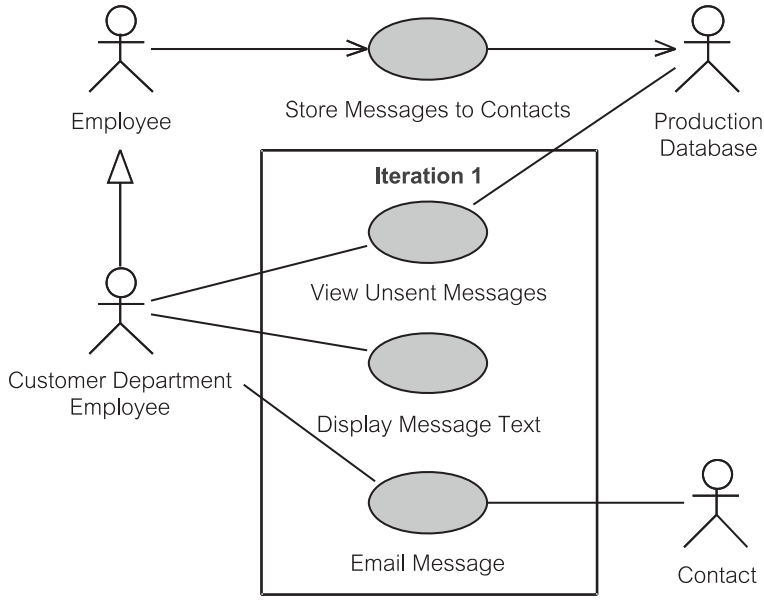


Рис. 8.2. Диаграмма сценариев использования для итерации 1

`View Unsent Messages` (просмотр непосланных сообщений) отвечает за показ списка, содержащего основную информацию о сообщениях, сохраняемых в `Production Database` и запланированных для отправки по электронной почте из `Customer Department Employee`. `Customer Department Employee` может пожелать отобразить полный текст сообщения (`Display Message Text` — отображение текста сообщения) перед принятием решения послать его по электронной почте (`Email Message` — сообщение электронной почты).

Как только сообщение будет успешно послано, `Email Message` устанавливает флаг, что это произошло. Данное действие не очевидно в диаграмме сценариев использования, потому что модель сама не фиксирует, как и где будет установлен «флаг отсылки». Возможно, например, что `Production Database` используется только сценарием использования `View Unsent Messages`, который читает сообщения и загружает их в БД, внутреннюю по отношению к ЕМ. Внутренняя БД не может быть актором в ЕМ-модели.

8.2. Документ сценария использования

Графическое представление модели сценария использования (то есть диаграмма сценария использования) служит важной цели визуализации («изображение стоит тысячи слов»), но реальная выгода моделирования сценария использования заключена в его текстовых описаниях. Текстовое описание сценария использования иногда называется просто **документом сценария использования**.

Документ сценария использования пишется согласно формату, предопределенному разработчиками для проекта. Существуют различные шаблоны формата [55]. Шаблоны делят документ на разделы и представляют другие соглашения написания. Документы сценариев использования являются требованиями пользователей. Так как требования пользователей связаны и зависят друг от друга, соглашение написания документа сценария использования должно предложить технологию компоновки различных частей документа.

Шаблон формата, используемый в этой книге, представлен в таблице 8.1. На практике шаблон будет также содержать «заголовочную» информацию — автора, дату создания, версию, состояние разработки, связанный документ контрольного примера и т. д.

Таблица 8.1. Документ сценария использования

Краткое описание	Расширенное описание
Акторы (Actors)	Идентифицированы из диаграммы сценариев использования и определены в хранилище диаграмм. Если акторов немного и они очевидны, нет никакой необходимости повторять в документе информацию диаграммы и хранилища
Предусловия (Preconditions)	Состояние условий, которые должны быть выполнены прежде, чем сценарий использования может выполнить свой контракт (свои задачи). Если предусловия не будут выполнены, сценарий использования может привести к ошибке, но это не означает, что сценарий использования должен проверять предусловия перед тем, как он «выстрелит»
Постусловия (Postconditions)	Набор условий, обязательно выполняемых после завершения сценария использования. Постусловия определяют состояние системы, когда сценарий использования заканчивает выполнение удовлетворительно и когда он вынужден исполнить альтернативный набор операций
Основной поток (Basic flow)	Списки и краткие определения основной последовательности логики выполнения сценария использования — его главные шаги. Детальное определение каждого шага отложено до описания *подпотоков*. Основной поток определяет «удачный путь» выполнения сценария [55], то есть он игнорирует любые исключительные ситуации или возможные ошибки
Подпотоки (Subflows)	Детально определяют шаги основного потока. Они также предполагают «удачный путь» выполнения сценария. Иногда в литературе (например, в [55]) они называются *расширениями* или *альтернативными потоками*
Потоки исключений (Exceptions flows)	Определяют поток нарушений логики выполнения, вызванный исключениями и ошибками. Если исключение представляет неожидаемый результат выполнения сценария использования, потоки исключений определяют «неудачные пути» сценария. В литературе также называются *альтернативными потоками* (например, в [61], [55])

8.2.1. Краткое описание, предусловия и постусловия

В последующем предполагается, что документ сценария использования имеет дело с итерацией 1 учебного примера EM; то есть, итерация 1 сценария использования `Manage Email` — управление электронной почтой (рис. 8.1) является целью документа.

Краткое описание

Итерация 1 сценария использования `Manage Email` позволяет служащему отображать и посылать сообщения деловым партнерам по электронной почте. Она имеет дело только с сообщениями, уже размещенными в БД. Одновременно может быть передано по электронной почте только одно сообщение.

Предусловия

1. Служащий находится в отделе работы с клиентами или как-то иначе уполномочен системным администратором обеспечивать EM-приложение.
2. БД EM содержит сообщения, которые нужно послать деловым партнерам по электронной почте.
3. Служащий связан с сервером электронной почты и уполномочен быть пользователем БД.

Постусловия

1. Программа обновила БД EM, чтобы отразить любую успешную передачу сообщений электронной почты.
2. БД EM осталась в неповрежденном состоянии, если произошли какое-либо исключение или ошибка.
3. Как только служащий покинет приложение, консольное окно закрывается.

8.2.2. Основной поток

Шаги последовательности выполнения сценария использования известны как **поток событий**. Поток событий делится на основной поток, подпотоки и потоки исключений. Здесь рассматривается **основной поток** итерации 1 сценария использования `Manage Email` (управление электронной почтой). Поток событий может включать эскизы интерфейса пользователь–компьютер. В итерации 1 интерфейс — текстовое консольное окно (то есть, не GUI).

Формирование эскизов связи компьютера пользователя с документами сценария использования — спорная тема. Некоторые эксперты доказывают, что документы сценария использования должны избегать решений проекта/реализации, связанных с видом и поведением интерфейсов пользователь–компьютер. Однако другая школа взглядов утверждает, что интерфейс пользователь–компьютер является фундаментальным требованием пользователя, которое должно рассматриваться на ранних стадиях процесса разработки. Интерфейс пользователь–компьютер не только товарный продукт, но также служит и важной средой связи в процессе сбора и разъяснения требований пользователя.

Основной поток

Функционирование сценария использования начинается, когда служащий желает просмотреть и отправить по электронной почте сообщения деловым партнерам.

Система отображает информационное сообщение и запрашивает у служащего имя пользователя (username) и пароль (password) — рис. 8.3.

```
            EMAIL MANAGEMENT SYSTEM - ITERATION !

Please enter your username: lburton
Please enter your password: psswd
```

Рис. 8.3. Эскиз 1 интерфейса пользователь–компьютер

- Система пытается соединить служащего с БД ЕМ.
- После успешной связи приложение отображает список меню, содержащий возможные опции, которые может запросить служащий. В меню на рис. 8.4 имеются четыре последовательно пронумерованные опции:
  1. Просмотреть непосланные сообщения (View Unsent Messages) деловым партнерам (см. ниже «S1 — View Unsent Messages»);
  2. Отобразить текст сообщения (Display Text of a Message), далее задается идентификатор сообщения (`message_id`) (см. ниже «S2 — Display Text of a Message»);
  3. Переслать сообщение по электронной почте (Email a Message), заданное идентификатором `message_id` (см. ниже «S3 — Email a Message»);
  4. Завершить программу (Quit this Program).

Если служащий выбирает выход из ЕМ-приложения, печатая цифру 4, сценарий использования завершает работу.

8.2.3. Подпотоки

Здесь определяются **подпотоки**, упомянутые в основном потоке. Подпотоки нумеруются для простоты ссылки из других мест документа сценариев использования. Соглашение, используемое в этой книге, идентифицирует подпоток символом S, за которым следует последовательное число (то есть, S1, S2 и т. д.). Соглашение твердо придерживается широко известной практики (например, [83]).

```
            EMAIL MANAGEMENT SYSTEM - ITERATION !

Please enter your username: lburton
Please enter your password: psswd

  1. View Unsent Messages
  2. Display Text of a Message
  3. Email a Message
  4. Quit this Program

Please write an option number [1, 2, 3 or 4]:
```

Рис. 8.4. Эскиз 2 интерфейса пользователь–компьютер

```
            EMAIL MANAGEMENT SYSTEM - ITERATION !

Please enter your username: lburton
Please enter your password: psswd

  1. View Unsent Messages
  2. Display Text of a Message
  3. Email a Message
  4. Quit this Program

Please choose an option [1, 2, 3 or 4]: 1

There are 2 unsent query message(s), as below:

Message ID: 14
DateCreated: 2002-06-21
Message Subject: Product missing
Contact Name: Pablo Romero
Organization: SBS Sydney

Message ID: 19
DateCreated: 2002-06-21
Message Subject: Log incomplete
Contact Name: Dorothy Norris
Organization: ABC Radio

  1. View Unsent Messages
  2. Display Text of a Message
  3. Email a Message
  4. Quit this Program

Please choose an option [1, 2, 3 or 4]:
```

Рис. 8.5. Эскиз 3 интерфейса пользователь–компьютер

S1 — View Unsent Messages (просмотр непосланных сообщений)

Информация, показанная в консольном окне, аналогична примеру на рис. 8.5. Список меню представляется после того, как будет показано последнее непосланное сообщение.

S2 — Display Text of a Message (отображение текста сообщения)

Служащий должен ввести `message_id` перед тем, как будет показан текст этого сообщения. Текст сообщения отображается так, как приведено на рис. 8.6. Список меню заново показывается ниже текста сообщения.

S3 — Email a Message (передача сообщения по электронной почте)

Служащий должен определить, с каким `message_id` должно быть передано сообщение по электронной почте. Он печатает `message_id` сообщения, которое посылается по электронной почте, после чего БД обновляется. Инфор-

```
  1. View Unsent Messages
  2. Display Text of a Message
  3. Email a Message
  4. Quit this Program

Please choose an option [1, 2, 3 or 4]: 2

Please enter the message id: 14

Message ID: 14
DateCreated: 2002-06-21
Contact Name: Pablo Romero
Organization: SBS Sydney
Message Subject: Product missing
Message Text:
The Product name for your ad CI223375XY is missing. Can you
please supply it?

  1. View Unsent Messages
  2. Display Text of a Message
  3. Email a Message
  4. Quit this Program

Please choose an option [1, 2, 3 or 4]:
```

Рис. 8.6. Эскиз 4 интерфейса пользователь–компьютер

мационное сообщение отображается в консольном окне после его успешной передачи, как показано на рис. 8.7. Список меню заново показывается после того, как сообщение было послано.

```
  1. View Unsent Messages
  2. Display Text of a Message
  3. Email a Message
  4. Quit this Program

Please choose an option [1, 2, 3 or 4]: 3

Please enter the message id you want to email: 14

The message Id = 14 has been emailed.
Date Sent: 2002-06-22

  1. View Unsent Messages
  2. Display Text of a Message
  3. Email a Message
  4. Quit this Program

Please choose an option [1, 2, 3 or 4]:
```

Рис. 8.7. Эскиз 5 интерфейса пользователь–компьютер

8.2.4. Потоки исключений

Потоки исключений описывают потоки действий, которые будут предприняты, когда возникнет исключение. Исключение может быть условием ошибки, но это не обязательно.

Подобно подпотокам, потоки исключений нумеруются для простоты ссылки из других мест документа сценария использования. Соглашение, используемое в этой книге, идентифицирует поток исключений символом E, за которым идет последовательный номер (то есть, E1, E2 и т. д.).

E1 — Incorrect username or password (неправильное имя пользователя или неправильный пароль)

Если в основном потоке актор вводит неправильное имя пользователя или неправильный пароль (рис. 8.3), система выводит сообщение об ошибке. Система разрешает актору повторно ввести имя пользователя и пароль, либо покинуть приложение. Актору дают три возможности, чтобы ввести правильные имя пользователя и пароль. Если все три раза будут неудачны, система отменяет регистрацию, и сценарий использования заканчивается.

E2 — Incorrect option (неправильная опция)

Если в основном потоке актор выберет неправильную опцию (рис. 8.4), система игнорирует введенное значение и заново показывает список опций. Если будут подряд три неудачных выбора, система прекращает работу с актором и повторно начинает сценарий использования (запрашивая регистрационное имя — рис. 8.3).

E3 — Too many messages (слишком много сообщений)

Если в подпотоке S1 число непосланных сообщений, запланированных актору, превышает заранее установленное число сообщений, которые могут рассматриваться одновременно, система отображает информационное сообщение о том, что в БД имеется большее количество сообщений, чем допустимо. Информационное сообщение отображается после того, как актору показывается заранее установленное число непосланных сообщений, и перед тем, как будет заново показан список меню.

E4 — Email could not be sent (сообщение электронной почты не может быть послано)

Если в подпотоке S3 почтовый сервер возвращает ошибку о том, что сообщение электронной почты не могло быть послано, система выдает актору информацию, что сообщение не было послано, и сценарий использования продолжается, показывая список меню.

8.3. Концептуальные классы

Объектная модель для BOM (глава 6) и для DOM (глава 7) определяла классы для соответствующих спецификаций сценариев использования. Классы назывались бизнес-сущностями в BOM (раздел 6.5.1) и классами предметной области в DOM (раздел 7.4.1). Такие классы часто упоминаются как концептуальные классы [55].

Концептуальный класс — определение для множества бизнес-объектов, то есть объектов, которые имеют бизнес-значение для организации. В реализованной системе экземпляры (объекты) концептуального класса размещаются в БД (типа `Production Database` — БД производства, смоделированной как актор на рис. 8.2). В этом смысле концептуальные классы представляют *объекты БД*.

Прикладная программа при необходимости имеет доступ к объектам БД и создает их копии в своей памяти. Таким образом, понятие концептуального класса имеет представление и в БД, и в прикладной программе. После того как концептуальные классы реализованы в программе, они становятся лишь одной категории — **прикладных классов**. Все прикладные классы для итерации 1 определяются в главе 11. Начальная объектная модель, обсужденная в этой главе, определяет только концептуальные классы.

Рис. 8.8 представляет диаграмму классов для концептуальных классов, необходимых в итерации 1. Диаграмма устанавливает ассоциации между классами. Ассоциации имеют имена и для них определены множественности. Классы содержат имена атрибутов и типы (строка, число, дата и т. д.). Операции же не определены.

Диаграмма классов содержит три класса: `Employee` (служащий), `Contact` (деловой партнер) и `OutMessage` (исходящее сообщение). Ассоциации названы `sender` (отправитель), `creator` (разработчик) и `for` (для). Задание имен ассоциациям делает ссылку на них более легкой, особенно в случаях, когда существует несколько ассоциаций между одними и теми же двумя классами (как имеет место для ассоциаций между `Employee` и `OutMessage`).

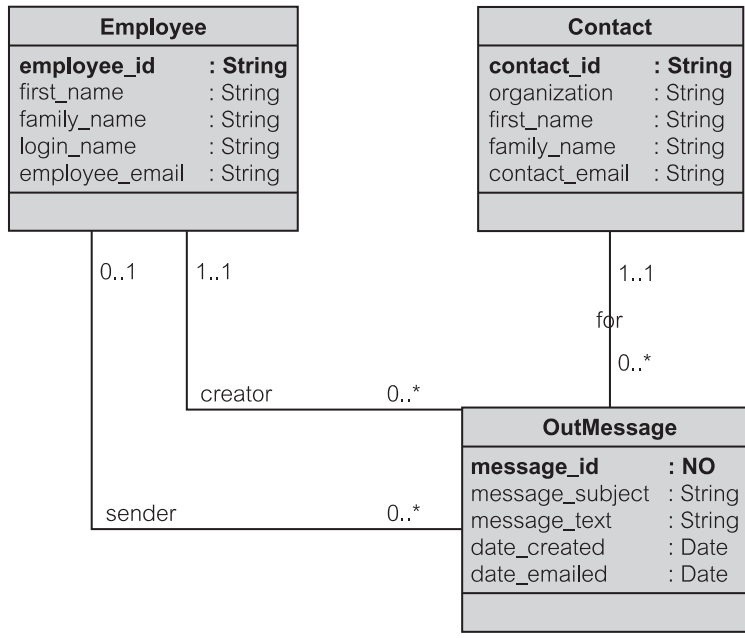


Рис. 8.8. Концептуальные классы в итерации 1

Имеется точно один Contact для каждого OutMessage. Contact может быть связан с нулем или несколькими OutMessage. Ассоциация по имени creator связывает OutMessage с Employee, кто создавал это OutMessage в БД. Ассоциация по имени sender идентифицирует Employee, который будет отправлять по электронной почте OutMessage. Эта ассоциация связана с нулем Employee, если отправитель для OutMessage еще не намечен.

Классы включают все атрибуты кроме тех, которые представляют ассоциации. Это совместимое с UML моделирование анализа. В UML-моделях проекта/реализации ассоциации станут атрибутами. **Атрибут ассоциации** имеет тип класса, с которым он связан.

В модели анализа на рис. 8.8 большинство атрибутов имеют тип string (строка). Тип атрибута message_id (идентификатор сообщения) — number (число) обозначен как NO. Два атрибута-даты имеют тип date (дата).

8.4. Дополнительная спецификация

Дополнительная спецификация — документ, который определяет требования и ограничения, не охваченные моделями и документами сценариев использования и концептуальных классов. Документ охватывает в основном нефункциональные требования и ограничения, то есть требования и ограничения неповеденческого характера — скорее ограничения и условия реализации.

Таблица 8.2 перечисляет требования и ограничения, представляющие типичную дополнительную спецификацию. Структура документа дополнительной спецификации может соответствовать этому списку.

Первые пять разделов документа дополнительной спецификации обычно известны как **FURPS**-особенности — аббревиатура, созданная из первых букв названий разделов на английском языке. С дополнением «другие ограничения», проблемы, отраженные в документе, упоминаются как особенности системы **FURPS+**.

Документ дополнительной спецификации должен быть начат на ранних стадиях проекта, чтобы идентифицировать ключевые вопросы, которые могут затронуть бюджет проекта и его выполнимость. Однако полное определение большинства дополнительных особенностей, как ожидается, будет определено в проекте позже. Подобно многим другим проектным продуктам, дополнительная спецификация растет вместе с проектом.

Соответственно, дополнительная спецификация для итерации 1 учебного примера весьма существенна. Главные проблемы здесь следующие:

- Функциональные возможности
  - EM — многопользовательское приложение.
  - Авторизация пользователя и его права доступа к различным средствам приложения управляются централизованно из БД, которая соединена с прикладной программой.

Таблица 8.2. Дополнительная спецификация

Краткое описание	Расширенное описание
Функциональные возможности (Functionality)	Этот раздел документа — единственный, который обращается к функциональным требованиям (поведению). Он обычно представляет функциональные требования, которые охватывают несколько сценариев использования и поэтому не учтены в документах конкретных сценариев использования. Например, здесь могут быть рассмотрены средства безопасности системы
Применимость (Usability)	Требования и ограничения, которые определяют то, что чувствуют акторы при применении системы. Применимость включает проблемы типа простоты использования системы, ее документации и возможностей справочной системы, обучения, требуемого для того, чтобы эффективно использовать систему, эстетики и логичности пользовательского графического интерфейса и подобных соображений
Надежность (Reliability)	Надежность определяет требуемую готовность системы по 24-часовой временной шкале, приемлемое среднее время между отказами, ожидаемую восстанавливаемость отказов разной сложности, точность продукции системы, и подобные проблемы
Выполнение функций (Performance)	Этот раздел включает ожидание времени отклика системы (в среднем и в пиковые времена), производительность операций, потребление ресурсов, возможное число параллельных пользователей и т. д. Требования выполнения функций могут изменяться для разных сценариев использования, так что этот раздел может относиться к определенным сценариям использования
Возможность сопровождения (Supportability)	Этот раздел лучше было бы назвать **удобством сопровождения и масштабируемостью**. Он определяет особенности системы, которые увеличивают удобство ее сопровождения и масштабируемость, то есть, непринужденность, с которой система может быть изменена, расширена, проверена, настроена на выполнение функций, переконфигурирована, перемещена на другие платформы, локализована и т. д.
Другие ограничения	Этот раздел включает различные важные характеристики проекта, которые не охвачены предыдущими пятью разделами. Он фиксирует стратегические решения относительно инфраструктуры проекта, юридические проблемы, которые могут затрагивать проект, соображения относительно интеграции с унаследованными системами, существующими в организации и т. д.

- Применимость
  - Не требуется никакого обучения для пользователя, знакомого с компьютером, чтобы он мог использовать итерацию 1 программы. Простого объяснения цели и основных особенностей приложения будет достаточно, чтобы использовать программу.
  - Итерация 1 — консольное приложение, которое ведет пользователя, обеспечивая однородный список опций меню. Единовременно может потребоваться только одна опция. Опции не взаимозависимы в том смысле, что все они доступны в любое время.
  - Будучи консольным, приложение может управляться с любого настольного пользовательского интерфейса.

- Надежность
  - Приложение должно быть доступно 24 часа каждый день недели. Не должно быть никакого связанного с БД времени простоя. О любых запланированных простоях почтового сервера из-за профилактик ЕМ-пользователи электронной почты должны быть уведомлены, по крайней мере, за 24 часа.
  - Отказ программы не должен ставить под угрозу корректность и целостность БД. Пользователь должен иметь возможность повторно начать программу после отказа и найти информацию БД непротиворечивой и не подверженной отказу программы.
- Выполнение функций
  - Нет никакого ограничения сверху на число одновременно работающих пользователей.
  - Время отклика системы не должно меняться, если число одновременно работающих пользователей — 100 или менее.
  - Время отклика для подпотоков S1 и S2 (раздел 8.2.3) должно быть меньше 5 секунд с 90-процентной вероятностью.
  - Время отклика для подпотока S3 (раздел 8.2.3) должно быть меньше 10 секунд с 90-процентной вероятностью для сообщений электронной почты, не превышающих 1 мегабайт в размере (включая любые приложенные документы). Однако обратите внимание, что итерация 1 не допускает использовать приложения в сообщениях электронной почты.
- Возможность сопровождения
  - Структурное проектирование системы должно соответствовать PCMEF-структуре, чтобы обеспечить надлежащее удобство сопровождения и масштабируемость (глава 9).
  - Для создания кода используется управляемая тестированием разработка. Для проверки кода используются приемочные испытания. Тестируемые единицы, полученные в результате управляемой тестированием разработки и приемочных испытаний, используются для **регрессионного тестирования**, когда код будет заменен на итерацию 2. Цель регрессионного тестирования состоит в том, чтобы гарантировать, что уже проверенный и принятый код не был нарушен в результате последующей разработки и программирования.
- Другие ограничения
  - Проект должен использовать БД Oracle, но он должен быть легко перестраиваемым для других реляционных БД.
  - Итерация 1 должна использовать Java и JDBC для доступа из программы к БД Oracle.

Резюме

1. Учебный пример, исследуемый в учебнике, называется ЕМ (Email Management — управление электронной почтой). ЕМ является функцией с СМ-предметной областью (Contact Management — управление деловыми

партнерами), рассмотренной в главе 7. По своим функциональным возможностям EM подобен коммерческим пакетам ПО, известным как Email Marketing (маркетинг с помощью электронной почты).

2. Сценарии использования EM, применяемые в итерации 1: `View Unsent Messages` (просмотр непосланных сообщений), `Display Message Text` (отображение текста сообщения) и `Email Message` (сообщение электронной почты).
3. Разделы документа сценария использования следующие: краткое описание, акторы, предусловия, постусловия, основной поток, подпотоки и потоки исключений.
4. Подобно классу предметной области, упомянутому в главе 7, *концептуальный класс* — определение для множества бизнес-объектов. В реализованном приложении концептуальные классы составляют одну группу (пакет) *прикладных классов*.
5. Документ *дополнительных спецификаций* определяет *FURPS+ особенности* системы. FURPS состоит из функциональных возможностей (functionality), применимости (usability), надежности (reliability), выполнения функций (performance) и возможности сопровождения (supportability). Плюс (+) означает добавление других, нежели определенных FURPS, особенностей.

Ключевые термины

Обзорные вопросы

Вопросы для обсуждения

1. Что такое модель сценариев использования, диаграмма сценариев использования и документ сценариев использования?
2. Как концепция альтернативного потока, часто используемая в литературе, относится к различным понятиям потоков, представленным в этой главе?
3. Какие особенности FURPS+ определяют удобство сопровождения системы и ее масштабируемость?

4. Какие особенности FURPS+ определяют юридические моменты для системы, типа закона неприкосновенности личной жизни?

Вопросы учебного примера

1. Какие могли бы быть удачные альтернативные названия для трех сценариев использования ЕМ на рис. 8.2?
2. Обратимся к рис. 8.7 и к определению потоков исключений в разделе 8.2.4. Имеется ли возможность использования исключения в сценарии на рис. 8.7, которое не рассмотрено в потоках определенных исключений?
3. Объяснение рис. 8.8 содержит следующее высказывание: «Ассоциация по имени `sender` (отправитель) идентифицирует `Employee` (служащий), который будет отправлять по электронной почте `OutMessage` (исходящее сообщение). Эта ассоциация связана с нулем `Employee`, если отправитель для `OutMessage` еще не намечен». Является ли это высказывание семантически верным? Если вы видите некоторую семантическую неточность, то как бы вы исправили это?
4. Позволяет ли итерация 1 модели хранить в БД ЕМ сведения о служащих, которые имеют в БД только групповую учетную запись? Объясните.

Примеры задач

Упражнения учебного примера

1. Модель сценария использования для учебного примера (рис. 8.2) проясняет, что ЕМ-приложение создает БД, которая используется многими АЕМ-приложениями (и ЕМ — только одно из этих приложений). Однако концептуальная модель для ЕМ (рис. 8.8) была создана без рассмотрения какой-либо существующей структуры БД (схемы). Это — приемлемая практика, потому что концептуальная модель — представление приложения относительно схемы БД, которая будет, в конечном счете, использоваться для соответствия между таблицами БД и классами предметной области приложения.

   Сказанное отразите на ситуации, в которой концептуальная модель должна быть приспособлена и расширена для включения указанных ниже требований (или, быть может, стоит рассмотреть существующую схему БД). Измените концептуальную диаграмму классов на рис. 8.8, чтобы рассмотреть следующие проблемы:

   а) Необходимо различать `PersonContact` (партнер-человек) и `OrganizationContact` (партнер-организация) как две категории `Contact` (деловой партнер). Хотя `OutMessage` (исходящее сообщение) всегда предназначается `for` (для) `Contact` (деловой партнер), деловым партнером может быть основной адрес электронной почты организации. В некоторых случаях это может быть единственный адрес электронной почты, который содержит ЕМ-подсистема для каждого делового партнера в этой организации. Возможно также, что `PersonContact` может быть индивидуумом, чья организация неизвестна.

б) Необходимо различать служащего, который запланирован для отправки исходящего сообщения, и фактического отправителя.
в) В некоторых случаях исходящее сообщение запланировано для отправки отделу, а не какому-то конкретному служащему. Любой служащий, работающий в этом отделе, может послать такое исходящее сообщение.
г) В некоторых случаях исходящее сообщение может быть послано только уполномоченными служащими согласно старшинству в штатном расписании.

Небольшой проект — временной протокол

Обратитесь к небольшому проекту временного протокола в главе 7. Напишите часть документа сценария использования (раздел 8.2) для временного протокола. Напишите только раздел документа, связанный с основным потоком. Включите эскизы или опытные образцы графических пользовательских интерфейсов (то есть экраны взаимодействия пользователя и компьютера).

Глава

9 Структурный проект

Для структурного проекта системы напрашивается аналогия со строительной индустрией. Дом не может быть построен, если архитектор не спроектировал его (возможно, навес — да, но не дом). Точно так же любая разумно большая система ПО не может быть построена без предшествующего структурного проектирования. Структура ПО — это основа, на которой должны базироваться все остальные проектные и программные решения.

Структуру системы следует рассматривать на ранних этапах процесса. Буч и др. [12] рассматривают структуру как единство трех основных характеристик любого процесса разработки, использующего UML. Эти три характеристики процесса следующие:

1. итерационный и пошаговый характер;
2. управляемый сценариями использования;
3. сконцентрированный вокруг структуры.

Что такое структура ПО? Определений много и они многогранны. Сжатое определение может быть таким: **структура ПО** — это организация элементов ПО в систему, предназначенную для достижения некоторой цели. Более детальные исследования показывают, что структура ПО связана с большим числом понятий типа организации определенных модулей ПО (классы, пакеты, компоненты), взаимосвязи между модулями, определения поведения модулей, масштабируемости модулей к большему количеству решений и т. д.

Что такое структурный проект? Снова возможны различные определения, но суть ясна. **Структурный проект** — множество решений, целью которых является работоспособная и эффективная структура ПО вместе с обоснованием этих решений. Обоснование, подчеркнутое в этой книге, — *понятность, удобство сопровождения и масштабируемость* системы. Ларман [55] выделяет четыре особенности структурного проектирования, а именно:

1. оно касается нефункциональных требований (см. дополнительную спецификацию в разделе 8.4);
2. включает крупномасштабные, фундаментальные решения системного уровня;
3. занимается взаимозависимостями и компромиссами и
4. обеспечивает формирование и оценку альтернативных решений.

9.1. Структурные уровни и управление зависимостями

Как и все производство ПО, структурное проектирование — непрерывная, итерационная и пошаговая работа. Первоначально структурные решения принимаются на основе широкого взгляда на структуру ПО. Одно из первых принятых решений касается структурирования системы на уровни модулей и установления принципов связи между модулями. Это тема данной главы. Более детальные структурные решения, типа связи внутри модуля, рассматриваются позже в соответствующих местах книги. Качественный структурный проект требует:

- иерархического **выделения уровней** модулей ПО, которое уменьшает сложность и делает более понятной зависимость модуля, запрещая непосредственную связь объектов, находящихся не на соседних уровнях, и
- использования стандартов программирования, которые делают зависимости модуля *видимыми* в компилируемых структурах программ и запрещают запутанные программные решения, использующие только структуры исполняемых программ.

Эти утверждения, особенно и в большой степени, истинны для современного объектно-ориентированного создания ПО. Понятие объекта вооружает инженера ПО множеством очень мощных абстракций программирования. Но если они используются неблагоразумно, получаются программы, которые невозможно ни понять, ни обслуживать даже теми программистами, которые написали их.

9.1.1. Структурные модули

Структурное проектирование — нечто вроде упражнения в управлении зависимостями модулей. Модуль А зависит от модуля В, если изменения в модуле В могут потребовать изменений в модуле А. Важно, чтобы эти зависимости не противоречили **брандмауэрам зависимостей** [66][1]. В частности, зависимости не должны быть между несоседними уровнями и не должны создавать циклы.

Структурное проектирование использует **упреждающий подход** к управлению зависимостями в ПО. Это делается так, чтобы решить на ранних стадиях процесса вопрос об иерархических уровнях модулей ПО и о брандмауэрах зависимостей между модулями. Такой подход является подходом *прямого проектирования* — от проекта к реализации. Цель состоит в том, чтобы разработать проект ПО, который минимизирует зависимости, возлагая структурное решение на программистов.

В конечном счете, упреждающий подход к управлению зависимостями должен быть дополнен **обратным подходом**, который подсчитывает зависимости в реализованном ПО. Это — подход *обратного проектирования* — от реализации к проектированию. Реализация может или не может соответство-

[1] Брандмауэр — система защиты доступа; брандмауэр зависимости — система, разрывающая зависимость между объектами. (*Прим. перев.*)

вать желаемому структурному проекту. Если это не так, цель состоит в том, чтобы сравнить количественные характеристики ПО с величинами, которые бы дала желаемая структура. Нежелательные зависимости должны быть точно определены и найдены.

Классы проекта

Предыдущие главы части 2 были сконцентрированы на моделировании «бизнес-объектов», классифицируемых как бизнес-сущности (глава 6), классы предметной области (глава 7) и концептуальные классы (глава 8). Однако «бизнес-объекты» — всего лишь множество классов в объектно-ориентированной программе.

В типичной программе нужны классы, ответственные за представление информации на экране компьютера пользователя, классы для доступа к БД, классы для выполнения алгоритмических вычислений и т. д. Используются различные имена, чтобы назвать все множество классов, которые должны быть разработаны и реализованы в компьютерной программе. Они попеременно называются классами проекта, классами ПО, прикладными классами, классами программы, классами системы или классами реализации. Здесь будет использован термин **классы проекта** или просто *классы*, но другие термины могут быть более подходящими в других местах книги (классы проекта — термин, также одобренный и UP — см. главу 6).

Пакеты

Классы проекта группируются в пакеты (packages) согласно структурному шаблону, принятому для разрабатываемого проекта. **Пакет** [106] — группировка элементов моделирования под назначенным именем. Пакет может содержать другие пакеты. Пакеты могут быть сгруппированы и структурированы в иерархические **уровни**, подходящие для выбранной структуры ПО.

В UML пакет — логическая концепция проекта. В конечном счете, пакеты должны быть реализованы и отображены в концепциях языка программирования. Современные языки, особенно Java и C#, обеспечивают прямое использование понятия пакета в конечной реализации. Поддержка пакета реализаций обеспечивается в форме пространства имен для классов и для импортирования других пакетов.

Пакет владеет своими *членами* (элементами) — удаление пакета из модели удаляет также и его члены. Из этого следует, что член (обычно класс) может принадлежать только одному пакету.

Пакет может импортировать другие пакеты. Это означает, что пакет `A` или элемент пакета `A` может обратиться к пакету `B` или к его элементам. Следовательно, класс принадлежит только одному пакету, но он может быть импортирован в другие пакеты. Импорт представляет зависимость между пакетами и их элементами.

Рис. 9.1 представляет пример UML-нотации пакета. Пакет может быть представлен без относящихся к нему членов/элементов (`Package A`). Пакет может зависеть от другого пакета (`Package A` зависит от `Package B`).

Отношение зависимости означает, что некоторые члены `Package A` обращаются некоторым способом к некоторым членам `Package B` (это может

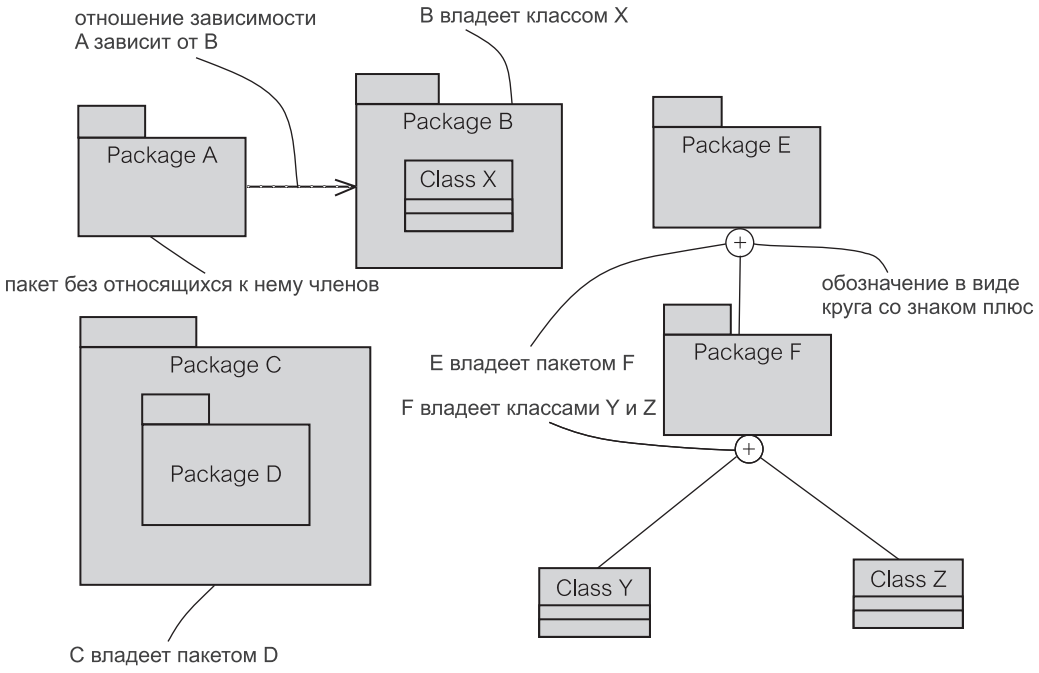


Рис. 9.1. Нотация пакетов

означать, что `Package A` импортирует некоторые элементы `Package B`). Включения двояки:

- Изменения в `Package B` могут воздействовать на `Package A`, обычно приводя к потребности перекомпилировать и перепроверить `Package A`.
- `Package A` может использоваться только (повторно использоваться) вместе с `Package B`.

Элементы пакета могут быть показаны включением их графических представлений в границы пакета-владельца или используя обозначение в виде круга со знаком плюс. На рис. 9.1 `Package B` владеет классом `Class X`, `Package C` владеет пакетом `Package D`, `Package E` владеет пакетом `Package F` и `Package F` владеет классами `Class Y` и `Class Z`.

9.1.2. Зависимости пакетов

Объекты в системе должны взаимодействовать друг с другом для решения ее задач. Для выполнения своих функций объекты должны зависеть друг от друга. Управление зависимостями не означает, что они по существу представляют проблему. Это лишь означает, что зависимости следует минимизировать, и ненужные должны быть устранены из структурного проекта.

Особенно неприятны **циклические зависимости** между объектами. К счастью, большинство циклических во времени пакетов можно избежать или сделать их относительно безопасными с помощью осторожного рефакторин-

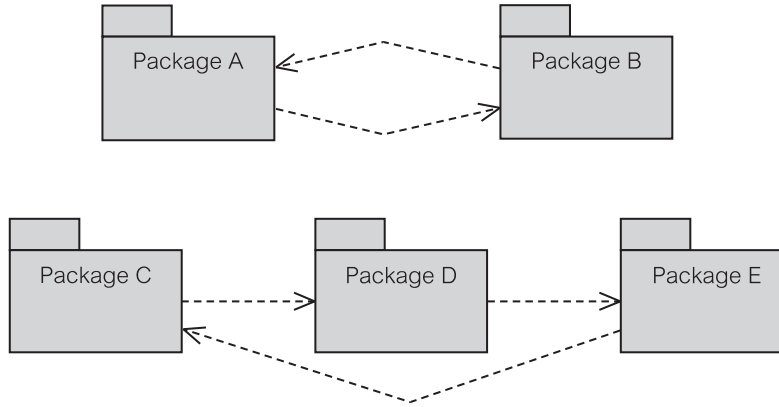


Рис. 9.2. Циклические зависимости между пакетами

га (перепроектирования — см. часть 3) или с помощью технологий программирования [66]. Рис. 9.2 показывает два примера циклических зависимостей между пакетами.

На рис. 9.2 `Package A` зависит от `Package B` и наоборот. Это означает, что связанное с зависимостью изменение в `Package B` потребует изменения и в `Package A`, которое может в свою очередь потребовать изменения в `Package B` и т. д. Окончательный вывод заключается в том, что такие два пакета являются неотделимыми с точки зрения понимания программы, удобства сопровождения и масштабируемости.

Точно так же изменение в `Package E` может потребовать изменения в `Package D`, а затем в `Package C`. Изменение в `Package C` могло в свою очередь потребовать изменение в `Package E` и т. д. Дорожка более длинная, но проблема та же самая.

Добавление нового пакета, как показано на рис. 9.3, может устранить циклические зависимости между пакетами (рис. 9.2). На рис. 9.3 элементы в `Package A`, от которых зависел `Package B`, были выделены в свой собственный `Package A2`. `Package B` теперь больше не зависит от `Package A`, он зависит от `Package A2`. Достаточно ясно, что и `Package A` зависит теперь от `Package A2`. Подобным образом во втором примере элементы в `Package C`, от которых зависел `Package E`, были выделены в свой собственный `Package C2`.

Горизонтальные структуры пакетов типа изображенных на рис. 9.3 называются **сегментами**. Когда циклические зависимости между пакетами устраняются путем добавления новых пакетов, как объяснено в данном разделе, структура зависимостей сегмента становится иерархической (а не линейной горизонтальной структурой).

9.1.3. Зависимости между уровнями

Как было сказано выше, пакеты могут быть сгруппированы и структурированы в иерархические уровни, подходящие для выбранной структуры ПО. Так как пакет может содержать другие пакеты, уровень сам является пакетом. В UML пакет уровня может быть обозначен как **«layer»** (уровень).

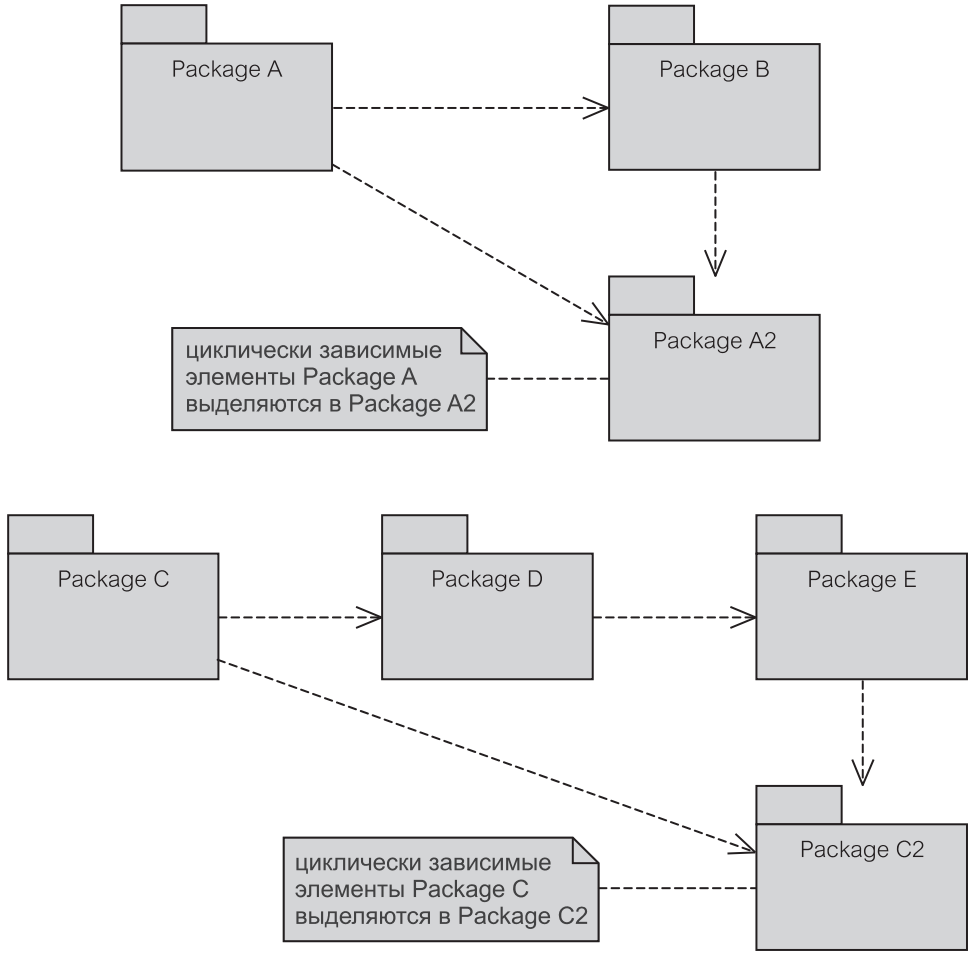


Рис. 9.3. Исключение циклических зависимостей между пакетами

С точки зрения структурного проектирования уровни представляют вертикальные структуры (рис. 9.4). Вертикальные уровни состоят из сегментов пакетов (раздел 9.1.2). Наложение вертикальных структур уровней на горизонтальные структуры сегментов создает иерархию зависимостей пакетов. Три критических показателя хорошего структурного проектирования уровней следующие:

- иерархия уровней не представляет сетевую структуру (где пакет может потенциально зависеть от любого другого пакета в системе);
- иерархия уровней минимизирует зависимости между пакетами;
- иерархия уровней устанавливает *стабильный* шаблон для жизненного цикла разработки системы.

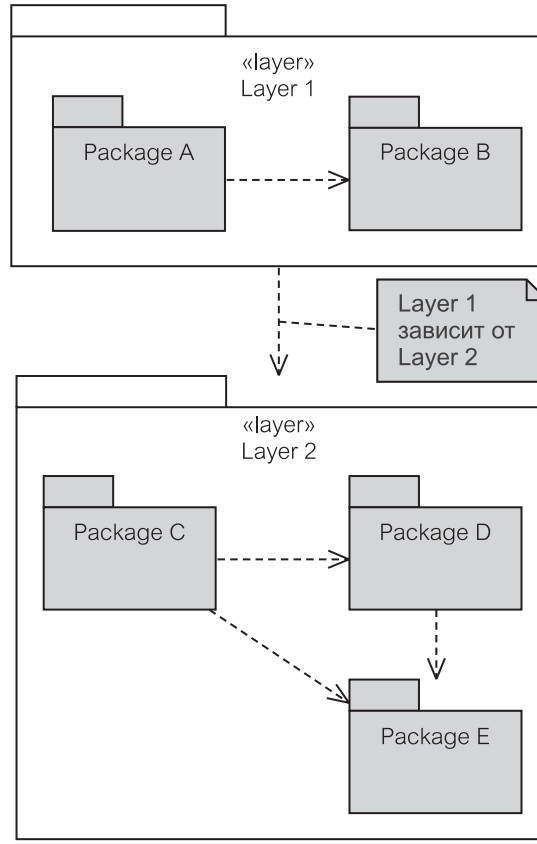


Рис. 9.4. Уровни в качестве пакетов

Первый показатель интуитивно очевиден. Сложность сетей растет по экспоненте с дополнением новых элементов к структуре. На практике все сложные работающие структуры, включая живые организмы и сделанные человеком системы, являются *иерархическими*.

Второй показатель утверждает, что иерархия уровней должна *минимизировать зависимости* между пакетами. Широко используемый метод достижения этого — делать более высокие уровни зависимыми от более низких уровней, но не наоборот. К сожалению, структуры зависимостей только сверху вниз не совсем реалистичны. В действительности будут существовать зависимости на нижнем уровне, но они могут быть сделаны относительно безопасными квалифицированным проектированием и программированием. Желательный результат таков, чтобы более высокие уровни зависели от более низких уровней, в то время как более низкие уровни все еще могли бы связываться с более высокими уровнями, но без создания неуместных (неуправляемых) зависимостей.

Третий показатель означает то, что иерархия уровней является **стабильным шаблоном**. **Стабильность** означает нечто стойкое к изменениям и со стабильными целями. Устойчивый структурный шаблон, единожды созданный, невосприимчив ни к каким изменениям. Стабильный шаблон определяет фиксированный набор «правил игры» при разработке ПО. В пределах этих правил отдельные шаги «игры» гибки.

Обратите внимание, что стабильность вертикальных уровней увеличивается в нисходящем направлении. Более высокие уровни зависят от более низких уровней. Требуется, чтобы более низкие уровни были устойчивыми, потому что любые изменения в них могут привести к «эффекту ряби» на более высоких уровнях [66].

`Layer 2` на рис. 9.4 **стабилен**, а `Layer 1` **нестабилен**. `Layer 1` зависит от `Layer 2`. `Layer 2` независим и поэтому может быть заменен новым без «эффекта ряби» в остальной части системы. Это — принцип и причина, стоящие за разрешением сильной зависимости (**сильной связи**) в нисходящем направлении и обеспечением слабой зависимости (слабой связи) в восходящем направлении.

Обратите внимание, что условие стабильности уровней означает неприемлемость технологии устранения циклических зависимостей между уровнями добавлением новых уровней. К счастью, существует технология программирования, обеспечивающая устранение циклических зависимостей или делающая их относительно безопасными.

9.1.4. Зависимости классов

Технологии программирования, которые позволяют устранять или выводить из структуры циклические зависимости между уровнями (и пакетами в целом), связаны скорее со структурным проектированием классов, чем самих пакетов. Дело в том, что:

- зависимости между уровнями преобразуются в зависимости (вызываются зависимостями) между пакетами;
- зависимости между пакетами преобразуются в зависимости между классами;
- зависимости между классами преобразуются в зависимости между элементами классов, и в первую очередь, между методами.

На рис. 9.5 `Layer 1` зависит от `Layer 2`, потому что имеется некоторый класс в `Layer 1`, который зависит от класса в `Layer 2`. `Package A` зависит от `Package B`, потому что имеется некоторый класс в `Package A`, который зависит от класса в `Package B`. Последствие таково, что если неприятные зависимости классов (то есть, зависимости классов, которые представляют циклы) могут быть устранены или сделаны безопасными, то вся структура уровней и пакетов ПО может быть намного более устойчива.

Основная технология программирования, предназначенная для разрушения циклов между классами и по этой причине приводящая к устранению циклов между пакетами и уровнями, использует концепцию *интерфейса*. Поддерживающей технологией является целевое использование *обработки событий*, возможно, на основе использования интерфейсов. Две технологии объясняются в разделах 9.1.7 и 9.1.8, после того, как будут описаны зависимости между методами.

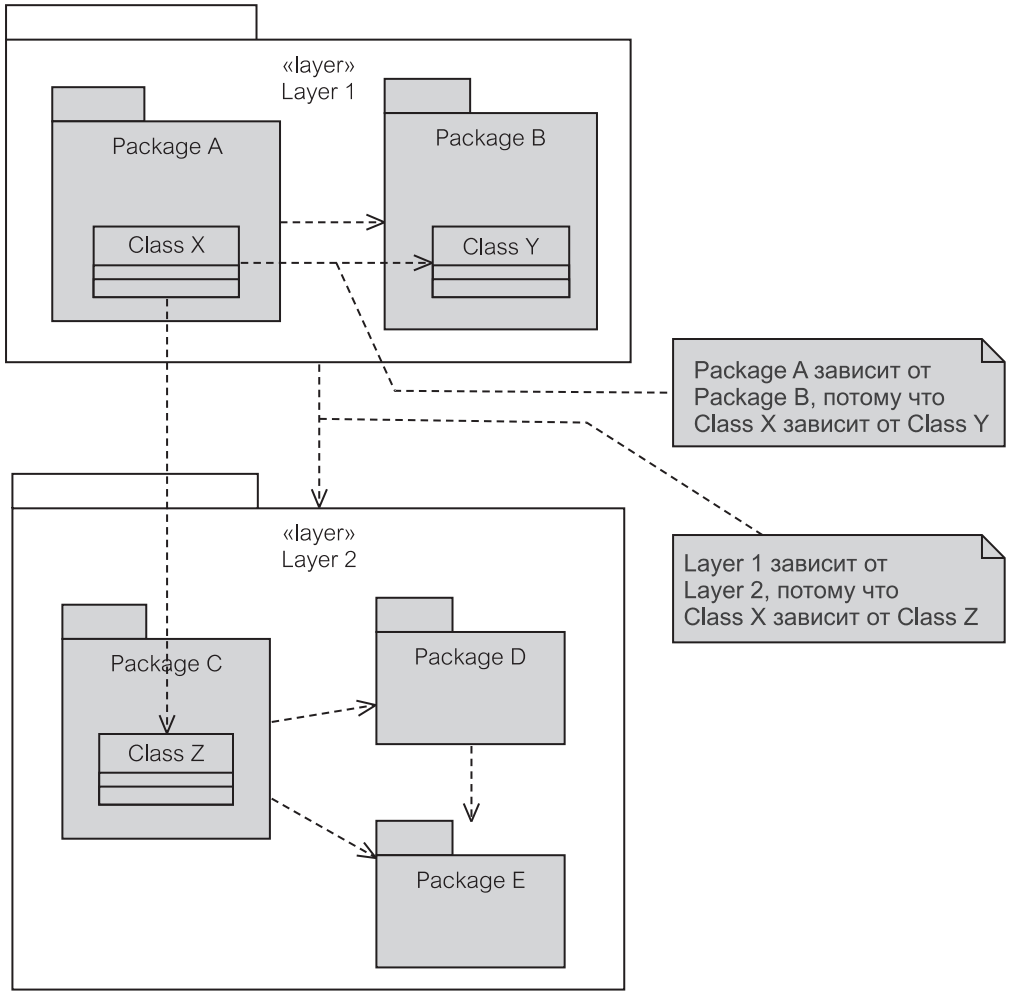


Рис. 9.5. Зависимости классов и вытекающие из этого зависимости уровней и пакетов

9.1.5. Наследование зависимостей

Одна из наиболее неприятных проблем в управлении зависимостями — зависимости классов и методов, вызванные наследованием реализации. **Наследование реализации** — средства структурного и поведенческого разделения функций между основным классом (или суперклассом) и его производными классами (подклассами), такие, что обращения к обслуживанию во время выполнения могут дать объект подкласса вместо объекта суперкласса. Объект подкласса — *своего рода* объект суперкласса, который унаследует особенности всего суперкласса, но он может изменить (переопределить) некоторые из этих особенностей, а также может добавить новые функциональные воз-

можности, определенные в подклассе. В результате этого, с точки зрения объекта-клиента поведение системы во время выполнения может быть различно в зависимости от конкретного объекта в дереве наследований, который обслуживает запрос.

Это понятие различного поведения в зависимости от объекта, который должен обслужить запрос, называется **полиморфизмом**. Вызов соответствующего метода, относящегося к инициализируемому классу во время выполнения, известен как **динамическое связывание** (или **позднее связывание**). Это действительно позднее связывание, поскольку обращение к методу (приводящее к переключению от объявления метода основного класса к объявлению метода подкласса) происходит во время выполнения, а не во время компиляции.

Наследование не всегда сопровождается полиморфизмом. Множество классов может быть организовано в иерархию от основного до конкретного класса, и, тем не менее, они могут и не переопределять какие-либо методы, объявленные их суперклассами. Поэтому они не обеспечивают полиморфное поведение. Полиморфное поведение обеспечивается **переопределением методов**. Оно означает, что подклассы обеспечивают более контекстно-определенную реализацию метода, объявленного в суперклассе, и поэтому изменяют его первоначальное поведение.

Переопределение метода отличается от перегрузки метода. Переопределение метода требует, чтобы подкласс обеспечил точно ту же самую сигнатуру (см. раздел 2.2.1), что и у метода, который подлежит переопределению. **Перегрузка метода** — это когда класс должен обеспечить множество методов с тем же самым именем, но разными сигнатурами. Java API дает много примеров перегрузки методов. Возьмем, например, метод `println()` (распечатать строку) объекта `System.out`. Метод `println()` — перегружаемый метод. Это фактически множество методов с одним и тем же именем и различными параметрами. Имеется версия `println()`, которая использует целое число, версия, которая использует объект `String` (строка), версия, которая использует символ, и т. д.

Ясно, что наследование реализации, полезное для **повторного использования кода**, дает сомнительные зависимости между классами и методами. Зависимости наследования можно расклассифицировать на две группы:

- зависимости времени компиляции между классами в дереве наследования;
- зависимости времени выполнения, включающие объекты-клиенты, обращающиеся к сервисам классов в дереве наследования.

Рис. 9.6 — простой пример, демонстрирующий *зависимости времени компиляции* между классами в дереве наследования и открывающий возможность для зависимостей времени выполнения. При интерпретации этого примера только в контексте информации, явно показанной на рисунке, объект `A` наследует объект `Object`, но это не обеспечивает его собственную переопределенную версию сервиса `wait()` (ожидать). Поэтому можно было бы ожидать, что между `A` и `Object` нет никакой зависимости наследования времени выполнения. Однако имеется зависимость времени компиляции, когда любые изменения в методе `wait()` объекта `Object` будут статически унаследованы объектом `A`.

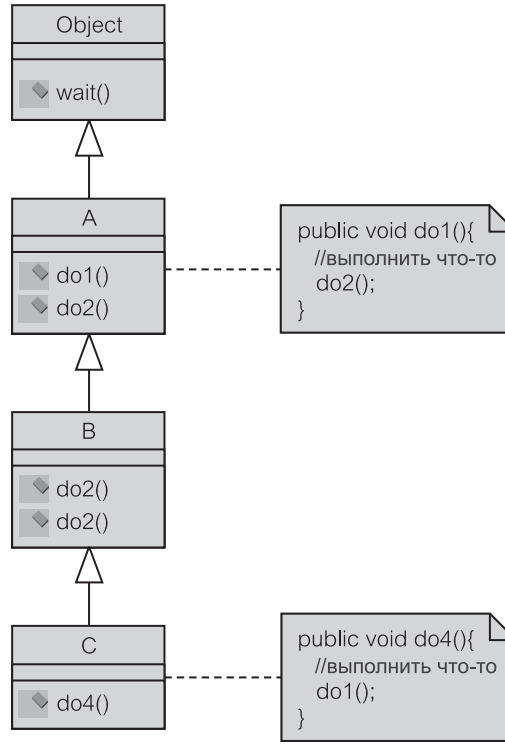


Рис. 9.6. Зависимости наследования времени компиляции

Вообще, все отношения обобщения в диаграмме представляют зависимости времени компиляции в направлении изображенных стрелок (`C` зависит от `B`, `B` зависит от `A`, `A` зависит от `Object`). Зависимости *транзитивны*, что означает: если `C` зависит от `B` и `B` зависит от `A`, то `C` зависит от `A` и т. д.

Однако имеются дальнейшие осложнения времени выполнения, скрытые в рис. 9.6. `B` наследует `A` и `B` переопределяет метод `do2()` объекта `A`. Это создает потенциально полиморфное поведение `do2()` и представляет вероятную зависимость времени выполнения в противоположном направлении от `A` к `B`. `A` будет зависеть от `B`, если метод `do1()` объекта `A` должен вызвать метод `do2()` объекта `B`, нежели свой собственный. Этими видами циклических зависимостей, полученных из-за наследования, очень трудно управлять, и они создают трудность сопровождения.

Рис. 9.7 иллюстрирует зависимости времени выполнения, включающие объект-клиент, обращающийся к сервисам классов в дереве наследования. Интересный (и специфический) аспект представленной модели заключается в том, что хотя объект `Test` (тестирование) инициализирован объектом `A` (чтобы представить его связь с объектом `Object`), но объект `Test` не использует сервисы объекта `A`. В результате `Test` не имеет зависимости наследования времени выполнения от `A`. Кроме того, зависимость `Test` от метода `wait()` — действительно статическая зависимость, обусловленная существованием связи от `Test` к `Object`.

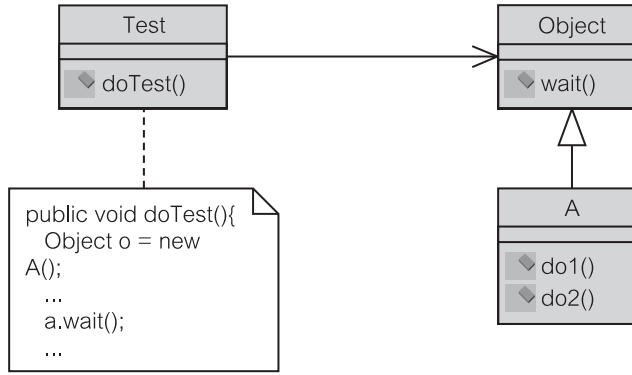


Рис. 9.7. Зависимости наследования времени выполнения

Обратите внимание, что зависимости времени выполнения могли бы перекрыть момент, когда `A` переопределяет метод `wait()`. Если `A` переопределил метод `wait()`, то зависимости времени выполнения существовали бы от `Test` к `Object` и `A`. Следующие подразделы иллюстрируют наиболее частые контексты, в которых происходят зависимости наследования времени выполнения.

Наследование без полиморфизма

Наследование реализации, несмотря на свою мощь повторного использования, дало статус нарушителя спокойствия многим «мастерским» по разработке ПО. Этот статус справедлив только до того момента, пока программисты чрезмерно используют и злоупотребляют наследованием реализации [63, 61]. Если же его использовать должным образом, наследование реализации остается мощной и очень полезной технологией.

Самый простой путь использования наследования — это когда подкласс не переопределяет унаследованные методы (не обеспечивает свои собственные реализации методов). В таком случае будет наследование без полиморфизма. Хотя **наследование без полиморфизма** не так уж и полезно, его легче всего понять и легче им управлять. В действительности же подкласс может переопределить некоторые, но не все унаследованные методы, таким образом ограничивая наследование без полиморфизма по отношению к отобранным методам.

Рис. 9.8 иллюстрирует, что класс `B` наследует `do2()`, но не переопределяет его. Поэтому `do3()` класса `B` использует метод `do2()`, полностью сформированный в родительском классе `A`.

Расширяющее и ограничивающее наследование

Расширяющее наследование — соответствующее использование наследования реализации. Суть его заключается в том, что подкласс наследует свойства своего суперкласса и обеспечивает дополнительные особенности для расширения своего определения. Подкласс в этом случае выступает своего

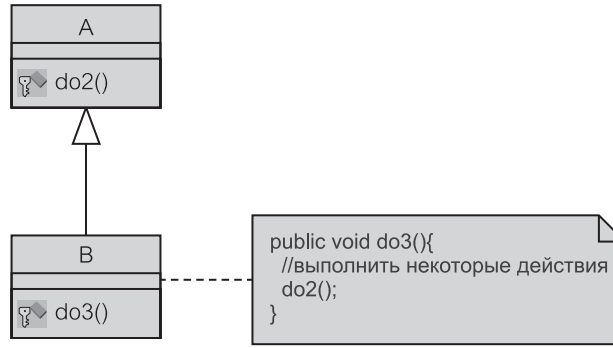


Рис. 9.8

рода разновидностью суперкласса. Переопределенные методы (если они переопределяются подклассом) должны быть выполнены так, чтобы обеспечить их определение и работу в соответствии с потребностями подкласса.

Рис. 9.9 — пример расширяющего наследования, где подкласс `B` наследует `do2()` и `do3()` от `A` и затем переопределяет `do3()`, возможно, расширяя функциональные возможности, полученные от унаследованного метода. Метод `do3()` может быть вызван как из объекта `A`, так и из объекта `B`. Он будет выполняться по-разному в зависимости от того, из какого объекта вызван.

Использование наследования реализации, отличного от расширяющего наследования, проблематично. Проблематичные наследования располагаются в диапазоне от различных форм ограничивающих наследований до просто недопустимого непосредственного наследования [61]. **Ограничивающее наследование** происходит, когда класс наследует метод и затем переопределяет его, лишь устраняя часть унаследованных функциональных возможностей. В результате этого, подкласс уже больше не является *разновидностью* суперкласса. В некоторых случаях ограничение кончается полным подавлением унаследованного метода. Это случается, когда метод реализован как пустой (то есть, не выполняющий никаких операций).

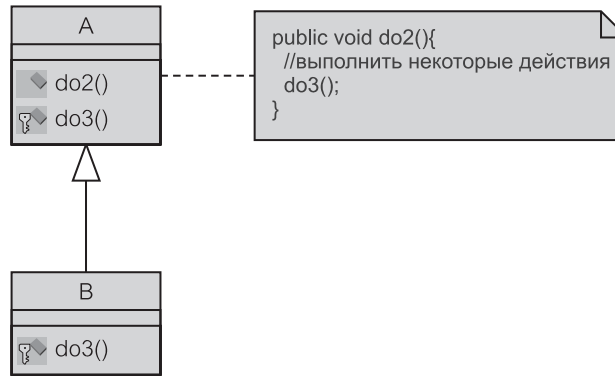


Рис. 9.9. Расширяющее наследование

Вызовы методов подкласса

Переопределение, а следовательно, и полиморфизм, дает возможность использовать **вызовы методов подкласса**. Это иллюстрируется на рис. 9.10. Даже притом, что класс `X` связан с классом `A`, тем не менее, когда `X` выполняет `do2()`, он фактически выполняет реализацию `do2()` класса `B`. Это происходит потому, что `do2()` выполняется на объекте `myA`, который является экземпляром класса `B`.

Вызов метода подкласса на рис. 9.10 представляет зависимость наследования времени выполнения от `X` к `B`, которая не видна в статической структуре программы времени компиляции. `X` имеет связь с `A`, но не с `B`. Такими зависимостями времени выполнения очень трудно управлять (именно потому, что они не видимы при статическом определении программы).

Вызовы методов суперкласса

Переопределение, а следовательно, и полиморфизм, допускает также и **вызовы методов суперкласса** (или **обратные вызовы**). Это иллюстрируется на рис. 9.11. На сей раз подкласс `B` явно использует реализацию предка `do2()`, хотя фактически и имеет свою собственную переопределенную версию `do2()`. Хотя такой вызов через `super` и можно объяснить, комбинация вызовов методов подкласса и суперкласса представляет неприятную проблему циклической зависимости вовлеченных в это классов.

Зависимости наследования можно рассматривать как форму зависимостей методов. Вообще, кроме обычных зависимостей методов существуют и другие виды зависимостей методов, в том числе делегирование и наследование.

9.1.6. Зависимости методов

Зависимости между классами переходят в зависимости между элементами класса (особенностями — на языке UML). Обычно зависимостями между элементами данных можно управлять относительно легко (хотя наследование реализации элементов данных может испортить картину). Зависимости меж-

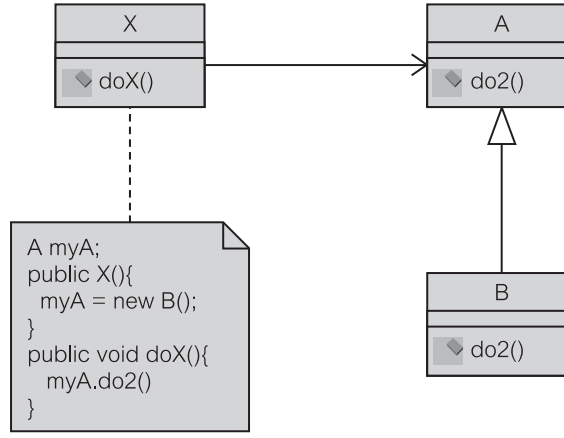


Рис. 9.10. Вызовы методов подкласса

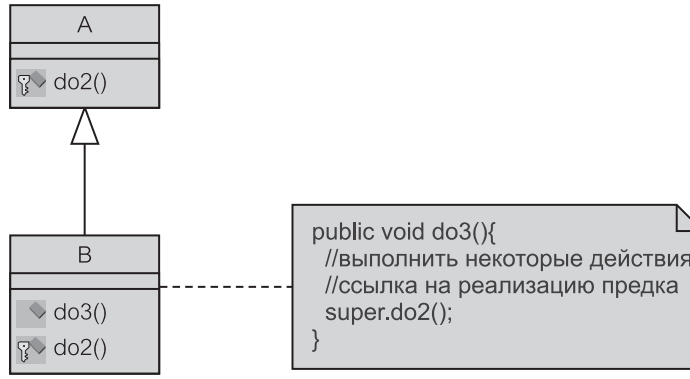


Рис. 9.11. Вызовы методов суперкласса

ду методами создают реальную проблему, в особенности, потому что на практике многие зависимости методов нельзя проследить, анализируя только статическую структуру программы времени компиляции.

Рис. 9.12 иллюстрирует зависимости методов и как они преобразуются в зависимости классов (и поэтому также и зависимости пакетов и уровней). Имеются два пакета, называемые `control` — управление и `entity` — сущность (названия не используют прописные буквы, следуя общепринятой практике). Имена классов начинаются с заглавной буквы, означающей конкретный пакет, к которому этот класс принадлежит. Следовательно, `CActioner` (класс «исполнитель» пакета `control`) обозначает, что класс принадлежит пакету `control`.

`CActioner` использует метод `do1()`, чтобы послать сообщение `do3()` классу `EEmployee` (класс «служащий» пакета `entity`). Поэтому `do1()` зависит от `do3()`. Зависимость распространяется на классы-владельцы и пакеты. `CActioner` зависит от `EEmployee`, и `control` зависит от `entity`. Точно так же `do2()` в `EOutMessage` (класс «исходящее сообщение» пакета `entity`) запускает метод `do3()` из `EEmployee`. Следовательно, `EOutMessage` зависит от `EEmployee`.

Обратите внимание, что зависимости метода сделаны явными в модели на уровне классов, устанавливая *однонаправленные ассоциации* между зависимыми классами. И `CActioner` (класс «исполнитель» пакета `control`), и `EOutMessage` имеют элементы данных `emp` (служащий) типа `EEmployee`, которые реализуют однонаправленные ассоциации к `EEmployee` и называются `ActEmp` (участник-служащий) и `OutEmp` (исходящее сообщение-служащий) соответственно.

Сделать зависимости методов статически видимыми в коде посредством явных ассоциаций между классами — настоятельно рекомендуемая практика [56]. Это важно, потому что зависимости методов часто не поддаются обнаружению при анализе исходного кода. В случае наследования или полиморфизма и по требованиям уровневой структуры формирователь сообщения (**объект-клиент**) часто не знает конкретного получателя сообщения (**объект-поставщик**) до времени выполнения.

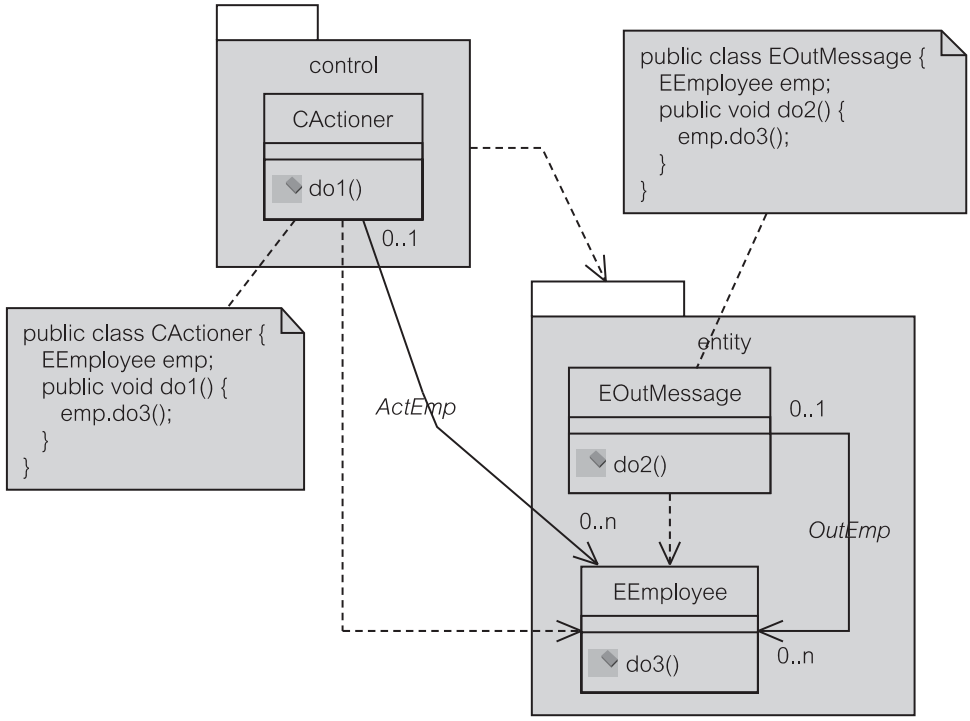


Рис. 9.12. Зависимости методов и вытекающие из этого зависимости классов и пакетов

Если анализ поведения программы обнаруживает зависимость методов между классами без явной ассоциации, можно считать, что программа нарушает структурное проектирование. Однако в некоторых случаях создание ассоциаций с тем, чтобы узаконить зависимость методов, само может быть нарушением структурного проектирования. Включение интерфейсов (раздел 9.1.7) и обработки событий (раздел 9.1.8) в проект может решить эту дилемму. Более сложные зависимости методов времени выполнения (называемые *зависимостями знакомства*) могут потребовать даже более радикальных решений проекта, как рассмотрено в разделе 9.1.9.

Зависимости методов при наличии делегирования

Передача сообщения реализуется как **синхронная связь** между клиентом и поставщиком сервиса. Сообщение от *объекта-клиента* просит, чтобы *объект-поставщик* исполнил сервис (метод). Интерпретация сообщения и средство его выполнения — на усмотрение объекта-поставщика. Это можно было бы назвать **делегированием** работы другому объекту.

Как в цепи военных распоряжений, объект может делегировать полномочия на выполнение работы другому объекту. То, что воспринимается объек-

том-клиентом как объектом-поставщиком, является на самом деле **делегирующим объектом**. Хотя работа делегирована, объект-поставщик (имеет псевдоним делегирующего объекта) не освобождается от договорной ответственности (к объекту-клиенту) выполнить работу.

Делегирование обычно необходимо, чтобы позволить объекту-клиенту получить услугу на одном уровне от объекта, находящегося на отдаленном (не соседнем) уровне. Иными словами, устойчивый шаблон структуры вертикальных иерархических уровней (раздел 9.1.3) будет разложен на случайную сеть взаимосвязанных объектов без какой-либо надежды понять или управлять сложностью системы и ее эволюцией.

Рис. 9.13 демонстрирует зависимости методов при наличии делегирования, когда шаблон уровней состоит из пакетов, названных `control` (управление), `entity` (сущность), `mediator` (посредник) и `foundation` (основание). Для ясности однонаправленные ассоциации, показывающие передачу сообщений, не изображены (но они запрограммированы в коде Java, представленном в UML-нотации).

На рис. 9.13 `CActioner` и `EOutMessage` запрашивают сервис `do3()` от `EEmployee`. Однако `EEmployee` делегирует выполнение сервиса классу `MBroker` (класс «посредник» пакета `mediator`), а `MBroker` делегирует его далее классу `FUpdater` (класс «корректировщик» пакета `foundation`). `FUpdater` исполняет сервис. Это сообщается назад объектам-клиентам по тому же пути, но в обратном направлении. Обратите внимание, что делегирование часто необходимо для согласования вертикальной структуры уровней, которая не допускает прямую связь между несоседними уровнями.

Последствие делегирования таково, что *клиент* может и не знать своего реального *поставщика* (и он даже может не хотеть знать это, пока не получит «требуемое»). В отличие от рис. 9.13 знание реального поставщика может быть недоступно из статического анализа программного кода и может быть скрыто за динамикой наследования (в частности, наследования интерфейса) и полиморфизма.

Поскольку выявление поведения программы во время выполнения — «не лучшая забава» для эксплуатационного персонала системы или менеджера проекта, практика *явных ассоциаций* между классами, занятыми в передаче сообщения более предпочтительна. Без явных ассоциаций **анализ воздействий** из-за изменений в поставляемом коде может быть невозможен.

Зависимости методов в присутствии наследования реализации

Как было видно в разделе 9.1.5, зависимости **наследования реализации** в значительной степени представляют специальный вид зависимостей сообщений (методов). Зависимости метода из-за наследования реализации трудны для управления. Многие из таких зависимостей возникают динамически во время выполнения и не видимы в структуре программы времени компиляции.

Однако имеются как «за», так и «против». Динамическое поведение из-за переопределения методов позволяет обеспечить сложные операции меньшими затратами. Подкласс должен только переопределить некоторые методы, чтобы обеспечить разнообразное поведение. Чтобы расширить функциональ-

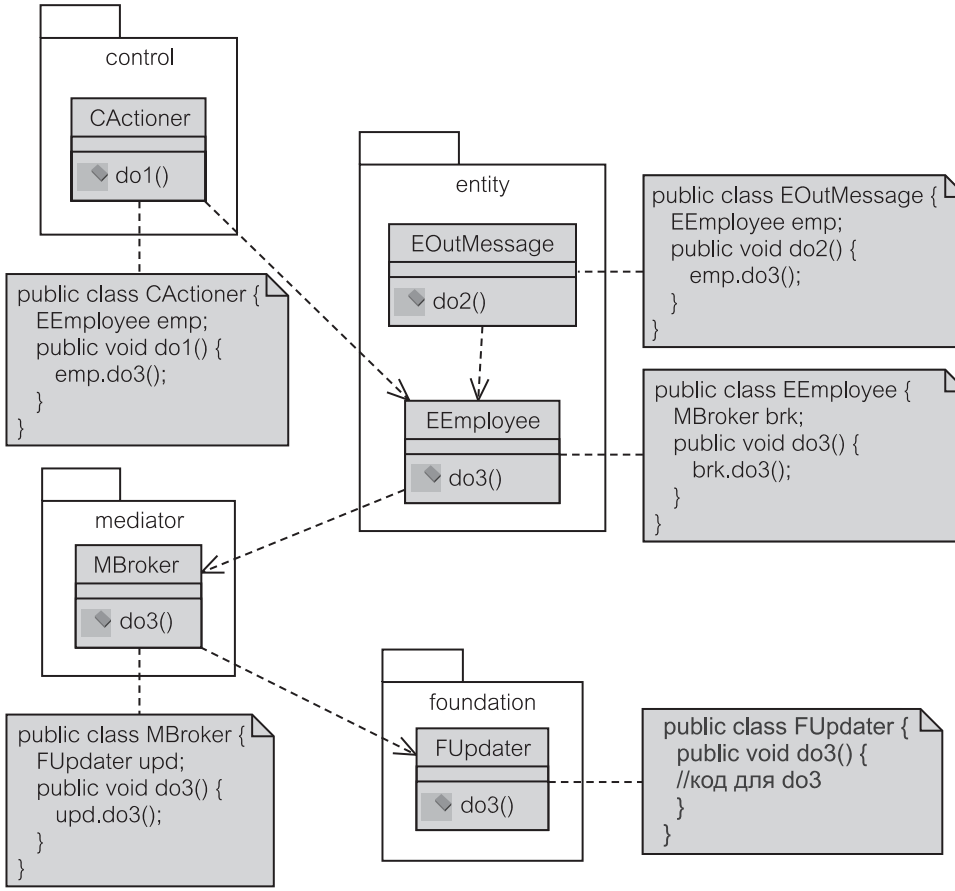


Рис. 9.13. Зависимости методов в присутствии делегирования

ные возможности, уже обеспеченные суперклассом, могут быть добавлены дополнительные методы.

Мало того, что многие зависимости методов из-за наследования реализации не поддаются обнаружению при анализе исходного кода, они также, по всей видимости, создадут циклы. Главная причина появления циклов — объединенное использование вызовов методов подкласса и суперкласса между объектом-клиентом и объектом-поставщиком. Рис. 9.14 иллюстрирует возможные осложнения в случае, когда передача сообщения происходит на объектах, участвующих в наследовании реализации.

Рис. 9.14 показывает, что `Client` — клиент имеет ссылку (`class1`) на `SubClass` — подкласс, но вместо этого хранит эту ссылку как тип `SuperClass` — суперкласс (последовательность 1). Практически назначение экземпляра `SubClass` ссылке на `SuperClass`, вероятнее всего, прои-

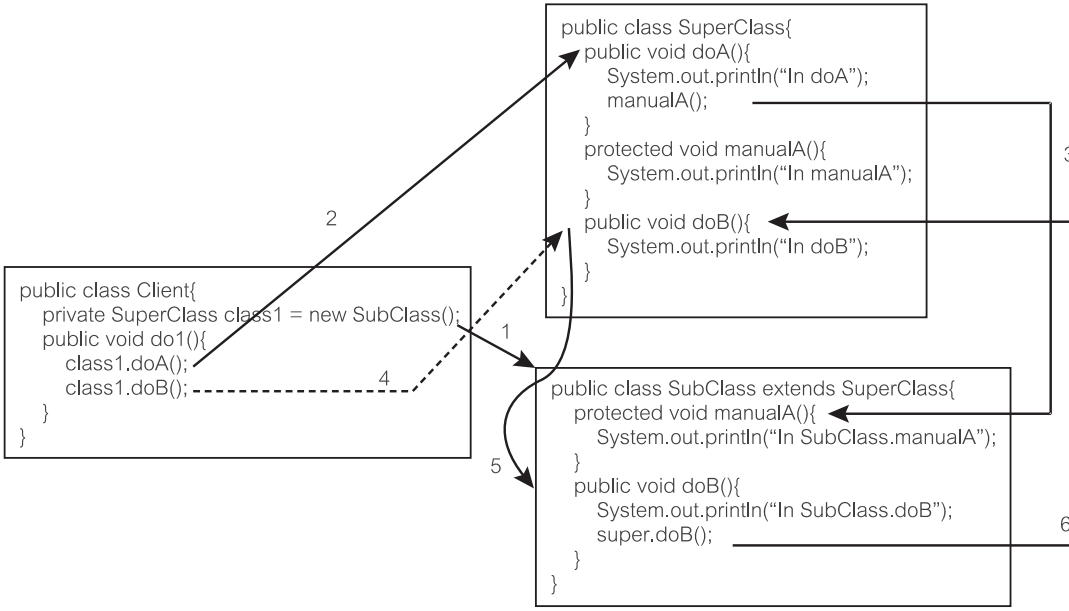


Рис. 9.14. Вызовы методов подкласса и суперкласса

зойдет во время выполнения, а не как простое назначение, показанное на рис. 9.14. Когда вызывается метод `do1()`, он выполняет `doA()` класса `SuperClass` (последовательность 2). Кажется, что `doA()` вызывает метод `manualA()`. Однако `manualA` класса `SuperClass` переопределен, и поэтому вместо него вызывается переопределенный метод (последовательность 3). Выполнение последовательностей 2 и 3 — пример вызова метода подкласса.

Затем `do1()` выполняет `doB()` класса `SuperClass`. И снова `doB()` переопределен классом `SubClass`, поэтому вместо метода класса `SuperClass` выполняется метод `doB()` класса `SubClass` (последовательность 5). Интересно, что метод `doB()` класса `SubClass` обеспечивает расширение метода `doB()` родителя. Он выполняет метод `doB()` своего родителя для завершения операции (последовательность 6). Последовательность 6 — пример вызова метода суперкласса.

Полиморфное поведение, включающее комбинацию вызовов методов подкласса и суперкласса, создает проблемы сопровождения. Вызовы методов суперкласса и подкласса связывают классы и их методы в беспорядочную сеть взаимосвязанных объектов, где даже различие между объектом-клиентом и объектом-поставщиком оказывается стертым. Ошибку в одном из методов этой последовательности выполнения трудно проследить, поскольку реальный экземпляр класса может быть определен только во время выполнения.

9.1.7. Интерфейсы

В UML 2.0 **интерфейс** — это объявление ряда *особенностей* (раздел 9.1.1), которые непосредственно не проявляются, то есть, никакие их объекты не могут быть непосредственно созданы. Интерфейс реализуется (осуществляется) с помощью объекта класса, который обеспечивает предоставление структуры и поведения интерфейса для остальной части программы. Объект, который реализует интерфейс, обеспечивает «*внешний вид*, который соответствует спецификации интерфейса» [106].

Концепция интерфейса UML 2.0 расширяет понятие интерфейса, используемого в популярных языках программирования (и в более ранних версиях UML). Интерфейс может объявлять атрибуты, а не только операции. В противоположность этому Java-интерфейс может содержать только элементы данных, которые должны быть константами (определенными как `static` — статический и `final` — заключительный).

Как последствие разрешения атрибутов в интерфейсах, можно создать ассоциации между интерфейсами и между интерфейсом и классом. Атрибуты, определенные как другой интерфейс или класс, представляют ассоциации. В UML 2.0 можно организовывать переход от интерфейса к классу через ассоциацию. В Java это невозможно.

Интерфейс иногда считают «чистым» абстрактным классом. Это не совсем так (хотя это — лучшее, что можно сделать на языках, которые не поддерживают интерфейсы, типа C++). **Абстрактный класс** — это класс, который содержит, по крайней мере, один метод, который не реализован (или не может быть реализован) этим классом, и поэтому его нельзя инициализировать. В **чисто абстрактном классе** никакой из методов не реализован.

Чистый или нет, класс всегда остается классом. На языках, которые поддерживают только **единственное наследование реализации**, типа Java, класс может расширять только один базовый класс (абстрактный или конкретный), но он может реализовать много интерфейсов. Это существенное практическое различие.

Данное различие заключается в том, что интерфейсы позволяют передавать в вызовах методов объекты, имеющие типы интерфейсов. Точный класс такого интерфейса не нужно определять до времени выполнения. Для объекта-клиента не имеет значения, каков класс поставщика. Существенно лишь то, что он реализует вызванный метод [56].

Интерфейсы в частности, но также и абстрактные классы, используемые в качестве основного входа в конкретные классы пакетов, обеспечивают механизмы, скрывающие внутреннюю сложность пакета и разрешающие расширение пакета без воздействия на объекты-клиенты в других пакетах. Понятие **доминирующего класса** может использоваться, чтобы реализовать эти механизмы в пределах пакета. Доминирующий класс реализует главные интерфейсы и абстрактные классы пакета. По словам Рамбо и др. [90] «доминирующий класс относится к категории интерфейса компонента».

Рис. 9.15 представляет UML-нотацию для интерфейса, абстрактного класса (изображается курсивом) и класса. Нотация интерфейса отличается отображением стереотипа «`interface`» (интерфейс). Интерфейс может быть

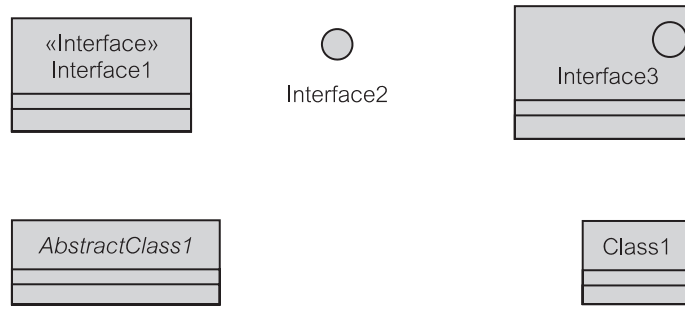


Рис. 9.15. UML-нотация для интерфейса, абстрактного класса и класса

показан обычным образом слева направо как явная метка, пиктограмма или «оформление».

UML 2.0 представил другую изобразительную разновидность, где круг (называемый «мячом») может быть дополнен полуокружностью (называемой «гнездом»). Однако нотация в виде «мяча» и «гнезда», кажется, не повышает изящности в больших моделях. Поэтому в этой книге мы будем избегать таких изображений в пользу нотации в виде художественного оформления (на самом деле не рекомендованного непосредственно в UML 2.0, но широко используемого на практике).

Зависимость реализации

Интерфейс может реализовать более чем один класс, и один класс может реализовать более чем один интерфейс. Множество интерфейсов, реализованных одним классом, называется его **предоставленным интерфейсом**. Это — обещание класса своим клиентам, что его экземпляры обеспечат средства интерфейса. Предоставленные интерфейсы определяются в UML 2.0 отношением зависимости между классом и интерфейсом, реализованным данным классом. Это называется **зависимостью реализации**.

Рис. 9.16 показывает UML-нотацию для зависимости реализации. Класс, из которого исходит стрелка, реализует интерфейс, на который указывает стрелка (обратите внимание, что стрелки показываются здесь пунктирами, но UML 2.0 использует сплошные линии). Класс `Class1` реализует оба интерфейса. Класс `Class2` реализует только `Interface2`.

Классы обеспечивают реализацию (код) всех операций своих интерфейсов. В случае, когда есть атрибуты, класс обещает, что любой из его экземпляров будет содержать информацию о типе и множественности атрибута и что он поставит тот атрибут любому объекту-клиенту, но сам класс не должен иметь этого атрибута в своей реализации [106].

Зависимость использования

После своего объявления интерфейсы могут использоваться классами (или другими интерфейсами), которые затребуют их. Говорят, что класс (или интерфейс) использует **требуемые интерфейсы**. Требуемые интерфейсы определяют сервисы, в которых нуждается класс (или интерфейс), таким образом,

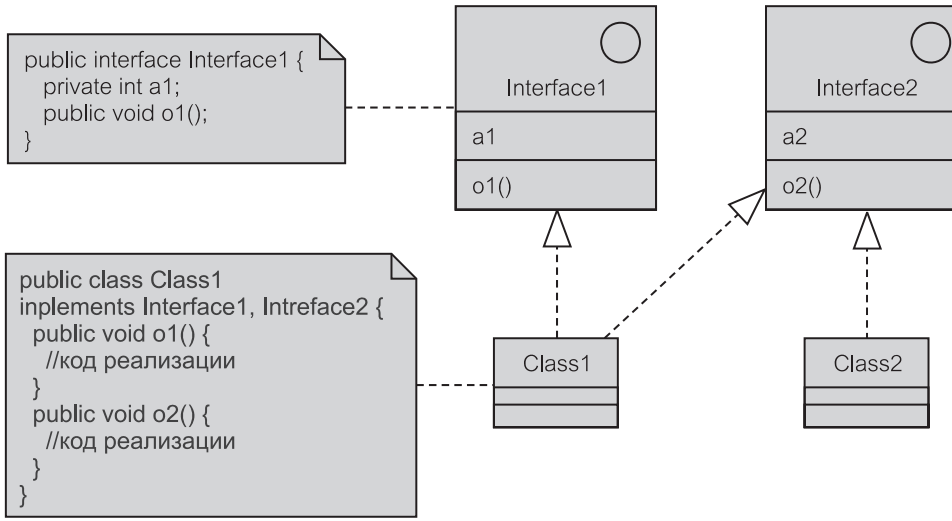


Рис. 9.16. Зависимость реализации

чтобы он мог выполнить свои собственные сервисы для своих клиентов. Требуемые интерфейсы определены в UML 2.0 отношением зависимости между классом (или интерфейсом) и требуемым интерфейсом. Это называется **зависимостью использования**.

Рис. 9.17 показывает UML-нотацию для зависимости использования. Класс или интерфейс, находящийся в начале стрелки, использует интерфейс, расположенный в конце стрелки (использование стереотипа «`uses`» в UML 2.0 не обязательно, но это общепринято). Класс `Class1` использует интерфейс `Interface1`, который, в свою очередь, использует `Interface2`.

`Class1` содержит метод `do1()`, который вызывает услуги операции `o1()`. В статическом коде неясно, какая реализация требуемого интерфейса будет выполнять услугу. Это может быть экземпляр любого класса, который реализует `Interface1`. Точный экземпляр будет определен во время выполнения, когда выполняемый экземпляр класса `Class1` задает величину элемента данных `myInterface`, чтобы обратиться к конкретному объекту конкретного класса.

Устранение циклических зависимостей с интерфейсами

Интерфейсы могут успешно использоваться для уменьшения зависимостей в коде. Программирование с интерфейсами позволяет объектам-клиентам не знать конкретные классы объектов, которые они используют, и классы, которые фактически реализуют эти интерфейсы. Один из наиболее важных принципов многократно используемого, ремонтопригодного и расширяемого объектно-ориентированного проекта таков: «Программа для интерфейса, а не для реализации» [32].

Предыдущее обсуждение показало, что циклические зависимости в системе должны быть устранены любой ценой. Оказывается, что интерфейсы

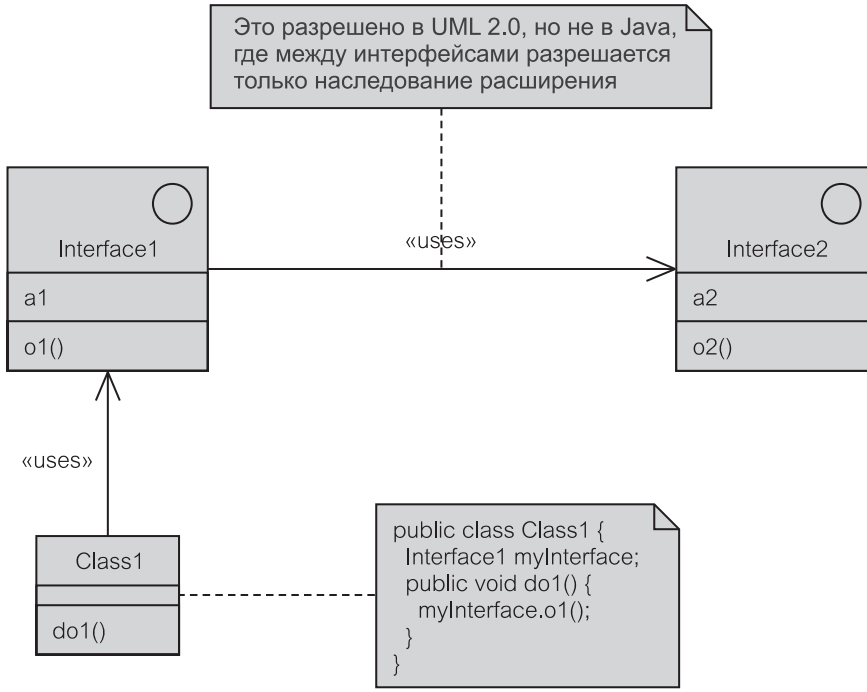


Рис. 9.17. Зависимость использования

представляют наиболее мощное оружие в борьбе с циклическими зависимостями [66]. Чтобы понять, почему дело обстоит именно так, рассмотрим пример циклической зависимости между пакетами `presentation` (представление) и `control` (управление) на рис. 9.18.

Цикл на рис. 9.18 вызван использованием классом `CInit` (класс «инициализация» пакета `control`) сервисов класса `PPrimaryWindow` (класс «первичное окно» пакета `presentation`) и использованием классом `PDialogBox` (класс «диалоговое окно» пакета `presentation`) сервисов класса `CActioner` (класс «исполнитель» пакета `control`). Две явные однонаправленные ассоциации устанавливают передачу сообщений между этими классами. Цикл особенно неприятен, потому что он образован между пакетами со стереотипом «`layer`» (уровень), и это нарушает принцип нисходящих зависимостей между уровнями, рассмотренный в разделе 9.1.3.

Решение дилеммы заключается в квалифицированном включении интерфейса в проект [66], как показано на рис. 9.19. В рассматриваемом примере цикл нарушается добавлением интерфейса `ICPresenter` (интерфейс «представитель» пакета `control`) в пакет `control`. Интерфейс определяет метод `do2()`, необходимый для класса `CInit`, который находится в том же самом пакете. Но интерфейс фактически реализован классом `PPrimaryWindow` в пакете `presentation`. `CInit` использует интерфейс, а `PPrimaryWindow` реализует его.

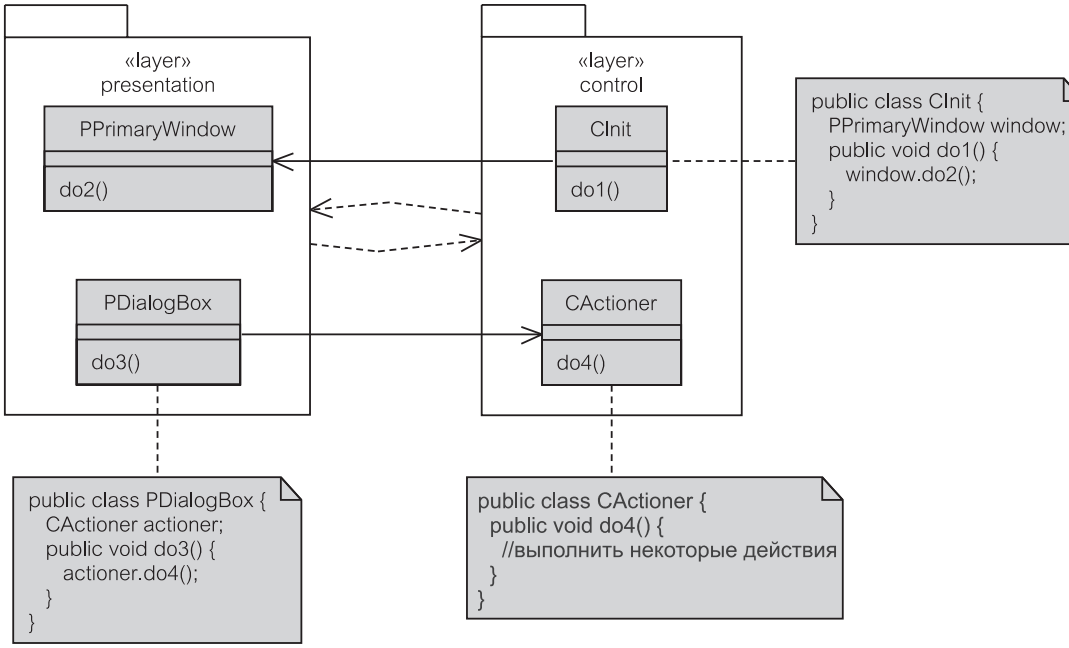


Рис. 9.18. Циклическая зависимость

Обратите внимание, что для того чтобы нарушить цикл, интерфейс и класс, который реализует его, должны находиться в различных пакетах. Это может быть удивительно для программиста-новичка, но это не должно быть открытием для проектировщика, ответственного за структурное проектирование системы. Часто более желательно поместить интерфейсы в пакет, который использует их, а не в пакет, который реализует их [66]. Фулер называет это **паттерном Отделенного интерфейса** [31]. Как альтернатива, интерфейсы могут быть выделены в отдельный пакет[2].

9.1.8. Обработка событий

Зависимости методов (раздел 9.1.6) используются в *синхронной связи* между клиентом и поставщиком сервиса. Сообщение от объекта-клиента просит, чтобы объект-поставщик выполнил сервис. Интерпретация сообщения и средства выполнения его — на усмотрение объекта-поставщика (это могло бы быть *делегированием* работы другому объекту).

В структурном проектировании синхронные сообщения нужно рассматривать отдельно от **асинхронной связи**, где методы «инициализируются», чтобы обслужить асинхронные **события**. Здесь в обработке событий происходит

[2] В компьютерной литературе слово pattern (шаблон) часто не переводится, и просто используется термин «паттерн». Это довольно устоявшийся прием. Мы будем следовать этой практике, чтобы различать термин pattern — шаблон программного продукта и термин framework — шаблон структуры программы, который будем переводить как шаблон или структурный шаблон. (*Прим. перев.*)

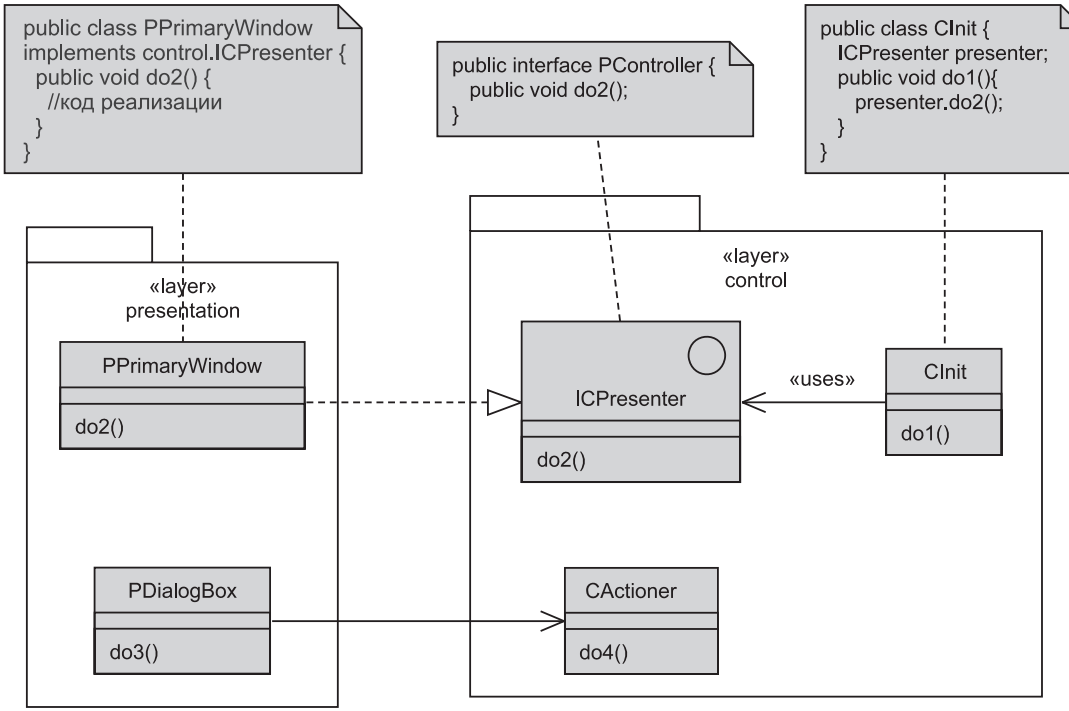


Рис. 9.19. Использование интерфейса для исключения циклической зависимости между методами

отделение создателя события (**объект-издатель**) от различных **приемников**/**наблюдателей** событий (обычно называемых **объектами-подписчиками**), которые хотят знать о возникновении события и которые будут выполнять свои собственные, возможно, различные действия.

В больших системах выполнять подписку может отдельный **объект-регистратор**, который обеспечивает «взаимодействие» между издателем и подписчиками. Чтобы зарегистрировать подписчика у объекта-издателя, объект-регистратор, действующий в интересах подписчика, вызывает метод издателя `addActionListener()` (добавить приемник операций) с объектом-подписчиком в качестве аргумента. Если же никакого объекта-регистратора нет, объект-подписчик непосредственно вызовет метод `addActionListener()`.

Метод, который перехватывает событие (типа нажатия мышью объекта `JButton` — класс «кнопка» пакета Swing — см. раздел 16.2), посылает сообщение «инициализации события» *объекту-издателю*. Обычно сообщение «инициализации события» и метод, который обрабатывает его, находятся в том же самом объекте-издателе. Однако метод, размещающий сообщение «инициализации события», зависит исключительно от метода, обрабатывающего это сообщение. Может даже быть так, что никакому объекту-подписчику событие не понадобится, и не будет никакой зависимости вообще.

Обычно объект-издатель создает **объект-событие** — издатель переводит внутренний смысл события в объект-событие (называя его как-то вроде `BCommandButtonEvent` — объект «событие от командной кнопки `B`»). Объект-событие передается (в режиме *обратной связи*) всем подписчикам, которые зафиксировали свои интересы в отношении нажатия кнопки мышью.

Обработка событий и зависимости уровней

Если при *синхронной передаче сообщения* объект-клиент `A` посылает сообщение объекту-поставщику `B`, то `A` зависит от `B`, потому что `A` ожидает некоторые результаты выполнения от `B`. В *асинхронной обработке события* отправитель сообщения — объект-издатель, но передача сообщения обрабатывается как обратный вызов. При обратной связи издатель не имеет никакой информации, да и интереса в том, как подписчик обрабатывает событие. Зависимость существует, но она незначительна с точки зрения структурного проектирования.

Подтверждение связи подписчиков и издателей вызывает более сильную зависимость. Если объект-регистратор добивается подтверждения связи, то это зависит и от издателя, и от подписчика. Если объект-подписчик фиксирует это (подтверждение связи) сам, то он (подписчик) зависит от издателя. Чтобы ослабить зависимости, связанные с подтверждением связи, подписчиков можно передавать методам регистрации в качестве аргументов, представленных как интерфейсы (см. ниже). В этом случае зависимости ослабляются, но будет требоваться дальнейший анализ программы, чтобы определить класс подписчика. Анализ включает определение классов, которые реализуют интерфейс подписчика.

Для удобства чтения в следующих примерах мы будем твердо придерживаться двух соглашений именования. Во-первых, методы, выполняющие регистрацию подписчиков для издателя, называются, начиная с фразы `add` (добавить) и кончая фразой `Listener` (приемник). Имя объекта-события помещается между этими двумя фразами, например `addXXXListener`, где `XXX` — имя объекта-события.

Второе соглашение применяется к методам, которые инициализируют события в классе-издателе. Класс-издатель будет обычно содержать «нормальные» методы, так же как и методы, инициализирующие события. Чтобы различать эти два вида методов, каждый метод, инициализирующий события, начинается с фразы `fire` (инициализировать) типа `fireCommandButtonEvent()` (инициализация события от командной кнопки).

Метод `fire` пробегает список подписчиков и для каждого подписчика вызывает его собственный метод `process` (обработка). Для удобства (это не является правилом) имя метода `process` может начинаться с фразы `process`, например, `processCommandButtonEvent()` (обработка события от командной кнопки). Метод `fire` обычно не выполняет каких-либо особенных вычислений и не реализует логику программы.

Рис. 9.20 иллюстрирует обработку события, включая представление и управление пакетами, имеющими стереотипы «`layer`» (уровень). `CActioner` (класс «исполнитель» пакета `control`) — единственный под-

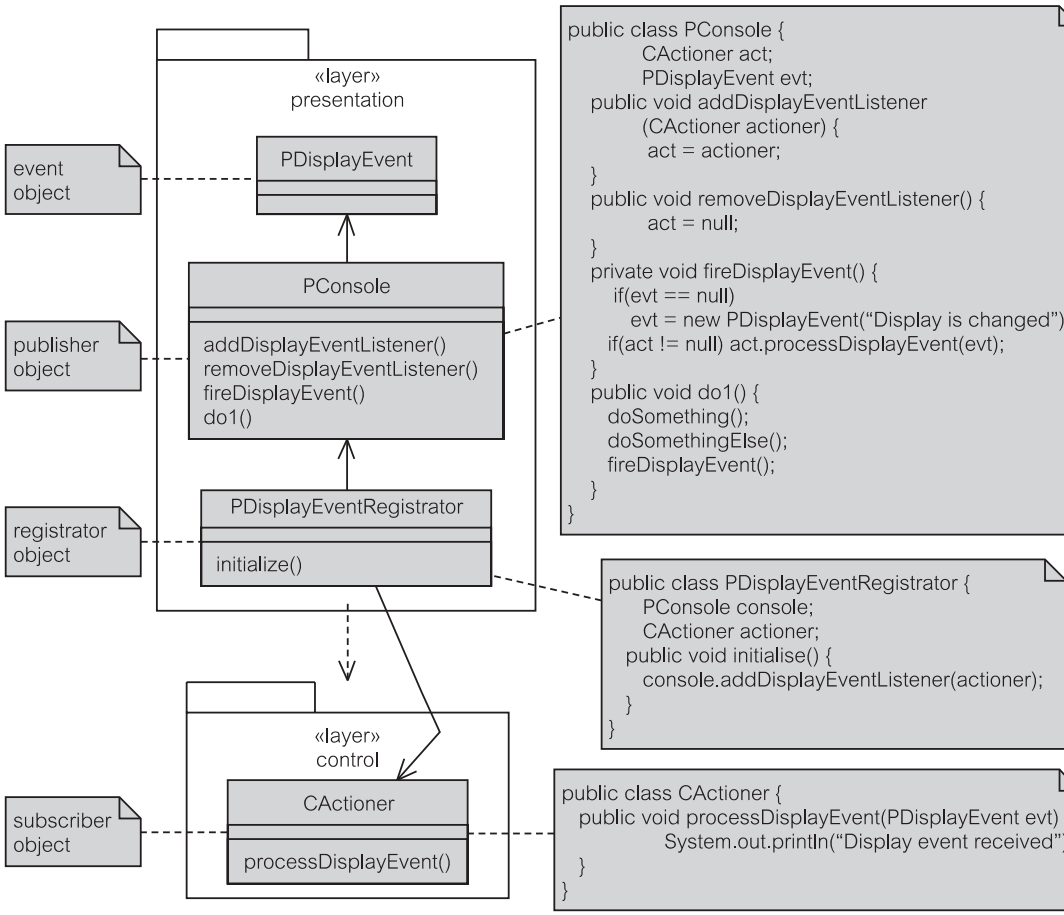


Рис. 9.20. Обработка событий и зависимости уровней

писчик на события класса `PConsole` — класс «консоль» пакета `presentation` (`PDisplayEvent` — класс «событие отображения» пакета `presentation`). `PDisplayEventRegistrator` (регистратор событий отображения) формирует подтверждение связи классам `PConsole` и `CActioner`. Метод `do1()` перехватывает и интерпретирует события, что инициализирует объект `fireDisplayEvent()` (инициализация события отображения). `PConsole` создает объект `PDisplayEvent`. Обратите внимание, что помещение `PDisplayEventRegistrator` в пакет `presentation` (представление) создает приемлемую нисходящую зависимость между пакетами `presentation` и `control` со стереотипом «`layer`» (раздел 9.1.3).

Обработка событий и интерфейсы

На рис. 9.20 класс-регистратор был помещен стратегически в пакет `presentation` со стереотипом «`layer`», а не в пакет `control` с тем же

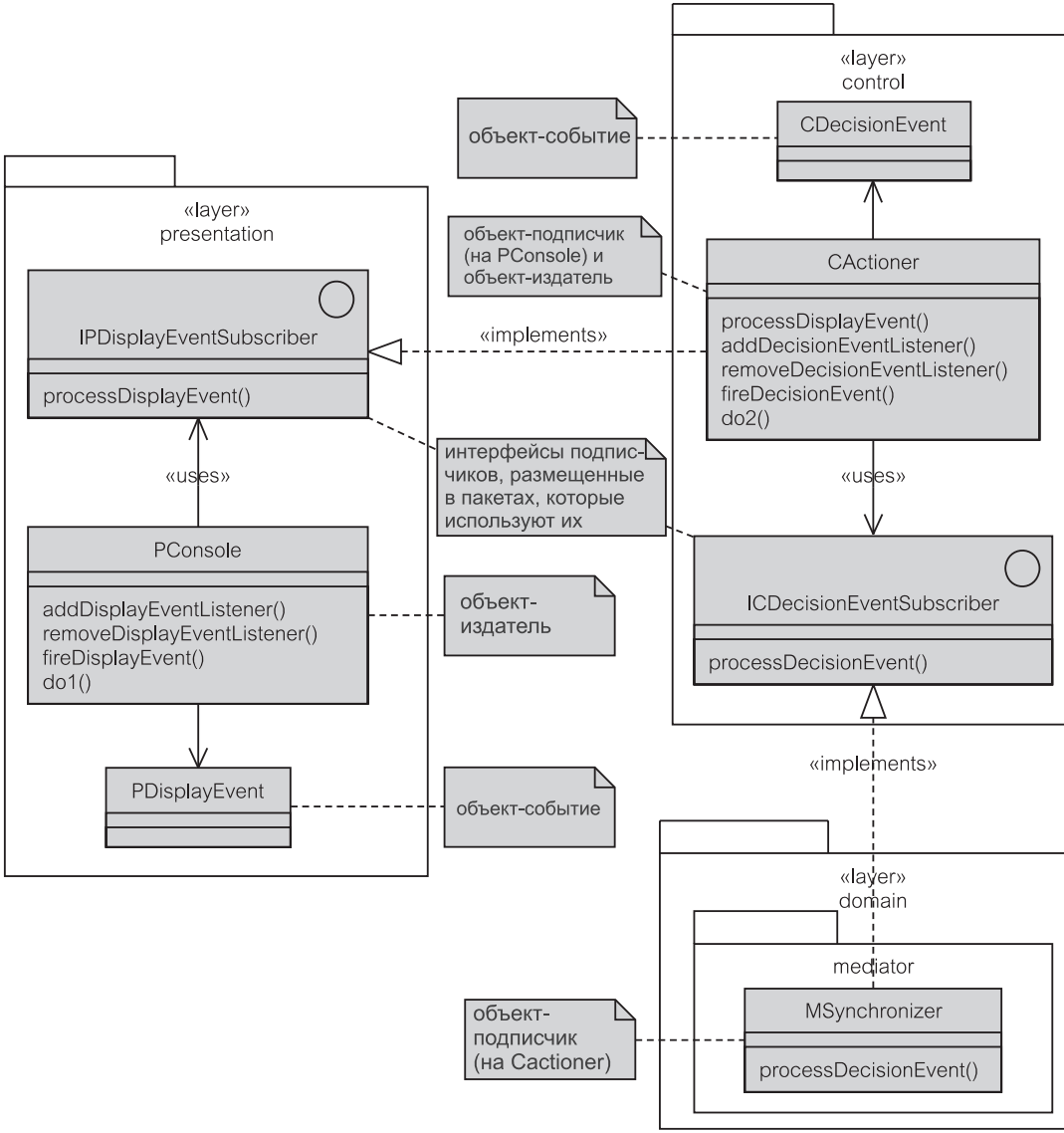


Рис. 9.21. Использование интерфейсов для уменьшения зависимостей от обработки событий

стереотипом, чтобы избежать нежелательных восходящих зависимостей между уровнями (раздел 9.1.3). Но такое стратегическое размещение классов имеет свои пределы, и в большом проекте с большой долей вероятности возможны циклические зависимости из-за передачи сообщений и обработки событий. Например, классы в пакете `presentation` могут также захотеть подписаться на издателей из других пакетов, возможно даже, что это не соседние пакеты.

Разрешение проблемы с циклами в обработке событий снова связано с квалифицированным использованием интерфейсов (раздел 9.1.7). Рис. 9.21 иллюстрирует, как даже нисходящая зависимость на рис. 9.20 может быть уменьшена до пренебрежительно малой величины. Зависимость на рис. 9.20 была вызвана обращением метода `fireDisplayEvent()` класса `PConsole` к `processDisplayEvent()` (обработка события отображения) в `CActioner`. На рис. 9.21 новый интерфейс `IPDisplayEventSubscriber` (интерфейс «подписчик на события отображения» пакета `presentation`) отделяет `PConsole` от `CActioner`. `PConsole` использует `IPDisplayEventSubscriber`, который реализует `CActioner`.

Рис. 9.21 также показывает, как обработка событий может быть объединена вместе, чтобы реализовать сложную систему. `PConsole` создает объект `PDisplayEvent` и использует интерфейс `IPDisplayEventSubscriber`, чтобы известить `CActioner` относительно объекта `PDisplayEvent`. `CActioner` предварительно обрабатывает `PDisplayEvent` с помощью своего метода `processDisplayEvent()` (обработка события отображения) и создает объект `PDecisionEvent` (класс «событие принятия решения» пакета `presentation`).

`CActioner`, имеющий теперь способности издателя, использует интерфейс `ICDecisionEventSubscriber` (интерфейс «подписчик на события принятия решения» пакета `presentation`), чтобы уведомить `MSynchronizer` (класс «синхронизатор» пакета `mediator`) относительно объекта `PDecisionEvent`. `MSynchronizer` обрабатывает `PDecisionEvent` своим методом `processDecisionEvent()` (обработка события принятия решения).

9.1.9. Знакомство

Как сказано в предыдущей части раздела 9.1, хорошее структурное проектирование требует создания таких брандмауэров зависимостей, которые минимизировали бы зависимости между соседними уровнями, устраняли циклические зависимости и удаляли прямые зависимости между несоседними уровнями. Использование делегирования, интерфейсов и обработки событий существенно влияет на эти показатели.

Однако имеются ситуации, когда объекты несоседних уровней могут быть связаны без нарушения принципов хорошего структурного проектирования. Это может произойти, когда объект «знакомится» во время выполнения с другим объектом (в несоседнем уровне) и хочет воспользоваться преимуществом этого знакомства, запрашивая услуги от этого объекта.

Знакомство соответствует ситуации, когда объект передает другой объект как аргумент своего метода. Более точно, объект `A` знакомится с объектом `B`, если другой объект `C` передает `B` к `A` в качестве аргумента сообщения к `A`. Связь объектов в результате знакомства — одна из технологий, соответствующая закону Деметера [58].

Рис. 9.22 иллюстрирует знакомство. `CActioner` (класс «исполнитель» пакета `control`) использует `retrieveEmployee()` (извлечь информацию о служащем), чтобы извлечь объект `EEmployee` (класс «служащий» пакета `entity`) из БД. Процесс извлечения не объясняется, но, вероятно,

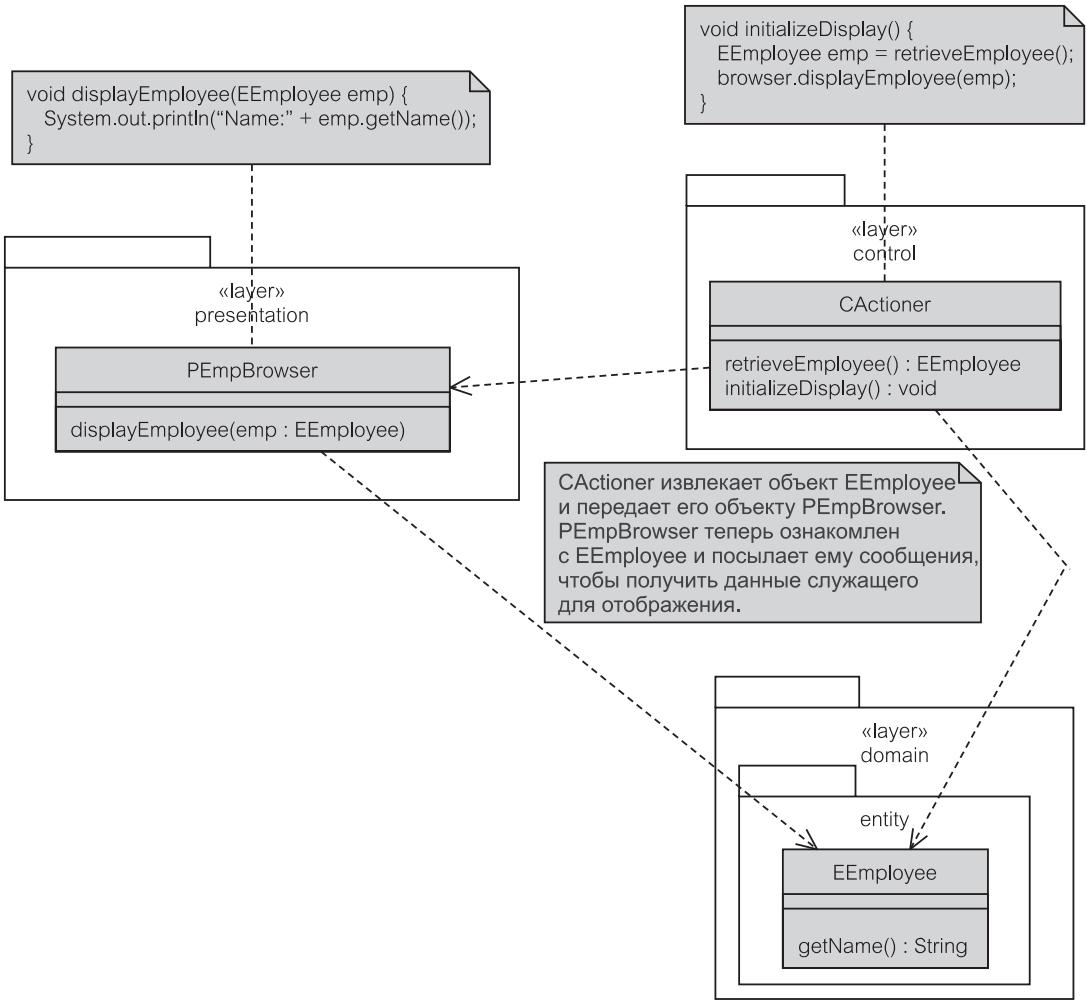


Рис. 9.22. Знакомство

`CActioner` делегирует запрос до уровня, ответственного за связь с БД. Как только информация о служащем будет извлечена, инициализируется объект `EEmployee`. `CActioner` передает `EEmployee` объекту `PEmpBrowser` (класс «браузер служащих» пакета `presentation`) в сообщении `displayEmployee(emp)` (отобразить информацию о служащем). `PEmpBrowser` теперь ознакомлен с `EEmployee` и посылает сообщение `getName()` (получение имени) к нему, чтобы получить имя служащего для отображения.

Зависимости знакомства и интерфейсы

Как показано на рис. 9.22, знакомство вводит зависимости. **Зависимости знакомства** — *зависимости методов*, возникающие динамически во время

выполнения. Сугубо динамический характер зависимостей знакомства отличает их от «регулярных» зависимостей методов, рассмотренных в разделе 9.1.6.

С точки зрения структуры «динамический» означает, что это трудно видеть и понять. «Динамический» означает также, что невозможно использовать явные ассоциации между классами, чтобы визуализировать (и поэтому узаконить) зависимости знакомства в коде. Иногда приемлемое решение может быть получено за счет включения интерфейсов в проект.

Проект на рис. 9.22 имеет два структурных недостатка:

- Пакет `control` зависит в восходящем направлении от пакета `presentation` (потому что `CActioner` вызывает `browser.displayEmployee(emp)` объекта `PEmpBrowser`).
- Пакет `presentation` зависит от несоседнего пакета `domain` (предметная область) (потому что `PEmpBrowser` вызывает `emp.getName()` объекта `EEmployee`).

Оба недостатка могут быть устранены с помощью интерфейсов, как показано на рис. 9.23. Технология подобна той, которая удаляет циклические зависимости с помощью интерфейсов (раздел 9.1.7). `ICEmpBrowser` (интерфейс «браузер служащих» пакета `control`) нарушает восходящую зависимость от `control` до `presentation`. `IPEmployee` (интерфейс «сотрудник» пакета `presentation`) нарушает зависимость знакомства от `presentation` до `domain`.

Пакет знакомств

Использование интерфейсов и обработки событий (в пределах возможностей, рассмотренных в разделах 9.1.7 и 9.1.8) в более сложных сценариях может и не обеспечить очевидное структурное решение зависимостей знакомства. Даже решение на рис. 9.23 скорее неудовлетворительно, потому что имеется зависимость реализации (раздел 9.1.7) от `EEmployee` до `IPEmployee`, занимающих несоседние пакеты.

Лучшее решение может быть получено от рассмотрения знакомства как такового, и которое требует самостоятельного управления. С этой целью может быть введен отдельный пакет знакомств. **Пакет знакомств** — это автономный пакет, состоящий только из интерфейсов. Пакет не представляет ни уровень, ни часть иерархии уровней.

Рис. 9.24 демонстрирует, как пакет `acquaintance` (знакомство) отделяет зависимости знакомств в изолированную проблему, которая может управляться независимо. Модель является примером принципа разделения задач, который должен быть характерной чертой любого хорошего структурного проекта [55]. Пакет `acquaintance` состоит только из интерфейсов. `IAEmployee` (интерфейс «служащий» пакета `acquaintance`) расширен классом `IAEmployeeWithSalary` (интерфейс «служащий и жалованье» пакета `acquaintance`), который в свою очередь реализован классом `EEmployee`.

Как и на рисунках 9.15 и 9.16, метод `initializeDisplay()` (инициализация отображения) класса `CActioner` использует метод `retrieveEmployee()`, чтобы извлечь из БД и инициализировать объект `EEmployee` (этот процесс использует другие объекты, не показанные на ди-

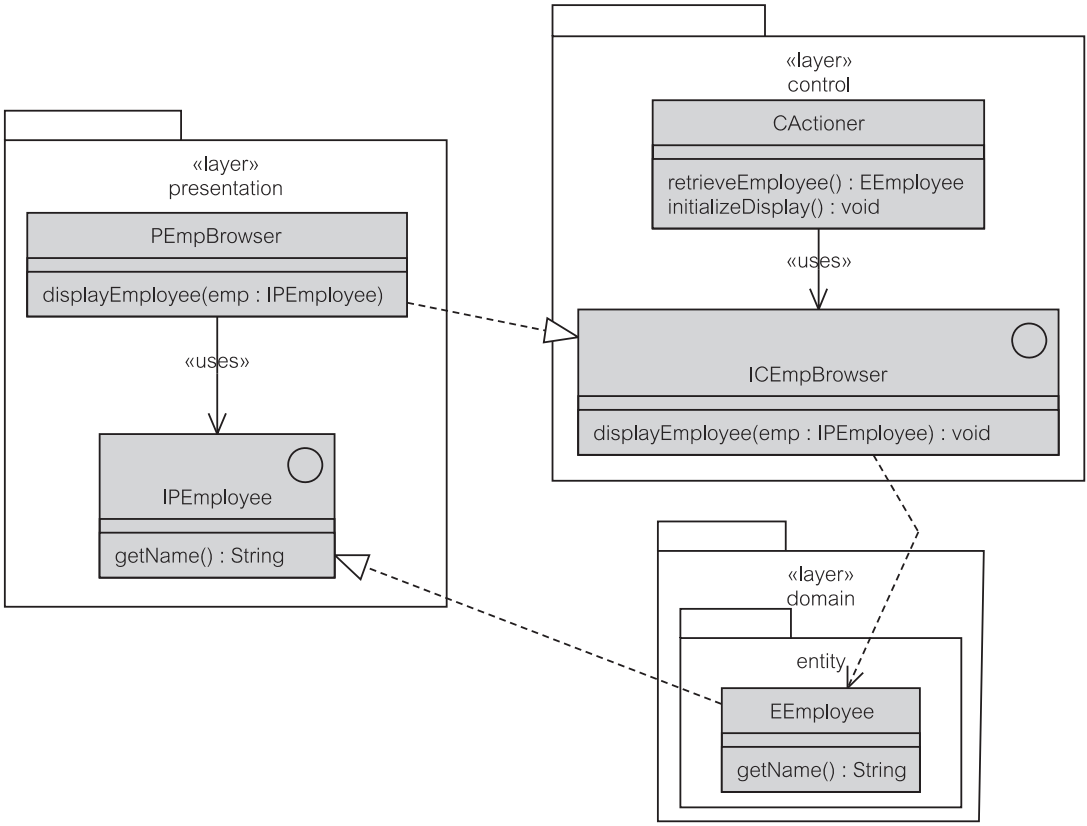


Рис. 9.23. Использование интерфейсов для понижения зависимостей

аграмме). Хотя *предоставленным интерфейсом* (раздел 9.1.7) класса `EEmployee` является `IAEmployeeWithSalary`, классу `CActioner` нужен только интерфейс `IAEmployee` (то есть вся информация о служащем, кроме жалованья).

Извлеченный объект (названный как `IAEmployee`) передается с помощью метода `displayEmployee(emp: IAEmployee)` — отобразить служащего — объекту `PEmpBrowser`. `PEmpBrowser` теперь знаком с `EEmployee` и использует его в сервисе `getName()`, который является частью *требуемого интерфейса* (раздел 9.1.7) класса `PEmpBrowser`. Если иерархией уровней является `presentation — control — domain (entity) — foundation` (представление — управление — предметная область (сущность) — основание), то `PEmpBrowser` использует `IAEmployee` для *нисходящего* доступа к `EEmployee` в *несоседнем пакете*.

В нижней части рис. 9.24 `FUpdater` требует интерфейс `IAEmployeeWithSalary` в методе `saveEmployee(emp: IAEmployeeWithSalary)`. Если какой-то клиент передает этому методу объект `EEmployee`, то метод `saveEmployee()` (сохранить служащего) мо-

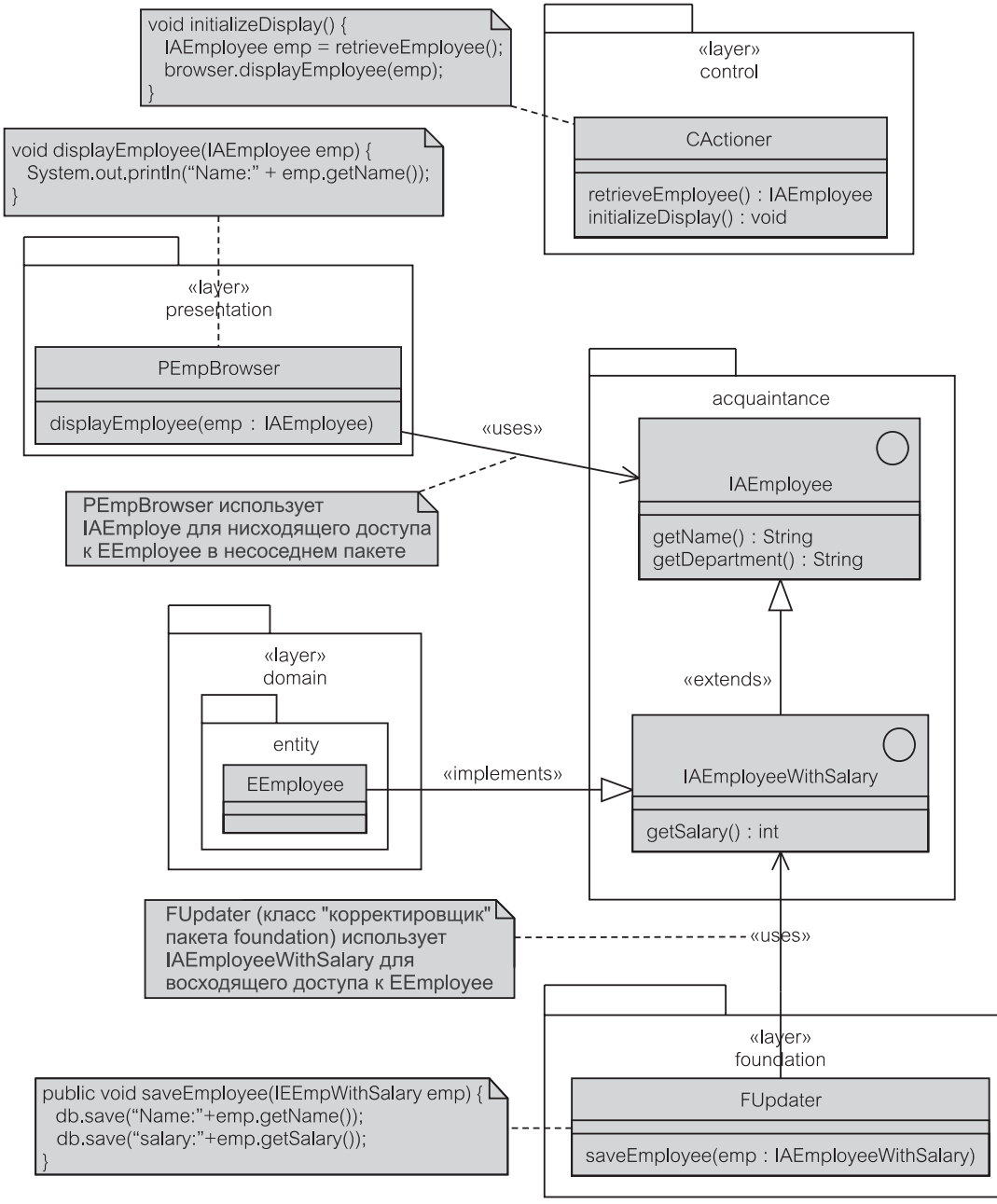


Рис. 9.24. Использование пакета знакомств

жет использовать `getSalary()` (получить величину зарплаты) или `getName()` (получение имени) для предварительного сохранения информации о служащем в БД. `FUpdater` использует `IAEmployeeWithSalary` для *восходящего* доступа к `EEmployee`.

9.2. Структурные шаблоны

Шаблон представляет собой конструкцию, на которой строится все решение проблемы. В технологии объектов шаблон — технология повторного использования [61]. Технология применяется к повторному использованию проекта в противоположность к повторному использованию кода. Шаблон обеспечивает скелет решения проблемы, который должен быть настроен и расширен, чтобы он мог выполнять полезную функцию. Настройка включает написание определенного кода, который «заполняет пробелы» в шаблоне (то есть, который реализует различные элементы шаблона, приспосабливая их к окончательному структурному и поведенческому проекту).

Наиболее известные шаблоны повторного использования — системы планирования ресурсов предприятия (enterprise resource planning — ERP) типа SAP или JDEdwards. ERP-системы предлагают и проектирование, и повторное использование кода. Это базовые пакеты ПО. ПО настраивается и расширяется в соответствии с перечнем потребностей клиента. Настройка должна быть выполнена в пределах скелетного проекта, заданного шаблоном.

Шаблон обеспечивает структурное проектирование системы. В этом смысле, каждый шаблон является **структурным шаблоном** независимо от того, является ли он программным продуктом типа ERP-системы или только идеей проекта и условием, наложенным на коллектив разработчиков. Последнее является предметом дальнейшего обсуждения.

9.2.1. Model-View-Controller (MVC)

Один из наиболее известных структурных шаблонов для проекта — **Model-View-Controller** (MVC) — модель-представление-контроллер — шаблон, разработанный как часть среды программирования Smalltalk-80 [52]. MVC оставил существенный след в объектно-ориентированном проектировании. Это классический пример использования принципа разделения задач в объектно-ориентированном проекте. В Smalltalk-80 MVC вынуждает программистов делить прикладные классы на три группы, которые специализируют и наследуют от трех имеющихся в Smalltalk абстрактных классов: Model (модель), View (представление) и Controller (контроллер).

Объекты-модели (Model) представляют объекты данных — бизнес-сущности и бизнес-правила в предметной области приложения. Изменения в объектах-моделях передаются объекту-представлению и объекту-контроллеру с помощью обработки событий. Эта обработка использует технологию издателя/подписчика. Модель является издателем и поэтому ничего не знает о своих представлениях и контроллерах. Объекты-представления и объекты-контроллеры подписываются на объект-модель, но они могут также инициировать изменения в объектах-моделях. Чтобы способствовать этой задаче,

модель обладает необходимыми интерфейсами, которые инкапсулируют бизнес-данные и бизнес-поведение.

Объекты-представления (View) воспроизводят GUI-объекты и представляют состояние модели в формате, необходимом пользователю, обычно в графическом отображении. Объекты-представления отделены от объектов-моделей. Представление подписано на модель, чтобы получать информацию относительно изменений модели и корректировать в связи с этим свое отображение. Объекты-представления могут содержать подпредставления, которые отображают различные части модели. Как правило, каждый объект-представление соединен с объектом-контроллером.

Объекты-контроллеры (Controller) представляют собой события от клавиатуры и мыши. Объекты-контроллеры отвечают на запросы, которые поступают от представлений и которые являются результатами взаимодействий пользователя с системой. Эти объекты присваивают значения нажатиям клавиш, нажатиям мыши и т. д. и преобразовывают их в действия на объекты-модели. Они будут посредниками между объектами-моделями и объектами-представлениями. Отделяя входные воздействия пользователя от визуального представления, они позволяют изменять ответы системы на действия пользователя без изменения GUI-представления, и наоборот, изменять GUI без изменения поведения системы.

Рис. 9.25 иллюстрирует точку зрения актора (пользователя) на связи между MVC-объектами. Линии представляют связи между объектами. События пользователя GUI перехватываются объектами-представлениями и передаются объектам-контроллерам для интерпретации и дальнейших действий. Объединение поведения представления и контроллера в единый объект считается плохой практикой в MVC.

Рассмотрим сценарий, где пользователь активизирует команду меню, чтобы отобразить информацию о клиенте на экране презентаций. Объект View получает событие и передает его своему объекту Controller. Объект Controller

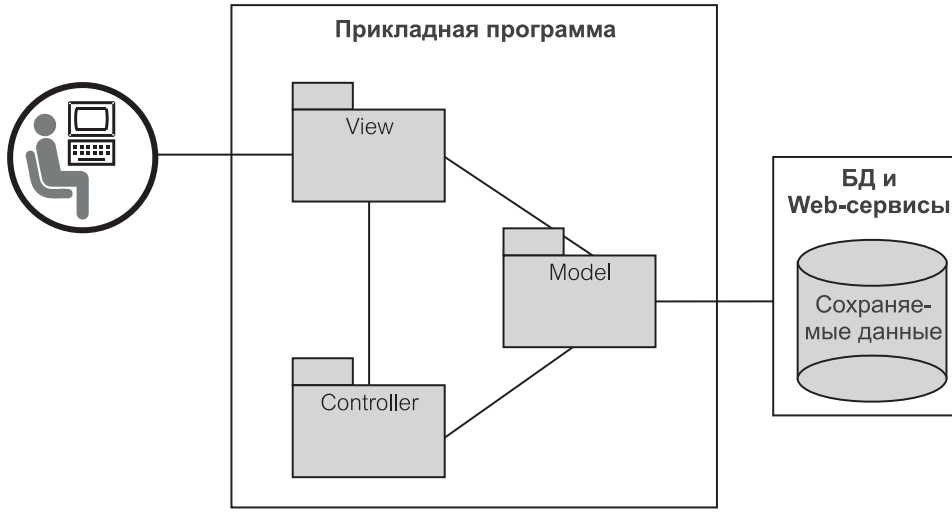


Рис. 9.25. Шаблон MVC

просит модель обеспечить данные о клиенте. Объект Model возвращает данные объекту Controller, который передает их View для отображения. Альтернативно объект Model может передавать данные непосредственно объекту View. Любые дальнейшие изменения состояния Model могут быть переданы объектам View, которые подписались на эти изменения. Таким образом, View может обновить свое отображение, чтобы отразить текущие значения бизнес-данных.

Принцип разделения задач в MVC имеет много преимуществ. Наиболее важные следующие [32, 55]:

- обеспечение раздельной разработки GUI, бизнес-данных и логики уровня модели;
- замена или переход к различным GUI без каких-либо радикальных изменений в модели;
- реорганизация или перепроектирование модели при сохранении GUI-представления пользователю;
- разрешение многих представлений одного и того же состояния модели;
- изменение способа реакции GUI на события пользователя без изменения GUI-представления (контроллер представления может быть даже изменен во время выполнения);
- разрешение возможности выполнения модели без GUI (например, для тестирования или в случае пакетной обработки).

9.2.2. Presentation-Control-Mediator-Entity-Foundation

Принципы MVC представляют наиболее современные структурные шаблоны и связанные с ними паттерны. Unified Process (UP) — унифицированный процесс — использует принципы MVC при разделении классов на граничные объекты, объекты управления и объекты-сущности — понятия, взятые из методики Objectory (объекты) [43]. Граничные объекты соответствуют объектам-представлениям (View) MVC, объекты управления — объектам-контроллерам (Controller) MVC, а объекты-сущности представляют объекты-модели (Model) MVC.

MVC-подход повлиял фактически на все структурные шаблоны, которые подчеркивают потребность в иерархических уровнях объектов [17, 31], чтобы управлять зависимостями объектов (раздел 9.1). Данный раздел представляет один такой шаблон, используемый в учебном примере книги. Шаблон называется **Presentation-Control-Mediator-Entity-Foundation** (PCMEF) — представление, управление, посредник, сущность, основание. Принципы PCMEF подобны другим популярным схемам выделения уровней [31].

PCMEF — уровневый структурный шаблон, организованный в виде вертикальной иерархии. Каждый уровень представляет собой пакет, который может содержать другие пакеты. Пакеты в пределах каждого уровня называются сегментами (разделы 9.1.2 и 9.1.3). Другими словами, структура состоит из вертикальных уровней, которые могут быть разделены на сегменты. Когда система увеличивается, сегменты сами могут быть организованы в систему подуровней, то есть сегменты могут быть слоистыми. Выделение уровней сегментов данная книга не рассматривает.

В строго уровневой структуре более высокие уровни используют услуги более низких уровней (и поэтому зависят от них), но не наоборот. Строго уровневая структура маловероятна на практике (раздел 9.1.3). Однако при правильном проектировании уровней можно получить структуру проекта, где более высокие уровни будут *неустойчивы*, а более низкие уровни все более и более *устойчивы* (раздел 9.1.3). Такой шаблон обеспечивает нисходящие зависимости и ограничивает восходящие зависимости во взаимодействии объектов.

Уровни PCMEF

PCMEF-шаблон состоит из четырех уровней (layer): `presentation` (представление), `control` (управление), `domain` (предметная область) и `foundation` (основание). Уровень `domain` содержит два предопределенных пакета: `entity` (сущность) и `mediator` (посредник). По сравнению с MVC-шаблоном `presentation` соответствует View MVC, `control` — Controller и `entity` — Model; `mediator` и `foundation` не имеют MVC-аналогов.

Главные зависимости PCMEF — нисходящие, как показано стрелками на рис. 9.26 — `presentation` зависит от `control`, `control` зависит от `domain` (от `mediator` и, возможно, но не обязательно, от `entity`), `mediator` зависит от `entity` и от `foundation`. Восходящие зависимости реализуются через свободные связи, облегченные интерфейсами, обработкой событий, пакетом знакомств и подобными технологиями, рассмотренными в разделе 9.1. Зависимости допускаются только между соседними уровнями.

Вертикальные уровни PCMEF задают структуру системы, инвариантную для всех итераций процесса пошаговой разработки. Нисходящие зависимости сделаны явными в соответствии с требованием задания однонаправленных ассоциаций между зависимыми классами в соседних уровнях. Горизонтальные зависимости в пределах уровней практически неограниченны, пока не приводят к циклическим зависимостям.

В Java зависимости между пакетами преобразуются в операторы import — импорт (раздел 9.1.1). Зависимости, показанные на рис. 9.26, соответствуют операторам импорта в листинге 9.1:

Листинг 9.1. Импорт пакетов

```
package presentation;
import control.*;

package control;
import domain.entity.*;
import domain.mediator.*;

package entity;
package mediator;
import entity.*;
import foundation.*;

package foundation;
```

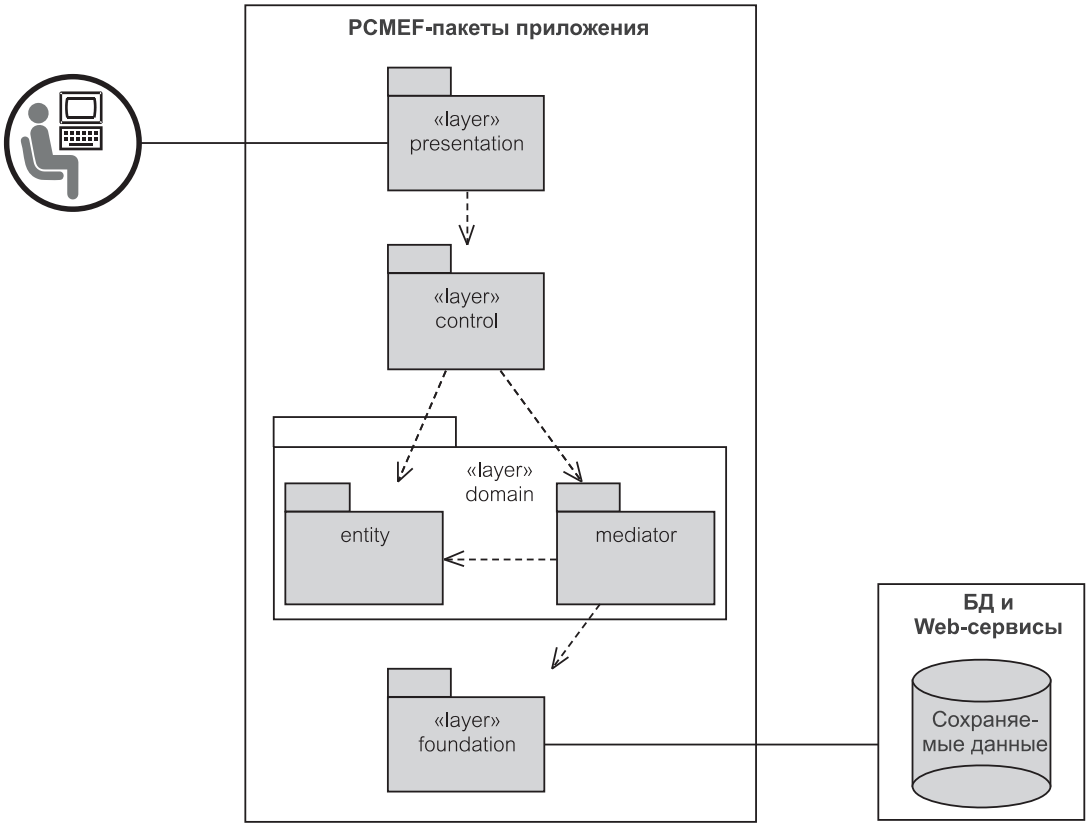


Рис. 9.26. PCMEF-шаблон

Уровень `presentation` содержит классы, которые определяют GUI-объекты. В среде Microsoft Windows многие классы `presentation` являются подклассами MFC-библиотеки (Microsoft Foundation Classes — основные классы Microsoft). В среде Java классы `presentation` основаны на классах и интерфейсах библиотеки Java Swing. Пользователь связывается с системой через классы `presentation`. Соответственно, класс, содержащий функцию `main` (основная) программы, обычно размещается в пакете `presentation` (альтернативно это может быть связано с пакетом `control`).

Уровень `control` управляет запросами уровня `presentation`. Он состоит из классов, ответственных за обработку взаимодействий пользователя (передаваемых объектам `control` от объектов `presentation`). В результате этого `control` ответствен за бóльшую часть логических, алгоритмических программных решений, основных вычислений и поддержания сеансов работы для каждого пользователя.

Пакет `entity` уровня `domain` имеет дело с запросами уровня `control`. Он содержит классы, представляющие «бизнес-объекты». Они хранят (в памяти программы) объекты, извлеченные из БД или созданные

с тем, чтобы поместить их в БД. Многие классы-сущности — контейнерные классы.

Пакет `mediator` уровня `domain` устанавливает канал связи, который будет посредником между классами пакетов `entity` и `foundation`. Посредничество обеспечивает две главные цели: во-первых, изолировать два пакета так, чтобы изменения в любом из них могли быть представлены независимо, и, во-вторых, устранить потребность в классах `control`, чтобы связаться непосредственно с классами `foundation` всякий раз, когда должны быть извлечены из БД новые объекты пакета `entity` (такие запросы от классов пакета `control` затем передаются через классы пакета `mediator`).

Уровень `foundation` отвечает за все коммуникации с БД и Web-сервисами, которые управляют сохраняемыми данными, необходимыми для прикладной программы. Это именно то, где установлены связи с БД и Web-серверами, формируются запросы к сохраняемым данным и инициируются транзакции БД.

Принципы PCMEF

Кроме уровневой структуры шаблон PCMEF определяет принципы проектирования (раздел 9.1) и поддерживает паттерны проекта (раздел 9.3), предназначенные должным образом управлять зависимостями объектов и предлагать решения, которые являются понятными, ремонтопригодными и расширяемыми. Основные PCMEF-принципы следующие:

- принцип нисходящей зависимости (Downward Dependency Principle — DDP);
- принцип восходящего уведомления (Upward Notification Principle — UNP);
- принцип соседней связи (Neighbor Communication Principle — NCP);
- принцип явной ассоциации (Explicit Association Principle — EAP);
- принцип устранения циклов (Cycle Elimination Principle — CEP);
- принцип именования классов (Class Naming Principle — CNP);
- принцип использования пакета знакомств (Acquaintance Package Principle — APP).

Принцип нисходящей зависимости утверждает, что основная структура зависимостей — нисходящая. Объекты более высоких уровней зависят от объектов более низких уровней. Следовательно, низшие уровни более устойчивы, чем высокие. Их трудно изменять, но (как это ни парадоксально) не всегда трудно расширить [66]. Интерфейсы, абстрактные классы, доминирующие классы и подобные им средства (раздел 9.1.7) должны инкапсулировать устойчивые пакеты, чтобы их можно было расширять при необходимости.

Принцип восходящего уведомления поддерживает слабую связь между нижними и верхними уровнями. Это может быть получено использованием асинхронной связи, основанной на обработке событий (раздел 9.1.8). Объекты в более высоких уровнях действуют как подписчики (наблюдатели) изменений состояний более низких уровней. Когда объект (издатель) в более низком уровне изменяет свое состояние, он посылает уведомление своим подписчикам. В ответ подписчики могут связаться с издателем (теперь в нисходящем направлении) так, чтобы их состояния были синхронизированы с состоянием издателя.

Принцип соседней связи требует, чтобы пакет мог непосредственно связываться только со своими соседними пакетами. Этот принцип гарантирует, что систему нельзя представить в виде непонятной сети взаимодействия объектов. Чтобы обеспечить этот принцип, передача сообщения между несоседними объектами использует делегирование (раздел 9.1.6). В более сложных сценариях для интерфейсов групп может использоваться пакет `acquaintance` — знакомство (раздел 9.1.9), чтобы помочь во взаимодействии, использующем отдаленные пакеты.

Принцип явной ассоциации реализуется в документах, явно разрешающих передачу сообщений между классами (раздел 9.1.6). Принцип преобладающе используется для классов с нисходящими зависимостями (см. выше принцип нисходящей зависимости) и для классов, часто использующих сотрудничество. Ассоциации, которые являются результатами этого принципа, однонаправлены (иначе они создали бы циклические зависимости). Однако нужно помнить, что не все ассоциации между классами существуют только из-за передачи сообщений между ними. Например, двунаправленные ассоциации могут быть необходимы, чтобы реализовать ссылочную целостность между классами.

Принцип устранения циклов гарантирует, что циклические зависимости между уровнями, пакетами и классами в пределах пакетов устранены. Циклические зависимости нарушают принцип разделения задач и могут стать главным препятствием возможности многократного использования. Циклы можно устранить, создавая новый пакет конкретно для данной цели (раздел 9.1.2) или с помощью организации одного из путей связи в цикле через интерфейс (раздел 9.1.7).

Принцип именования классов дает возможность определить по имени класса, какому пакету он принадлежит. С этой целью каждое имя класса имеет в PCMEF префикс в виде первой буквы имени пакета (например, `EInvoice` — класс пакета `entity`). Тот же самый принцип применяется для интерфейсов. Каждое имя интерфейса имеет префикс в виде двух заглавных букв: первая — буква «`I`» (означающая, что это интерфейс), а вторая — буква, определяющая пакет (например, `ICInvoice` — интерфейс пакета `control`).

Принцип использования пакета знакомств — последствие принципа соседней связи. Пакет `acquaintance` состоит из интерфейсов, которые передают вместо конкретных объектов эти объекты в качестве аргумента при вызове методов. Интерфейсы могут быть реализованы в любом PCMEF-пакете. Это позволяет осуществлять эффективную связь между несоседними пакетами при централизации управления зависимостями в единственном пакете знакомств. Потребность в пакете `acquaintance` объяснялась в разделе 9.1.9 и обсуждается снова ниже в контексте PCMEF.

Знакомство в PCMEF+

Знакомство возникает тогда, когда внутри метода сообщение посылается объекту, являющемуся параметром этого метода (раздел 9.1.9). Поскольку знакомство приобретается динамически во время выполнения, должны быть предприняты специальные меры, чтобы избежать чрезмерных связей объек-

тов из-за знакомств. Принцип использования пакета знакомств гарантирует, что будет удовлетворен принцип соседней связи в присутствии знакомства.

В PCMEF знакомство в *нисходящем* направлении к объектам в соседних пакетах никогда не требуется, потому что такие объекты связывают ассоциации (это означает, что сообщение может быть послано в атрибуте объекта-клиента (`this`), который связывается с объектом-поставщиком). Знакомство в *восходящем* направлении к объектам в соседних пакетах должно использовать интерфейсы, чтобы устранить циклы (раздел 9.1.9). Знакомство, которое в действительности охватывает *несоседние* объекты, должно быть проведено через интерфейсы, сгруппированные в пакете `acquaintance`. Рис. 9.24 приводит хороший пример этого.

Рис. 9.27 показывает, как **пакет `acquaintance`** эффективно расширяет PCMEF-шаблон в шаблон **PCMEF+**. Интерфейсы пакета `acquaintance` могут быть реализованы и/или использованы классами любого PCMEF-пакета (см. объяснение реализации и зависимостей использования в разделе 9.1.7). Внешний пакет на рис. 9.27, содержащий PCMEF-пакеты, используется только для целей визуализации — практически он не существует.

Снова обратите внимание, что пакет `acquaintance` группирует интерфейсы как для восходящих связей, так и для нисходящих связей с несоседними объектами. В общем, имеются пять возможностей для проведения связи объектов через интерфейсы `acquaintance`:

1. `F` использует интерфейсы `A`, реализованные в `D`, `C` или `P`;
2. `D` использует интерфейсы `A`, реализованные в `C` или `P`;
3. `C` использует интерфейсы `A`, реализованные в `P`;
4. `P` использует интерфейсы `A`, реализованные в `D` или `F`;
5. `C` использует интерфейсы `A`, реализованные в `F`.

Пакет `acquaintance` можно рассматривать как прикладную библиотеку интерфейсов. Подобно любой библиотеке, пакет `acquaintance` должен быть разработан с учетом его стабильности. Как упомянуто выше в контексте принципа нисходящей зависимости, устойчивый пакет нелегко поддается изменениям, но он может быть расширен, чтобы учесть развитие системы.

То же самое правило применимо и к внешним библиотекам, используемым прикладной программой, типа пакетов Java API. PCMEF-пакеты, использующие типы данных Java, зависят от пакета `java.util`, PCMEF-пакеты, занимающиеся обработкой событий, зависят от `java.awt.event` и т. д. (Относительная) стабильность библиотек Java гарантирует, что подключенные к ним PCMEF-пакеты вряд ли создадут кошмар сопровождения в будущем.

Развертывание PCMEF-уровней

Структурный шаблон PCMEF ничего не говорит относительно развертывания уровней в узлах компьютерной сети. Уровни устроены так, что их можно размещать независимо, как компоненты. **Компонент** определяется в UML 2.0 как «модульная часть системы, которая инкапсулирует ее содержание и чье объявление можно заменить в пределах среды этой системы» [106]. UML 2.0

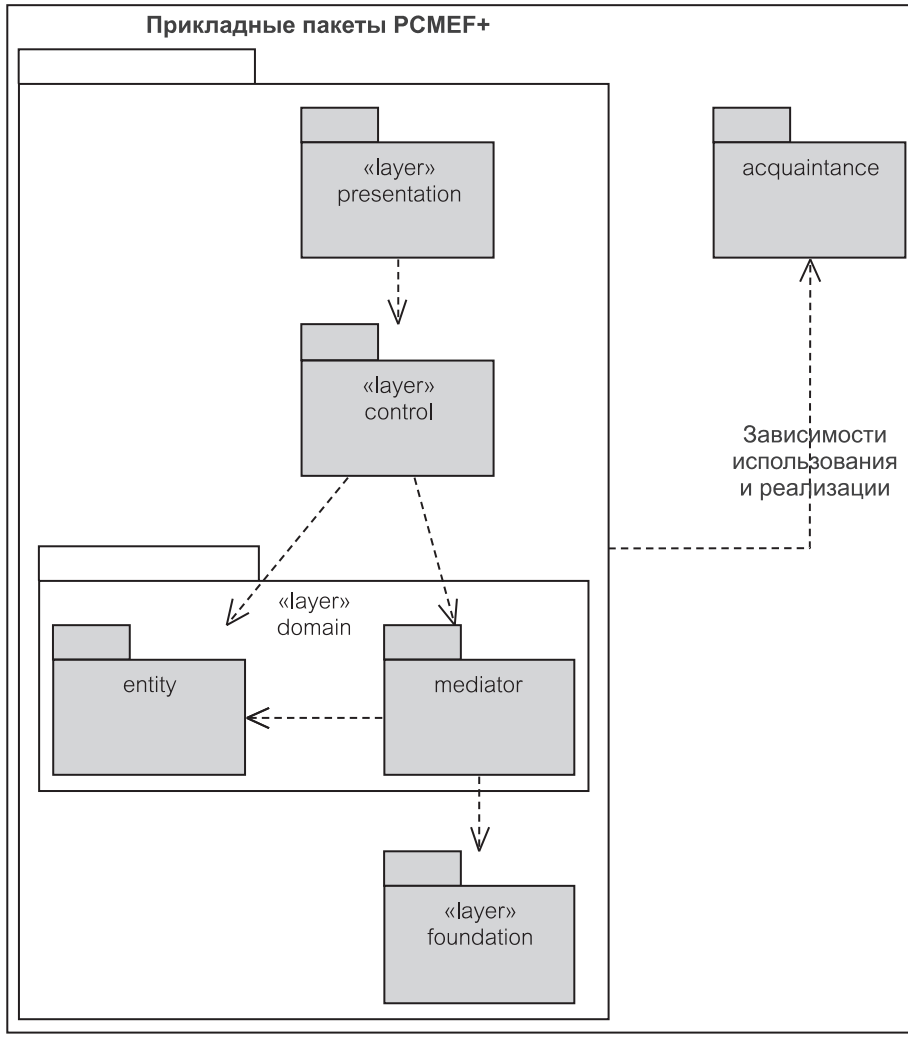


Рис. 9.27. Пакет знакомств в PCMEF+

обеспечивает также диаграмму размещения компонентов, чтобы показать ввод их в действие в среде выполнения.

Развертывание PCMEF-уровней как компонентов завершается некоторой **клиент-серверной структурой**. Как минимум, структура состоит из прикладного клиента и сервера БД/Web-сервера. Это — **двухъярусная структура**, где прикладная программа, расположенная в одном узле компьютерной сети, управляет всеми PCMEF-объектами ПО, а сервер БД/Web-сервер в другом узле управляет сохраняемыми данными (рис. 9.26). Двухъярусная структура применима как к приложениям с многопользовательскими БД, так и к однопользовательским настольным приложениям. Настольные приложения запускают процессы клиента и сервера на одной и той же машине.

Двухъярусная структура, как описано выше, является системой с **толстым клиентом**. В более сложных случаях PCMEF-уровни — за исключением уровня `presentation` — могут быть перемещены на компьютеры сервера. Это приведет к структуре с **тонким клиентом**. Структура с тонким клиентом может быть двухъярусной или многоярусной. Например, сервер БД может быть размещен на двух узлах, чтобы отделить бизнес-БД и правила целостности (запрограммированные в виде хранимых процедур и триггеров) от более прагматичных задач БД, связанных с хранением и управлением данными. Это приводит к **трехъярусной структуре**, где PCMEF-объекты ПО все еще запускаются на клиентском уровне или, возможно, некоторые объекты `domain` (предметная область) и `foundation` (основание) перемещаются в средний уровень.

Возможны также **многоярусные структуры**. В распределенной системе некоторые PCMEF-уровни могут быть отображены EJB- или .NET-компонентами и развернуты отдельно. В приложениях, которые должны отображать и обрабатывать большие объемы данных своих БД, уровень `domain` может быть развернут на сервере объекта. В этом случае Object Storage API (библиотека размещения объектов) обеспечивает соответствие между сервером сохраняемых объектов, оставшимися прикладными уровнями PCMEF и общим сервером реляционной БД.

9.3. Структурные паттерны

Обсуждение структурного проектирования не полно без идентификации, именования и объяснения компромиссов паттернов проекта, поддерживающих структурные шаблоны типа PCMEF. **Паттерн проекта** означает и объясняет лучшие и широко подтвержденные теорией и практикой решения задач проектирования. Со времен появления работы Гамма и др. [32] изучение паттернов проекта процветало и переросло в важную отрасль знаний из области разработки ПО. Книга Гамма и др. [32] была написана четырьмя видными авторами. По этой причине эти паттерны обычно называются паттернами **Банды четырех** (Gang of Four — GoF).

Паттерны увеличивают информированность соответствующих приложений общих технологий об управлении зависимостями, о функциональных возможностях многократного использования, о средствах защиты информации, о работе с абстракциями и т. д. Изучение паттернов делает возможным избежать «изобретения велосипеда». Предыдущая часть этой главы использовала много разных паттернов без их определения и описания. Этот раздел повторно рассматривает предыдущее обсуждение и объясняет, какие паттерны использовались. Все паттерны, рассмотренные ниже, — паттерны GoF.

9.3.1. Фасад

Назначение **паттерна Фасад** (Façade) описано у Гамма и др. [32] как «интерфейс более высокого уровня, который делает подсистему более легкой в использовании». Его цель — «минимизировать связь и зависимости между подсистемами».

«Интерфейс более высокого уровня» — это не обязательно концепция интерфейса, как представлено в разделе 9.1.7. Это может быть абстрактный класс или конкретный доминирующий класс, также объясненные в разделе 9.1.7. Дело в том, что интерфейс с более высоким уровнем инкапсулирует главные функциональные возможности подсистемы (пакета) и обеспечивает основные или даже единственную точку входа для клиентов пакета.

Объекты-клиенты связываются с пакетом через объект Фасад. Объект Фасад делегирует свою работу другим объектам пакетов, если это необходимо. Последствием этого является сокращение путей связи между пакетами и уменьшение числа объектов, с которыми клиенты пакета имеют дело. Фактически пакет оказывается скрытым за объектом Фасад.

Рис. 9.28 демонстрирует класс `EEntity` (класс «сущность» пакета `entity`) в качестве Фасада к пакету `entity` (сущность). Клиенты пакета, такие как `CActioner` (класс «исполнитель» пакета `control`) и `MBroker` (класс «посредник» пакета `mediator`), посылают запросы к `EEntity`, который передает их далее к соответствующим объектам в пределах пакета. `EEntity` — конкретный доминирующий класс. Он мог бы быть *внешним классом*, а три оставшихся класса могли бы быть **внутренними классами**. (В Java внутренние классы содержат свои определения внутри определения внешнего класса.) Делая `EEntity` интерфейсом или абстрактным классом, можно далее уменьшить зависимости клиентов от пакета `entity`.

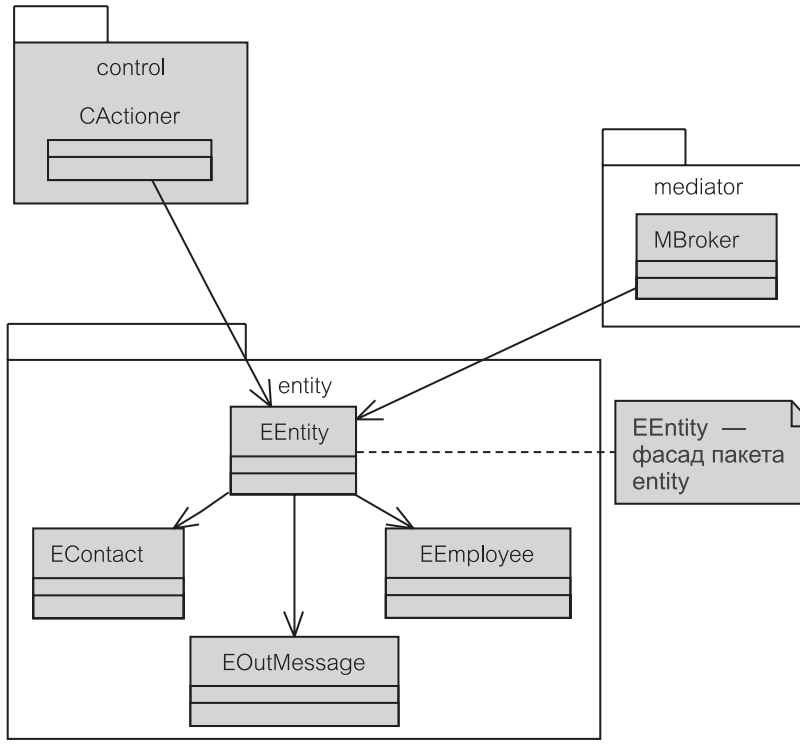


Рис. 9.28. Паттерн Фасад

9.3.2. Абстрактная фабрика

Паттерн Абстрактная фабрика (Abstract Factory) обеспечивает «интерфейс для создания семейств связанных или зависимых объектов без определения их конкретных классов» [32]. В противоположность паттерну Фасад, являющемуся «интерфейсом более высокого уровня», который мог означать конкретный класс, интерфейс в Абстрактной фабрике — настоящий интерфейс (что предпочтительно) или абстрактный класс (раздел 9.1.7).

Абстрактная фабрика позволяет приложению вести себя по-разному, обращаясь к одному из нескольких семейств объектов, скрытых за интерфейсом Абстрактная фабрика. Значение параметра конфигурации может определять, к какому семейству нужно обратиться.

Рис. 9.29 приводит пример, где приложение может работать, используя либо семейство объектов консольного отображения, либо семейство объектов GUI-окон. `IPPresentation` (интерфейс «представление» пакета `presentation`) — Абстрактная фабрика, которая передает создание конкретных объектов или классу `PConsole` (класс «консоль» пакета `presentation`) или `PWindow` (класс «окно» пакета `presentation`), являющимся конкретными фабриками (т. е., классами, которые непосредственно формируют необходимые объекты), в зависимости от того, как конфигурируется система. Объекты-клиенты (типа `PInit` — класс «инициализация» пакета `presentation`) обращаются к конкретным объектам через свой интерфейс (`IPPresentation`).

Поскольку Абстрактная фабрика — интерфейс, который реализуется во всех семействах классов, расширяя их, поддержка новых семейств может

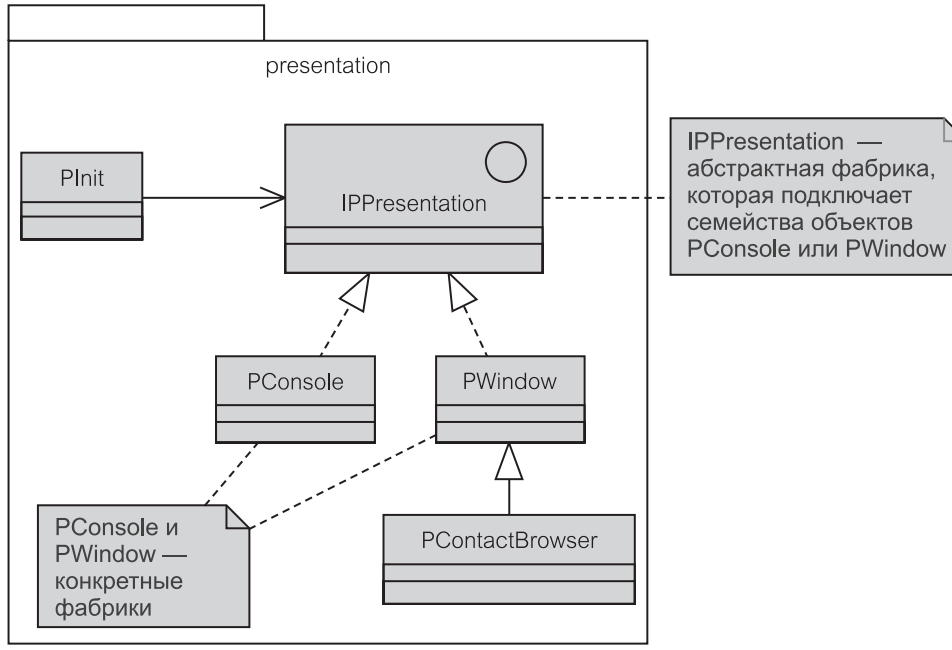


Рис. 9.29. Паттерн Абстрактная фабрика

привести к эффекту ряби у существующих конкретных классов. Гамма и др. [32] обсуждают несколько практических решений, связанных с этой проблемой.

При более близком рассмотрении паттерн Абстрактная фабрика можно считать разновидностью паттерна Фасад. Интерфейс Абстрактная фабрика может использоваться как «интерфейс более высокого уровня», через который организуется связь с пакетом, формируется множество конкретных классов и инкапсулируются классы, которые выполняют реальную работу внутри пакета.

9.3.3. Цепочка обязанностей

Цель **паттерна Цепочка обязанностей** (Chain of Responsibility) — «избежать непосредственного соединения отправителя запроса с его получателем, давая возможность более чем одному объекту обработать запрос» [32]. Цепочка обязанностей — всего лишь другое название концепции **делегирования** (раздел 9.1.6).

Если используется Цепочка обязанностей, объект-клиент, который посылает сообщение, не имеет прямой ссылки на объект, который, в конечном счете, поставляет услугу. Этот паттерн необходим, чтобы реализовать в проекте принцип соседней связи (раздел 9.2.2).

Рис. 9.30 дает пример паттерна Цепочка обязанностей. Объект `PWindow` (класс «окно» пакета `presentation`) получает запрос `displayContact()` (отобразить делового партнера) и посылает сообщение `retrieveContact()` (извлечь делового партнера) объекту `CActioner` (класс «исполнитель» пакета `control`). `CActioner` не может обработать запрос и пересылает его объекту `EContact` — класс «деловой партнер» пакета `entity` (если требуемый объект `EContact` был ранее инициализирован и существует в памяти программы) или к `MBroker` — посредник (если объект `EContact` должен быть извлечен из БД). Если объект находится в памяти, `EContact` обеспечит обслуживание, и объект будет возвращен объекту `PWindow` для отображения. В противном случае `MBroker` делегирует запрос объекту `FReader` (класс «чтение» пакета `foundation`), чтобы объект мог быть извлечен из БД и инициализирован так же, как раньше инициализировался объект `EContact` перед тем, как он был передан объекту `PWindow`.

9.3.4. Наблюдатель

Назначение **паттерна Наблюдатель** (Observer) — «определить зависимость 'один ко многим' между объектами так, чтобы в случае изменения состояния объекта все зависимые от него объекты были уведомлены и автоматически обновлены» [32]. Известный также как **паттерн Издание-подписка**, паттерн Наблюдатель используется в обслуживании асинхронной связи при обработке событий (раздел 9.1.8).

Паттерн связывает два типа объектов: **субъект** (объект, который наблюдается) и **наблюдатель**. В разделе 9.1.8 субъект назывался *объектом-издателем*, а наблюдатель назывался *объектом-подписчиком*. Субъект может иметь много наблюдателей, которые подпишутся на него. Все наблюдатели уведом-

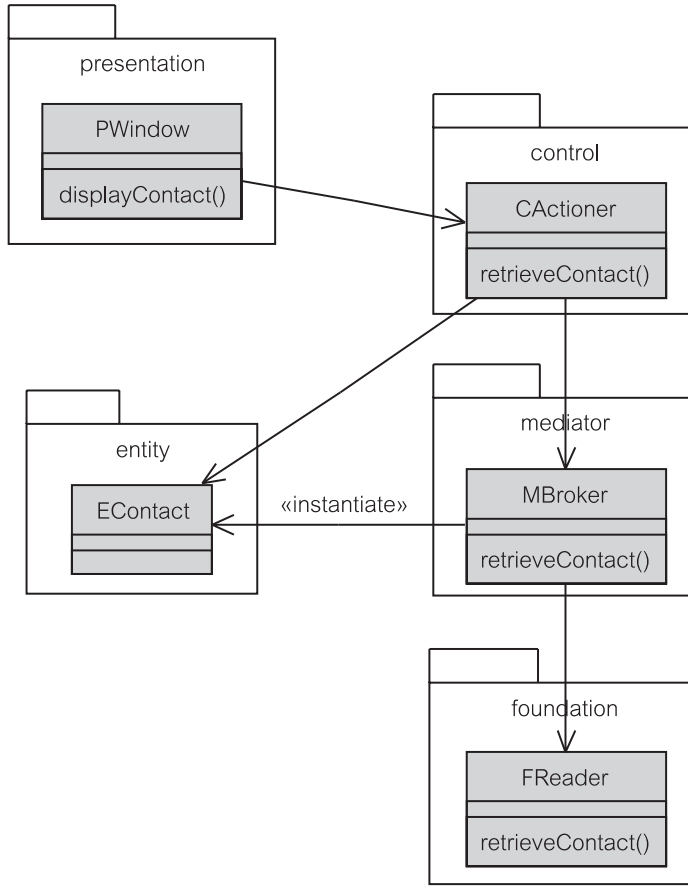


Рис. 9.30. Паттерн Цепочка обязанностей

ляются об изменениях состояния субъекта и могут затем выполнить необходимую обработку, чтобы синхронизовать свои состояния с состоянием субъекта. Наблюдатели не связаны друг с другом и могут выполнять различную обработку в ответ на уведомления субъекта об изменениях его состояния.

Хотя в определении паттерна Наблюдатель упоминаются зависимости, паттерн создает слабую связь между субъектами и наблюдателями. Уведомления передаются наблюдателям автоматически. Субъект не знает конкретные классы наблюдателей при условии, что уведомления основаны на интерфейсах, которые реализуют наблюдатели (раздел 9.1.8). Субъекты и наблюдатели работают в отдельных потоках, обеспечивая таким образом еще более низкую связь. Низкая связь паттерна Наблюдатель может (и должна) дать преимущество в уровневых структурных шаблонах для обеспечения восходящей связи между уровнями.

PCMEF-шаблон санкционирует использование паттерна Наблюдатель для обеспечения принципа восходящего уведомления (раздел 9.2.2). Цепь уве-

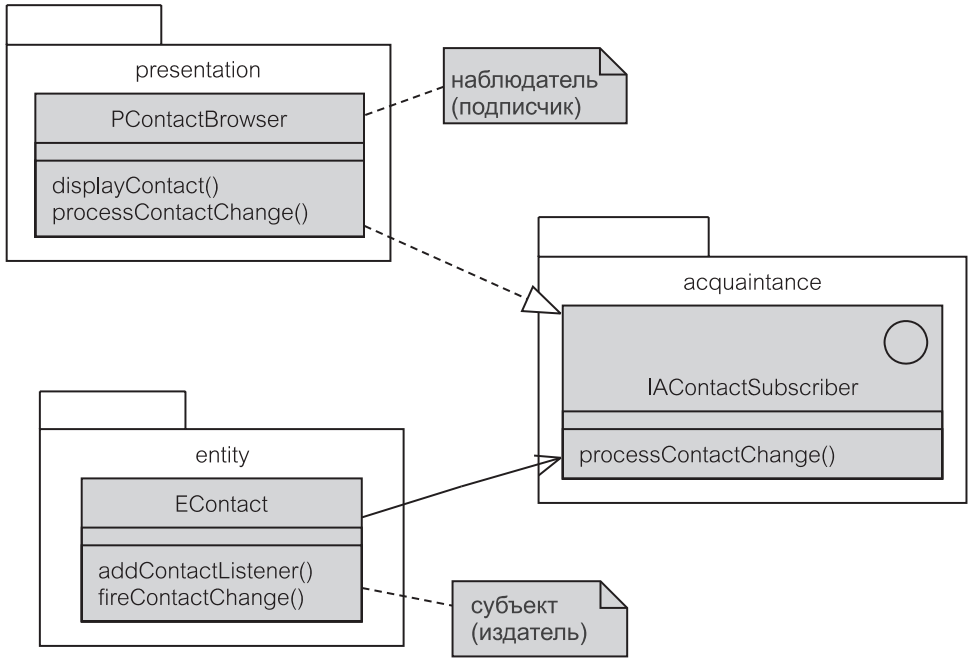


Рис. 9.31. Паттерн Наблюдатель

домлений между уровнями восходящая. Уровень `foundation` уведомляет об изменениях своего состояния пакет `mediator` (посредник), который передает их дальше пакету `entity` (сущность) и уровню `control` (управление). Уровень `control` (управление) посылает уведомления вплоть до уровня `presentation` (представление).

Рис. 9.31 показывает, как паттерн Наблюдатель используется в восходящем уведомлении от пакета `entity` до уровня `presentation`. `PContactBrowser` (класс «браузер деловых партнеров» пакета `presentation`) подписан на `EContact` (класс «деловой партнер» пакета `entity`). Поскольку два класса находятся в несоседних пакетах, для обеспечения принципа использования пакета знакомств (раздел 9.2.2) включен пакет `acquaintance` (знакомство). Пакет `acquaintance` содержит интерфейс `IAContactSubscriber` (интерфейс «подписчик на деловых партнеров» пакета `acquaintance`).

На рис. 9.31 нет никакого специального *объекта-регистратора* (раздел 9.1.8). Здесь `EContact` использует метод `addContactListener()` (добавить приемник информации о деловых партнерах) для регистрации `IAContactSubscriber` в качестве своего наблюдателя. В действительности наблюдателем является `PContactBrowser`, который реализует интерфейс `IAContactSubscriber`. Когда состояние `EContact` каким-то образом изменяется, метод `fireContactChange()` (инициализация изменения деловых партнеров) уведомляет об изменении интерфейс

PContactBrowser, вызывая метод processContactChange() (выполнить изменение деловых партнеров) у интерфейса IAContactSubscriber. Метод processContactChange() может затем запросить метод displayContact() (отобразить делового партнера), чтобы показать новое состояние EContact в окне браузера.

9.3.5. Посредник

Паттерн Посредник (Mediator) определяет объекты, которые инкапсулируют взаимную связь между другими объектами, возможно, из различных уровней. Паттерн «обеспечивает свободную связь, храня явные ссылки объектов друг на друга, что позволяет вам независимо изменять их взаимодействие» [32].

Паттерн Посредник занимает свое законное место в PCMEF-шаблоне, где уровень mediator является посредником между уровнем foundation (основание) и пакетом entity — сущность (раздел 9.2.2). Уровень control (управление) — другой пример паттерна Посредник, так как он является посредником между уровнем presentation (представление) и пакетом entity.

В PCMEF пакет mediator отделяет объекты пакета entity и объекты пакета foundation. Он действует так же, как паттерн Фасад (раздел 9.3.1) между объектами пакетов entity и foundation, сокращая, таким образом, число взаимосвязей между ними. Паттерн Фасад отличается от паттерна Посредник тем, что протокол взаимосвязи Фасада является однонаправленным, а сам он обеспечивает связь «один ко многим». В противоположность этому Посредник заменяет взаимосвязи «многие ко многим» взаимосвязями «один ко многим» к промежуточным элементам (*коллегам*), которые здесь будут посредниками.

Кроме того, пакет mediator обеспечивает синхронизацию между субъектами (foundation) и наблюдателями (entity), таким образом обеспечивая расширенную форму паттерна Наблюдатель (раздел 9.3.4). В паттерне Наблюдатель объект-посредник, который инкапсулирует сложную семантику изменений связей между субъектами и наблюдателями, известен как Менеджер изменений [32].

На рис. 9.32 приведен пример. Объект класса CActioner (класс «исполнитель» пакета control) делегирует метод deleteContact() (удалить делового партнера) объекту класса MBroker (класс «посредник» пакета mediator). Задача MBroker двойственная. Во-первых, требуется делегировать метод deleteContact() далее объекту класса FUpdater (класс «корректировщик» пакета foundation), который отвечает за удаление делового партнера из БД (БД также удалит любые исходящие сообщения к этому партнеру, все еще находящиеся в ней).

Во-вторых, если удаление в БД выполнено успешно, MBroker должен проверить, находится ли EContact (класс «деловой партнер» пакета entity) в памяти программы вместе с какими-либо EOutMessages (класс «исходящие сообщения» пакета entity), относящимися к нему. Если это так, MBroker посылает сообщение removeMessages(cnt) (удалить сообщения), чтобы удалить EOutMessages, а затем метод cnt.delete() (удалить), чтобы удалить EContact.

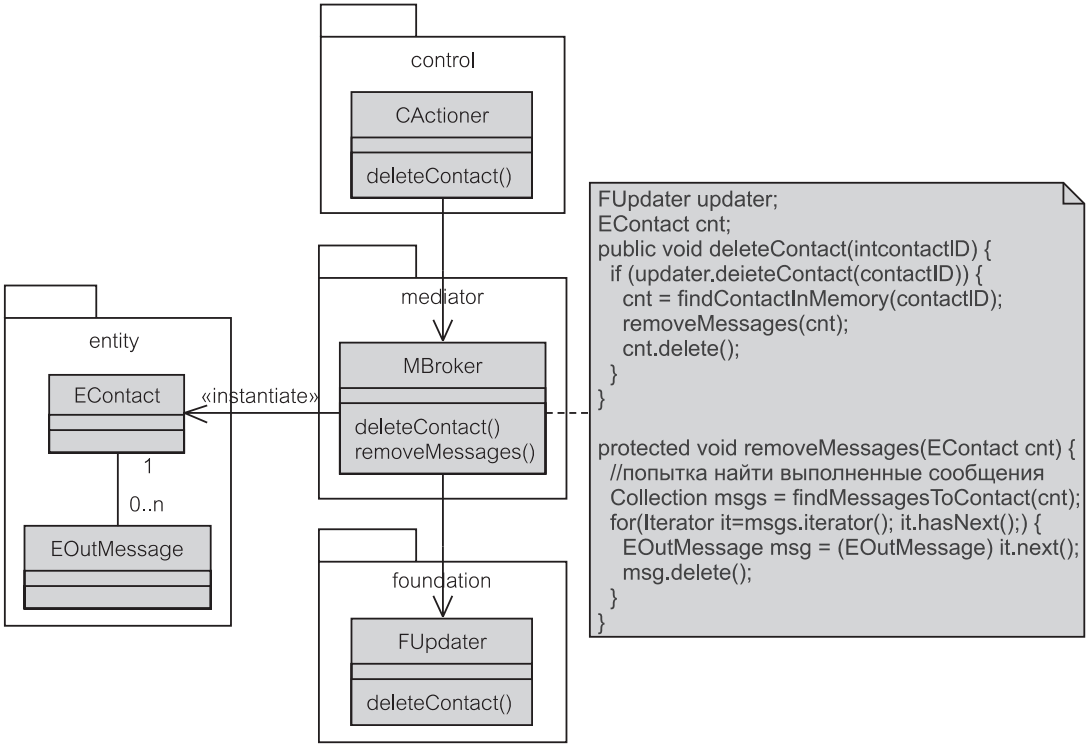


Рис. 9.32. Паттерн Посредник

Резюме

1. *Структура ПО* определяется как организация элементов ПО в систему, предназначенную для достижения некоторой цели.
2. *Структурный проект* — множество решений, целью которых является работоспособная и эффективная структура ПО вместе с обоснованием этих решений. Нормальный структурный проект использует иерархическое *выделение уровней* объектов ПО (*классы проекта*) и гарантирует, что *зависимости* между объектами минимизированы и видимы в структурах программы во время компиляции.
3. Структурные уровни построены из *пакетов*, которые содержат классы проекта. Пакеты могут быть вложены. Соответственно, необходимо рассматривать три уровня структурных зависимостей: *зависимости уровней*, *зависимости пакетов* и *зависимости классов*.
4. С точки зрения функционирования *зависимости методов* являются главной категорией зависимостей в программе. Зависимости методов возникают из-за передачи сообщений между объектами. *Объекты-клиенты* зависят от *объектов-поставщиков*. При использовании делегирования объект-поставщик может быть *делегирующим объектом*, который просто делегирует услугу ее реальному поставщику.

5. *Интерфейс* это объявление ряда особенностей (атрибутов и операций), которые непосредственно не конкретизируются. *Интерфейс Java* позволяет использовать только атрибуты, которые являются константами. *UML-интерфейс* допускает любые атрибуты, включая атрибуты, которые формируют ассоциации к классам и ассоциации между интерфейсами.
6. Интерфейс отличается от *абстрактного класса* или *чисто абстрактного класса*. На языках с единственным наследованием реализации класс может быть подклассом более одного класса (и один из этих классов может быть абстрактным или чисто абстрактным). Однако класс может реализовать несколько интерфейсов (а один и тот же интерфейс может быть реализован несколькими классами).
7. Использование интерфейсов приводит к двум видам зависимостей: *зависимость реализации* (когда класс реализует интерфейс) и *зависимость использования* (когда класс использует интерфейс). Интересно, что в отличие от других зависимостей зависимости реализации и использования могут быть весьма выгодны в структурах ПО. В частности, они могут использоваться, чтобы нарушить циклические зависимости между классами, пакетами и уровнями.
8. *Зависимости событий* — вторая главная категория зависимостей в программе (кроме зависимостей методов). Чтобы быть точными, зависимости событий — своего рода зависимости методов, но такие, что передача сообщений является асинхронной. В обработке событий зависимость возникает между *объектом-издателем* и *объектом-поставщиком*. В отличие от синхронных зависимостей методов зависимости событий более слабые, более устойчивые к программным изменениям и ими легче управлять.
9. Объединение обработки событий и интерфейсов создает наиболее мощный механизм облегчения *управления зависимостями* в структурах ПО.
10. *Знакомство* определяет ситуацию, когда объект передает другой объект в аргументе своего метода. Передаваемый объект может быть (и так часто бывает) интерфейсом. *Зависимости знакомства* являются зависимостями методов, возникшими динамически во время выполнения. Знакомство — очень полезная практика программирования, но ее невидимость в структурах программ времени компиляции является проблемой. Одним из способов разрешения проблемы является размещение интерфейсов, используемых для знакомства, в отдельном *пакете знакомств*.
11. *Структурный шаблон* — скелет решения для разработки ПО, который стимулирует разработчиков широко использовать структурное проектирование. Один из наиболее известных структурных шаблонов — MVC.
12. Эта книга использует структурный шаблон по имени PCMEF. PCMEF определяет ряд строгих принципов проектирования ПО. Цель состоит в том, чтобы минимизировать зависимости между объектами ПО и облегчить понятность, удобство сопровождения и масштабируемость в окончательной системе. Шаблон называется PCMEF+, когда должно быть подчеркнуто наличие пакета знакомств.
13. *Паттерн проекта* называет и объясняет лучшие и широко подтвержденные практику и знания, используемые для решения задач проектирования.

Когда паттерн служит целям структурного проектирования, он называется *структурным паттерном*.

14. Паттерны, которые наиболее известны в PCMEF-шаблоне: Фасад, Абстрактная фабрика, Цепочка обязанностей, Наблюдатель и Посредник. Все они паттерны GoF (Банды четырех).

Ключевые термины

Обзорные вопросы

1. Зависимости между объектами ПО могут быть проанализированы с различных точек зрения и на различных уровнях абстракции. Которая из этих точек зрения или уровней абстракции является наиболее уместной для понятия *брандмауэра зависимости*?
2. Книга использует термин *класс проекта*, что означает в зависимости от контекста любой из следующих связанных с ним терминов: класс ПО, класс системы, прикладной класс, класс реализации и класс программы. Объясните различия в акценте этих связанных терминов.
3. Имеются два главных метода устранения циклических зависимостей между пакетами. Что это за методы?
4. Если бы вы должны были расклассифицировать различные зависимости в соответствии с их воздействием на структурное проектирование, то какова была бы эта классификация? Под воздействием мы подразумеваем степень, в которой эти зависимости должны быть минимизированы. Предположите, что воздействие может быть неприятным, нейтральным или полезным.
5. Что было бы хорошим альтернативным названием для PCMEF-пакетов?
6. Который из паттернов GoF, обсужденных в этой главе, лучшим образом сочетается с принципом восходящего уведомления? Объясните.
7. Который из паттернов GoF, обсужденных в этой главе, может использоваться, чтобы реализовать доминирующий класс? Объясните.

Примеры задач

Упражнения учебного примера

1. Рассмотрим подпоток *S1 — просмотр непосланных сообщений* из раздела 8.2.3. Подумайте насчет классов проекта, которые могут потребоваться, чтобы реализовать этот подпоток. Изобразите диаграмму классов с PCMEF-пакетами и назначьте классы проекта в этих пакетах. Вам могут понадобиться интерфейсы, чтобы гарантировать PCMEF-принципы.

   Подумайте относительно последовательности операций, необходимых для организации подпотока. Добавьте в диаграмму эти операции к классам.

2. Рассмотрим подпоток *S1 — просмотр непосланных сообщений* из раздела 8.2.3. Учтем также упражнение 1, приведенное выше. Изобразите альтернативную диаграмму классов для этого подпотока. Используйте PCMEF-соглашение об именах, чтобы назвать классы. Не показывайте пакеты. Вам могут понадобиться интерфейсы, чтобы гарантировать PCMEF-принципы.

   Альтернативная диаграмма должна использовать сценарий, где просмотр непосланных сообщений получается путем запрашивания объекта `employee` (служащий) относительно исходящих сообщений, размещенных в нем. Если эти исходящие сообщения не в кэше памяти, программа должна сделать «экскурс» в БД.

   Подумайте относительно последовательности операций, необходимых для выполнения этого сценария. Добавьте в диаграмму эти операции к классам.

3. Рассмотрим подпоток *S3 — передача сообщений электронной почты* из раздела 8.2.3. Подумайте насчет классов проекта, которые могут потребоваться, чтобы реализовать этот подпоток. Используйте PCMEF-соглашения об именах, чтобы назвать классы. Не показывайте пакеты. Вам могут понадобиться интерфейсы, чтобы гарантировать PCMEF-принципы.

   Рассмотрите сценарий, где для передачи по электронной почте исходящее сообщение должно быть извлечено из БД, если только оно не находится в кэше памяти. В пакете управления используйте отдельные классы для извлечения исходящего сообщения и для отправки его.

   Подумайте относительно последовательности операций, необходимых для выполнения этого сценария. Добавьте в диаграмму эти операции к классам.

Небольшой проект — управление информацией о партнерах

Один из сценариев использования, связанных с учебным примером EM — `Manage Contact Information` (управление информацией о деловых партнерах) — рисунки 7.4 и 8.1. Подобно EM, `Manage Contact Information` является подмножеством CM (рис. 7.1). В свою очередь, CM является подмножеством AEM (напомним, что EM — Email Management — управление электронной почтой, CM — Contact Management — управление деловыми партнерами, AEM — Advertising Expenditure Measurement — оценка расходов на рекламу).

CIM (Contact Information Management — управление информацией о партнерах) связано с поддержанием текущей информации относительно деловых партнеров. Согласно определению в глоссарии предметной области (таблица 7.1), деловой партнер — это человек или организация, которые взаимодействуют с AEM или с которыми делают бизнес. Практически, все деловые партнеры являются людьми. Однако в ряде случаев лицо, которое связывается с AEM, анонимно для AEM (или персональные характеристики для AEM не интересны). К тому же в некоторых случаях деловой партнер не представляет никакую организацию, а является частным лицом (и, тем не менее, AEM хочет хранить информацию относительно этого человека). По этим причинам CIM делает различие между `PersonContact` (партнер-личность) и `OrganizationContact` (партнер-организация) как двумя видами `Contact` (деловой партнер).

Функциональные возможности CIM состоят из четырех стандартных операций, которые могут быть выполнены на бизнес-объекте — Create (создать), Read (читать), Update (корректировать) и Delete (удалить). Это называется CRUD-функциональными возможностями (по первым буквам операций). Рис. 9.33 изображает диаграмму сценариев использования для CIM. Четыре основных сценария использования смоделированы как абстрактные из-за различий в обработке `PersonContacts` и `OrganizationContacts`.

`Read Contact` (читать делового партнера) на рис. 9.33 расширен двумя другими сценариями использования, потому что данные о деловом партнере необходимо извлечь из БД прежде, чем прикладная программа сможет корректировать или удалить его. `Update OrganizationContact` (корректировать партнера-организацию) расширен сценарием использования `Create PersonContact` (создать партнера-личность), чтобы отразить тот факт, что `PersonContact` может быть создан в процессе корректировки `OrganizationContact`. После создания `PersonContact` он может быть присвоен сценарию использования `OrganizationContact`, расширяя `Update PersonContact` (корректировать партнера-личность) — см. Assign PersonContact to OrganizationContact (назначить PersonContact сценарию использования OrganizationContact) на рис. 9.33.

`Create PersonContact` расширен сценарием использования `Assign PersonContact to OrganizationContact` (назначить Assign PersonContact сценарию использования OrganizationContact). Это означает, что `PersonContact` может быть (но не обязательно) добавлен к `OrganizationContact`, когда тот создан.

Абстрактный сценарий использования `Read Contact` извлекает информацию о деловом партнере из БД и представляет список деловых партнеров в окне браузера, как показано на рис. 9.34. Окно предназначено для конкретного сценария использования `Create OrganizationContact` (создать партнера-организацию). Возвращаясь к глоссарию предметной области (таблица 7.1), можно вспомнить, что имеются шесть предопределенных типов организаций: рекламодатели (Advertisers), группы рекламодателей (Advertiser groups), агентства (Agencies), группы агентств (Agency groups), рынки сбыта (Outlets), и поставщики (Providers). Каждая организация может быть одного, нескольких или ни одного из этих типов.

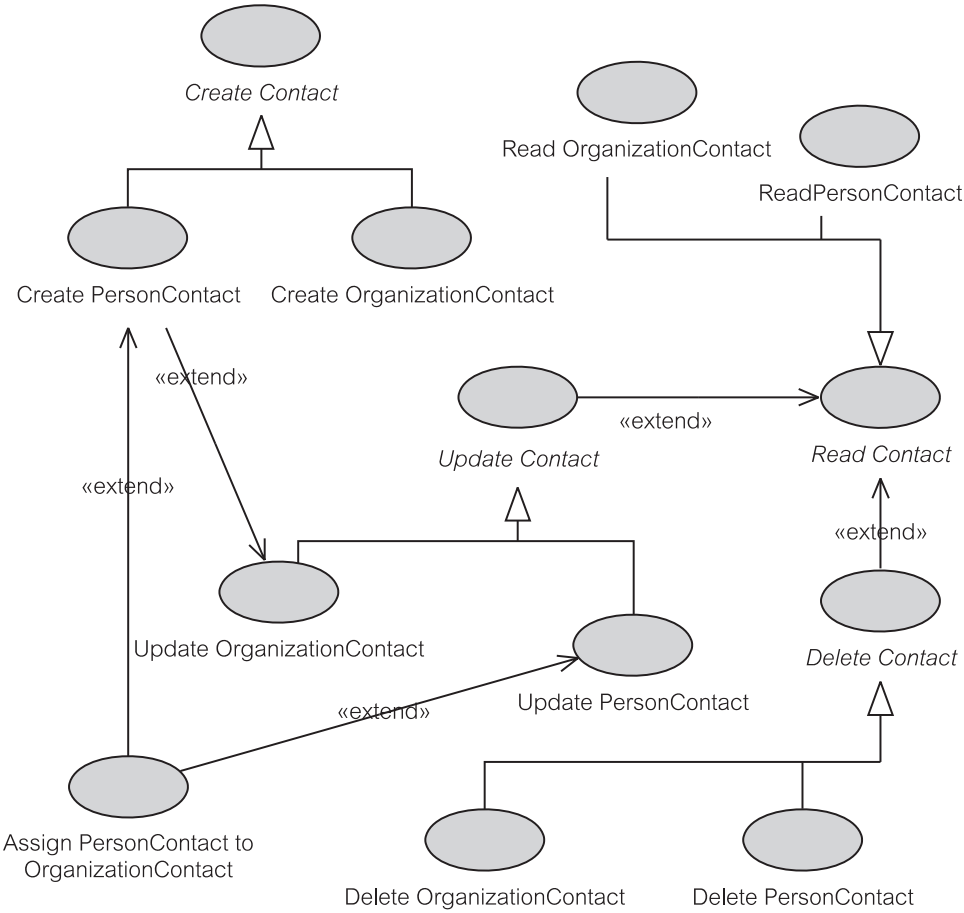
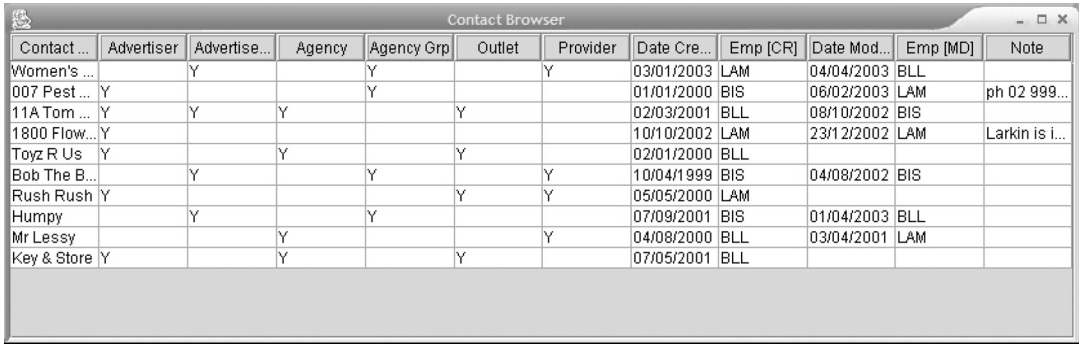


Рис. 9.33. Диаграмма сценариев использования для CIM



Contact ...	Advertiser	Advertise...	Agency	Agency Grp	Outlet	Provider	Date Cre...	Emp [CR]	Date Mod...	Emp [MD]	Note
Women's ...		Y		Y		Y	03/01/2003	LAM	04/04/2003	BLL	
007 Pest ...	Y			Y			01/01/2000	BIS	06/02/2003	LAM	ph 02 999...
11A Tom ...	Y	Y	Y		Y		02/03/2001	BLL	08/10/2002	BIS	
1800 Flow...	Y						10/10/2002	LAM	23/12/2002	LAM	Larkin is i...
Toyz R Us	Y		Y		Y		02/01/2000	BLL			
Bob The B...		Y		Y		Y	10/04/1999	BIS	04/08/2002	BIS	
Rush Rush	Y				Y	Y	05/05/2000	LAM			
Humpy		Y		Y			07/09/2001	BIS	01/04/2003	BLL	
Mr Lessy			Y			Y	04/08/2000	BLL	03/04/2001	LAM	
Key & Store	Y		Y		Y		07/05/2001	BLL			

Рис. 9.34. Окно браузера для OrganizationContact

Столбцы Date Created (дата создания), Emp [CR] (создавший служащий), Date Modified (дата изменения) и Emp [MD] (изменивший служащий) содержат даты и инициалы служащих, которые создавали и последний раз изменяли информацию о деловом партнере. Для каждого OrganizationContact CIM может зафиксировать несколько, одного или ни одного PersonContact. Список PersonContact может быть извлечен и показан в отдельном окне браузера (здесь не изображенном).

Сценарий использования Read Contact показывает только основную информацию относительно деловых партнеров. Полная информацию о деловых партнерах включает другие характеристики, типа адреса, телефона и т. д. Эта информация предназначена для других сценариев использования, как будет рассмотрено ниже.

Сценарий использования Create Contact (создать делового партнера) предоставляет актору «окна ввода» (диалоговые окна) для задания характеристик OrganizationContact или PersonContact. Функциональные возможности этого сценария использования можно видеть на примерах окон ввода, представленных рисунками 9.28 и 9.29. Обратите внимание, что эти окна — только прототипы и могут быть изменены в процессе детального проектирования. Например, для сценария использования Create Contact следует учесть четыре основные функциональные кнопки:

- OK (то есть, сохранить введенные характеристики в БД, закрыть окно и вернуться к окну браузера);
- Cancel — отменить (то есть, отменить всю введенную информацию и вернуться к окну браузера);
- Clear — очистить (то есть, очистить всю введенную информацию, не закрывать окно и позволить повторный ввод;
- Save — сохранить (то есть, сохранить введенные характеристики в БД, не закрывать окно и позволить переход к следующему деловому партнеру).

Рис. 9.35 отображает прототип окна, имеющего три страницы с закладками для Create OrganizationContact. Это средство позволяет вводить характеристики (Details) организации, включая почтовый адрес (Postal Address) и адрес для курьера (Courier Address), используя отдельные закладки. Окно не обеспечивает ввод PersonContact, связанных с OrganizationContact. Назначение PersonContact сценарию использования OrganizationContact может быть выполнено через сценарии использования Create PersonContact или Update OrganizationContact.

Рис. 9.36 изображает прототип окна для Create PersonContact. Сценарий использования позволяет назначить PersonContact сценарию использования OrganizationContact. Естественно, что в этом случае OrganizationContact должен существовать. Функциональные кнопки в прототипе окна отсутствуют.

Рис. 9.37 представляет прототип окна для сценария использования Update OrganizationContact. Как смоделировано в диаграмме сценария использования (рис. 9.33), этот сценарий использования может быть расширен двумя другими сценариями использования: Create PersonContact и Assign PersonContact to OrganizationContact. По этой причине окно

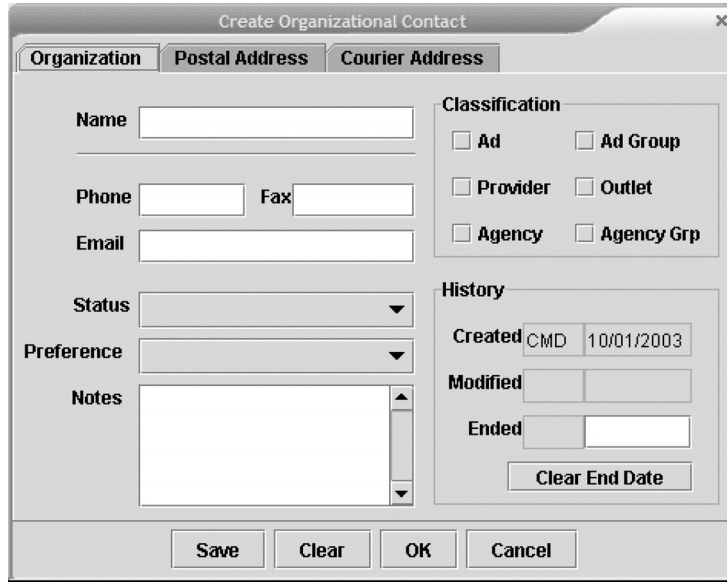


Рис. 9.35. Окно ввода для OrganizationContact

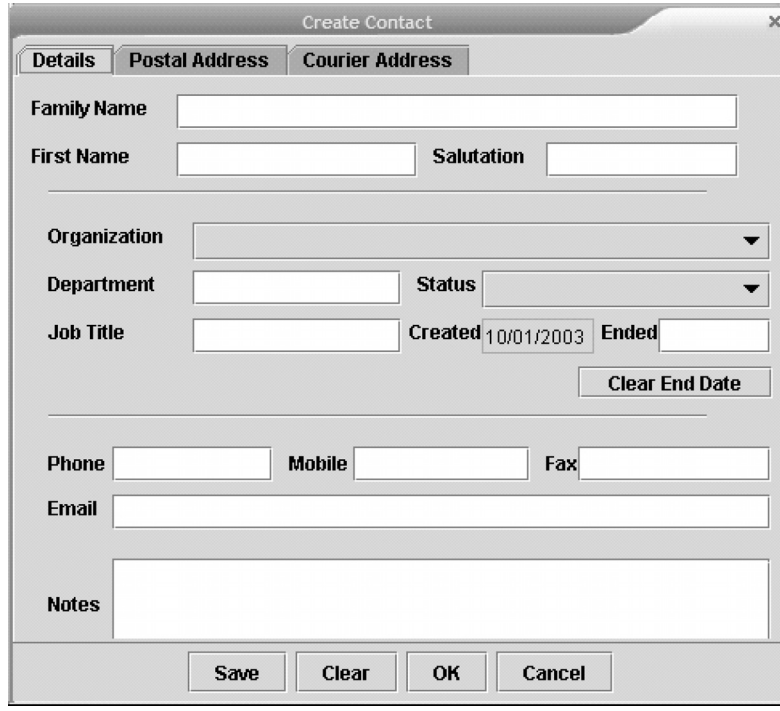


Рис. 9.36. Окно ввода для PersonContact

должно иметь еще одну страницу с закладкой по имени `Contacts` (деловые партнеры).

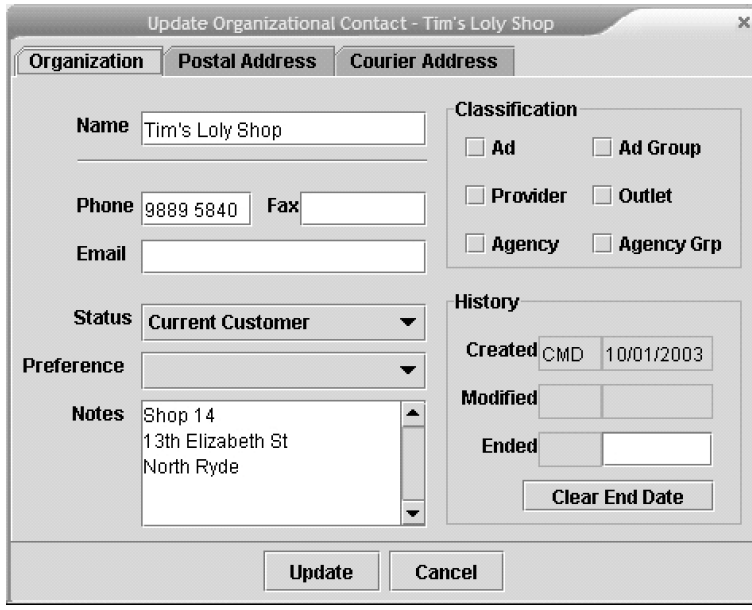


Рис. 9.37. Окно изменений для OrganizationContact

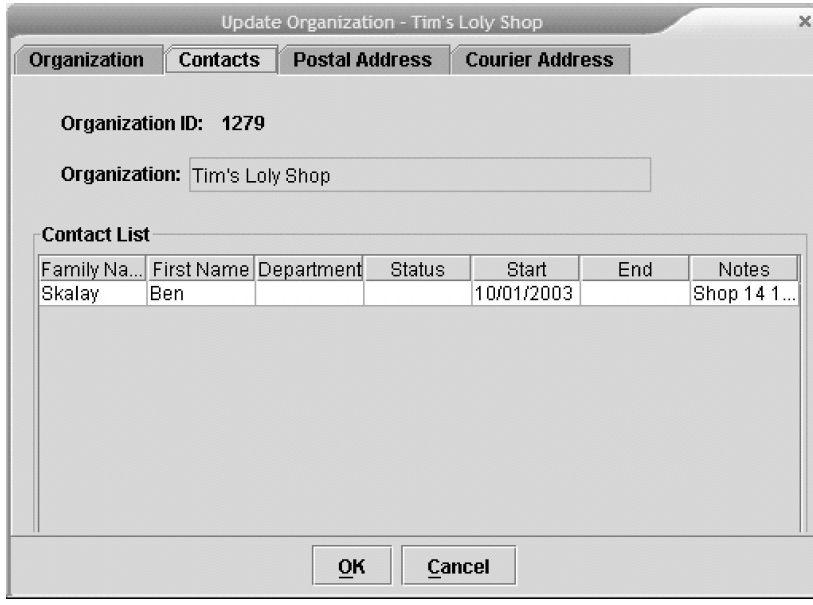


Рис. 9.38. Страница с закладкой для назначения PersonContact сценарию использования OrganizationContact

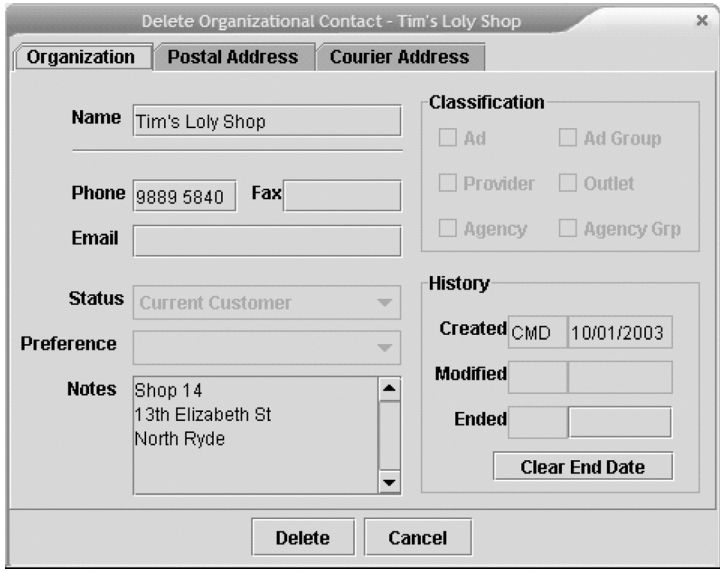


Рис. 9.39. Окно удаления для Delete OrganizationContact

Рис. 9.38 показывает окно корректировки для `OrganizationContact`, в котором открыта страница с закладкой `Contacts`. Эта страница с закладкой позволяет назначить существующий `PersonContact` сценарию использования `ContactList` (активизируя сценарий использования `Assign PersonContact to OrganizationContact`) или создать новый `PersonContact` (рис. 9.36) перед выполнением этой операции. Создание нового `PersonContact` осуществляется с помощью сценария использования `Create PersonContact`.

Сценарий использования `Update PersonContact` представляет то же самое окно, что и `Create PersonContact` (рис. 9.36). Различие лишь в том, что поля в окне содержат величины для `Update PersonContact`. Имеются также различия в отношении бизнес-правил и допустимых действий (например, количество функциональных кнопок может быть различно).

Сценарий использования `Delete OrganizationContact` (удалить партнера-организацию) позволяет удалить `OrganizationContact` после того, как будет отображена соответствующая информация (рис. 9.39). В этом окне нет никакой страницы с закладкой `Contacts`. Это означает, что CIM не позволяет удалить `OrganizationContact` и все связанные с ним `PersonContacts` в одном действии. Кроме того, `OrganizationContact` не может быть удален, если он имеет связанные с ним `PersonContacts`.

Удаление `PersonContacts` выполняется сценарием использования `Delete PersonContact`. Прототип окна для него здесь не показан, но поля окна те же самые, что и у `Create PersonContact` (рис. 9.36).

Упражнения

1. На основе спецификаций небольшого проекта разработайте концептуальную диаграмму классов. Используйте информационное содержание прототипов окон, чтобы выявить классы и их атрибуты. Определите множественность и участие ассоциаций. Объясните модель.
2. На основе спецификаций небольшого проекта разработайте диаграмму классов проекта, соответствующую шаблону PCMEF+. Имена классов следует задать согласно соглашению PCMEF+. Покажите только наиболее важные операции. Объясните модель.
3. Обратитесь к вашему решению упражнения 2. Обсудите использование в вашей модели пяти паттернов GoF, рассмотренных в главе 9. Если некоторые паттерны не использовались, измените вашу диаграмму классов так, чтобы воспользоваться преимуществом всех пяти паттернов. Объясните модель.

Глава 10

Проектирование и программирование базы данных

Фактически все информационные системы обрабатывают данные, размещенные в **базе данных** (БД). Прикладные программы обращаются к БД для получения данных, переноса данных в память программы для их обработки и записи любых изменений данных назад в БД. Как правило, одновременно к БД обращается много различных программ. Каждая программа может иметь несколько экземпляров, запускаемых с различных рабочих станций *клиентов* и обращающихся к одной и той же БД. БД выполняет функции *сервера*, предоставляющего услуги многим программам одновременно.

ПО, ответственное за хранение и манипуляцию данными в БД, называется **системой управления базой данных** (СУБД). СУБД обеспечивает абстракцию, которая предоставляет данные сервера программам клиента в более понятных терминах чем биты и байты. Абстракция называется **моделью БД** (или **моделью данных**). Преобладающей моделью БД для деловых информационных систем является **реляционная модель**. **Язык структурированных запросов** (Structured Query Language — SQL) — реляционный язык, который прикладные программы должны использовать, чтобы получить доступ к БД.

БД характеризуется следующими свойствами:

- *Большая* — ее размер часто измеряется в гигабайтах или терабайтах. Прикладная программа может использовать только маленькую часть данных БД, размещаемую в памяти оперативного запоминающего устройства (ОЗУ) компьютера.
- *Сохраняемая* — БД размещается на энергонезависимом (сохраняемом) вторичном хранилище типа магнитных и оптических дисков. Данные остаются на дисках постоянно, независимо от того, включены ли и работают ли дисководы.
- *Многопользовательский доступ* — одновременно могут иметь доступ к БД много пользователей и много прикладных программ. Большая часть ПО СУБД, называемая **администратором транзакций**, обеспечивает этот параллелизм.
- *Восстанавливаемая* — кроме управления параллелизмом, администратор транзакций отвечает за восстановление БД после отказов аппаратного и программного обеспечения. После некоторых отказов аппаратных средств восстановление может потребовать использования *резервных копий* БД (сохраняемые копии БД, создаваемые через определенный интервал времени).

- *Последовательная* — СУБД обеспечивает целостность (устойчивость) своей БД, гарантируя, что никакие пользователь или прикладная программа не смогут нарушить бизнес-правила, определенные для работы с данными. Бизнес-правила реализуются программными средствами, известными как *ограничения ссылочной целостности* и *триггеры*.
- *Безопасная* — СУБД гарантирует безопасность своей БД, означающую, что только надлежащим образом удостоверенные и уполномоченные пользователи и прикладные программы будут иметь доступ к данным и выполнять внутренние программы БД (*хранимые процедуры*).
- *Расширяемая* — БД обеспечивает достаточную степень логической и физической независимости данных. Независимость данных означает, что изменения (расширения) по отношению к проекту БД не нарушают работу существующих прикладных программ (существующим программам не интересны новые изменения, и они должны работать, как и прежде). Расширяемость касается логических изменений структуры БД (таких как добавление нового столбца в таблице) и физических изменений в хранении БД (таких как изменения индексов).

Являясь центральным хранилищем информации предприятия и ключевых стратегических ресурсов, БД должна быть разработана очень тщательно. Главная трудность возникает из-за требований многопользовательского доступа к данным. Если много пользователей и программ должны иметь доступ к БД, проект БД должен быть квалифицированным компромиссом между индивидуальными требованиями и ограниченными возможностями. Это означает, что проектирование БД обычно является центральной операцией, независимой от разработки конкретных приложений и систем (нечто, о чем нельзя полностью рассказать в книге, которая рассматривает единственный учебный пример, посвященный одной прикладной программе).

10.1. Быстрое обучение реляционным базам данных с точки зрения разработки программного обеспечения

Модель реляционных БД доминирует на рынке корпоративных БД. Этой модели теперь уже более тридцати лет, и некоторые реляционные СУБД (особенно, Oracle) использовались, чтобы хранить данные предприятия, приблизительно двадцать лет. Коммерческие интересы и инерция бизнеса приведут к тому, что мы будем видеть реляционные БД как доминирующую силу на много лет вперед.

Реляционная модель представляет данные как **записи** (то есть, **строки**) в **таблицах** (называемых также **отношениями**). Записи в различных таблицах (или в одной и той же таблице) не могут быть связаны видимыми пользователем **навигационными связями**. Для этой цели используется концепция *ссылочной целостности*, основанной на понятии *внешнего ключа* (раздел 10.1.2). SQL-операторы преобразуются в операции реляционной алгебры. Три из этих операций — выборка (`select`), проекция (`project`) и соединение

(join) — являются обязательными в модели [23]. Никакие навигационные связи или другие физические пути доступа не должны требоваться для этих операций, чтобы получить результаты из БД.

Ясно, что реляционная модель налагает различные ограничения на разработку системы и ПО. Быстрое обучение, представленное ниже, предназначено для объяснения наиболее существенных из этих ограничений. Оно использует простую БД MovieActor («Фильм-Актер»). Эта БД позже связывается с прикладным программированием на Java.

10.1.1. Таблица

Модель реляционной БД основана на строгих формальных понятиях теории математических множеств и логики предикатов. Реляционная концепция **таблицы** рассматривается как математическое множество (точнее, математическое отношение), хотя и с несколькими неточностями, чтобы сделать концепцию практической (давайте признаем, что реальная жизнь более сложна, чем теория). Таблица состоит из фиксированного числа **столбцов** и изменяющегося числа **записей** (*строк*). Столбцы должны быть **простых типов данных**, таких как числа или строки символов. Листинг 10.1 показывает таблицу по имени movie (кинофильм), которая состоит из трех столбцов и двух строк. Содержание таблицы было извлечено из БД SQL-оператором select (выбор).

Листинг 10.1. Оператор select для получения содержимого таблицы

```
Оператор select для получения содержимого таблицы
SQL> select * from movie;
M0VIE_CODE  MOVIE_TITLE                  DIRECTOR
----------  ---------------------------  ----------------------
        10  Interview with the Vampire   Neil Jordan
        11  The Birdcage                 Mike Nichols
```

Звездочка (*) в операторе select запрашивает получение всех столбцов, определенных в таблице. Оператор select в листинге 10.1 представляет **диалоговый SQL**-запрос. SQL-запросы должны также использоваться внутри программ, типа Java-программ, чтобы получить доступ к БД. Это называется **программным** использованием SQL (в противоположность интерактивному использованию). Программный SQL известен как **вложенный SQL**, потому что некоторая форма SQL (это может быть JDBC или SQLJ) вложена в операторы прикладной программы.

В устойчивых программах звездочка в операторах select не должна использоваться, потому что последующие изменения в определениях таблиц (типа добавления столбца) нарушат функционирование прикладных программ. Программист должен перечислить в операторе select все требуемые столбцы, как указано ниже:

```
SQL > select movie_code, movie_title, director from movie;
```

SQL-операторы нечувствительны к регистру. Следующий оператор, хотя и не очень красивый, эквивалентен вышеприведенному оператору:

```
SQL > SELECT movie_code, MOVIE_title, DIRECTOR from movie;
```

movie			
movie_code	NUMBER(5)	<pk>	not null
movie_title	VARCHAR2(30)		null
director	VARCHAR2(20)		null

Рис. 10.1. Структура таблицы movie

Рис. 10.1 показывает проект структуры таблицы, соответствующий содержанию таблицы в листинге 10.1. Проект таблицы использует два отделения: верхнее содержит название таблицы, нижнее содержит списки столбцов, типы данных столбцов, указатели ключей, а также допускает ли столбец значения `null`.

Проект на рис. 10.1 использует *типы данных* Oracle. Столбец `movie_code` (код фильма) использует целые числа, содержащие до пяти цифр. Другие два столбца содержат строки переменной длины, содержащие до 30 и 20 символов соответственно.

Пуристы могут утверждать, что `movie_code` — не число, потому что нет смысла использовать `movie_code` для каких-либо математических вычислений. Однако делая `movie_code` числом, можно использовать средства СУБД, позволяющие автоматически формировать последовательные числа для `movie_code`, когда в таблицу помещаются строки кинофильмов. Это достоинство перевешивает любое мнение пуританина относительно данной проблемы, в особенности для больших таблиц.

Если таблица — математическое множество, то содержание таблицы не должно иметь двойных элементов, то есть одинаковых строк. Обеспечение этого гарантирует наличие уникального ключа. **Уникальный ключ** (называемый также **потенциальным** или **альтернативным ключом**) состоит из одного или нескольких столбцов, которые не могут иметь одинаковые значения в любых двух строках таблицы. Один такой произвольно выбранный уникальный ключ, как наиболее удобный в проекте таблицы, называется **первичным ключом**. На рис. 10.1 это обозначено символом `<pk>` (primary key — первичный ключ).

В большинстве практических ситуаций необходимо, чтобы каждая таблица имела первичный ключ для идентификации ее строк. Это эквивалентно использованию *идентификаторов объектов* в системах объектов, чтобы идентифицировать каждый объект класса [61]. В некоторых случаях большинство реляционных СУБД позволяют использовать таблицы без уникального идентификатора; это означает, что могут быть вставлены одинаковые строки. Одинаковые строки имеют точно те же самые значения во всех столбцах, видимых пользователю. Внутренне, однако, СУБД в этом случае все же обеспечит уникальную идентификацию с помощью специального псевдо-столбца, называемого `row_id` (идентификатор строки), или другим подобным образом.

По определению первичный ключ не позволяет использование значения `null`. Если для столбца допускается значение **`null`**, это означает, что столбец может не иметь никакой величины в некоторых строках. Могут быть две причины для использования значения `null`: значение в настоящее время неизвестно или значение недопустимо для данной строки.

Как только структура таблицы определена с помощью инструментального CASE-средства, это CASE-средство может автоматически создать SQL-оператор `create table` (создать таблицу). Для случая, изображенного на рис. 10.1, это завершится SQL-скриптом, который будет содержать оператор `create table`, как показано в листинге 10.2 (информационный комментарий в последних строках добавлен после того, как был создан скрипт). Если CASE-средство связано с БД, разработчик может запросить автоматический запуск скрипта в БД, чтобы таблица была сразу же создана. В другом случае скрипт может быть сохранен в файле и выполнен в более позднее время. Определения таблиц, созданные в этом процессе, известны как **схема БД**.

Листинг 10.2. Оператор create table

```
Create table movie

Create table movie (
   movie_code          NUMBER(5)           not null,
   movie_title         VARCHAR2(30),
   director            VARCHAR2(20),
   constraint PK_MOVIE primary key (movie_code)
)
/ -- Oracle допускает использование / вместо ;
/ -- в качестве разделителей SQL-операторов
```

10.1.2. Ссылочная целостность

Поскольку реляционная модель явно отвергает **навигационные связи** (раздел 10.1), реляционные БД представляют концептуальные отношения между записями таблиц посредством принципа ссылочной целостности. **Ссылочная целостность** использует понятие внешнего ключа, чтобы связать записи в одной таблице с записями в другой (или даже в той же самой) таблице.

Внешний ключ — это набор столбцов (а часто это только один столбец), чьи значения соответствуют значениям первичного ключа в другой (или той же самой) таблице. Это соответствие значений внешнего и первичного ключей устанавливает связи между записями.

Рис. 10.2 приводит пример таблиц, связанных ссылочной целостностью. Графическая нотация, используемая в рис. 10.2, называется **логической моделью БД**. Когда она дополнена способом хранения и другими физическими свойствами используемой СУБД, модель называется **физической моделью БД**.

Модель на рис. 10.2 связывает актеров и кинофильмы. Каждая запись в таблице `listed_as` (перечисление) связывает одного актера с одним кинофильмом и дает порядковый номер, под которым актер внесен в список (чтобы можно было сделать различие между ведущими и второстепенными актерами). Первичный ключ `listed_as` состоит из двух столбцов: `actor_code` (код актера) и `movie_code` (код фильма). Один столбец `actor_code` представляет внешний ключ `<fk1>` (fk — foreign key — внеш-

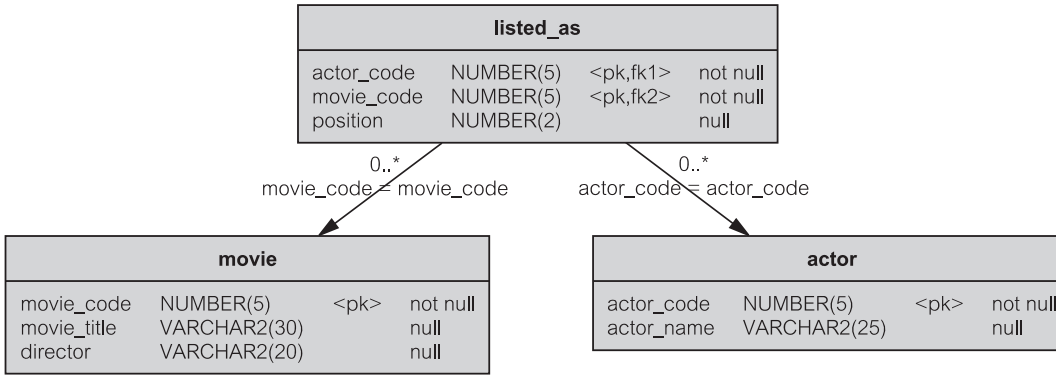


Рис. 10.2. Логическая модель БД для MovieActor

ний ключ) к записям таблицы `actor` (актер), а один столбец `movie_code` является внешним ключом `<fk2>` к записям таблицы `movie` (кинофильм). Обратите внимание, что в этом случае не могут быть значения `null` у `actor_code` и `movie_code`, хотя, вообще-то, значения `null` допустимы для внешнего ключа (означающие, что запись с таким внешним ключом не связана ни с какой записью в первичной таблице).

Стрелки показывают направление соответствия внешнего и первичного ключей. Они также представляют отношения «один ко многим» между таблицами. Один и тот же актер может играть в различных кинофильмах, и он/она будет в этом случае несколько раз перечислен(а) в списке таблицы `listed_as`. Точно так же один кинофильм может быть связан с несколькими актерами. Отношения объявляются с ограничениями эквивалентности в ссылочной целостности. Каждое значение `movie_code` в таблице `listed_as` должно иметь значение `movie_code` в таблице `movie`. Тот же самый принцип применяется и к `actor_code`.

Листинг 10.3 представляет типовое содержание БД для схемы БД на рис. 10.2. Анализ содержания может помочь в понимании принципа ссылочной целостности.

Листинг 10.3. Содержимое БД MovieActor

```
Операторы select для отображения содержимого БД MovieActor

SQL> select * from movie;
MOVIE_CODE MOVIE_TITLE                    DIRECTOR
---------- ---------------------------    ---------------
        10 Interview with the Vampire     Neil Jordan
        11 The Birdcage                   Mike Nichols
SQL> select * from actor;
ACTOR_CODE  ACTOR_NAME
---------- ------------
       100 Brad Pitt
```

```
      101 Tom Cruise
      102 Antonio Banderas
      105 Robin Williams
      106 Gene Hackman
SQL> select * from listed_as;
ACTOR_CODE  MOVIE_CODE POSITION
---------- ---------- ---------
      100          10         1
      101          10         2
      102          10         3
      105          11         1
      106          11         2
```

10.1.3. Концептуальная модель в сравнении с логической моделью БД

Моделирование БД может быть выполнено на различных уровнях абстракции. Работа с **абстракцией** означает, что приняты решения моделирования на определенном уровне детализации, представляющем эту модель. Детали, считающиеся несоответствующими уровню абстракции, в модели опускаются.

Логическая модель БД типа представленной на рис. 10.2, относится к реляционной технологии БД, которая связывает объекты данных ссылочной целостностью. Фактически же модель на рис. 10.2 относится также и к конкретной СУБД (Oracle). Модель на более высоком уровне абстракции, которая не связывается с конкретной технологией БД, называется **концептуальной моделью БД**.

Технология моделирования **«сущность-отношение»** (entity-relation — ER) — наиболее известная технология концептуального моделирования БД (раздел 2.1.2). Рис. 10.3, эквивалентный представленному ранее рис. 2.5, использует ER-нотацию, чтобы изобразить концептуальную модель, которая соответствует логической модели на рис. 10.2.

Концептуальная модель БД может также быть представлена с использованием UML-диаграммы классов. Эта способность диаграмм классов уже

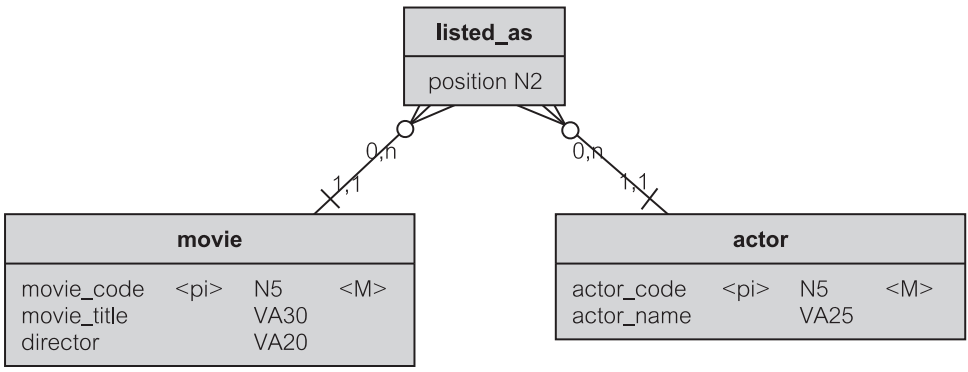


Рис. 10.3. Концептуальная ER-модель для MovieActor

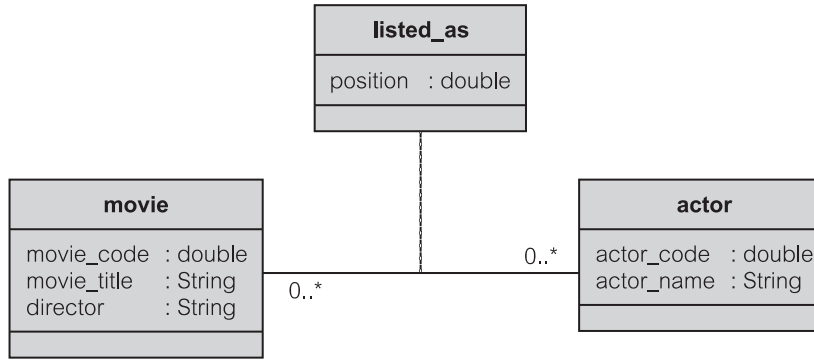


Рис. 10.4. Концептуальная UML-модель классов для MovieActor

демонстрировала свои преимущества в главе 7 (при обсуждении моделирования классов предметной области) и главе 8 (при представлении концептуальной модели классов для итерации 1 — рис. 8.8).

Диаграмма классов на рис. 10.4 соответствует ER-диаграмме на рис. 10.3. Интересный аспект этого примера заключается в том, что он приводит к UML-концепции класса ассоциации. **Класс ассоциации** — класс, который представляет отношение ассоциации между другими классами. Моделирование с классом ассоциации необходимо, когда сама ассоциация имеет атрибуты, и требуется класс, чтобы хранить эти атрибуты. Класс `listed_as` — класс ассоциации на рис. 10.4, дополненный пунктирной линией от этого класса до отношения ассоциации.

10.1.4. Реализация бизнес-правил

БД представляет собой собрание записей, связанных ссылочной целостностью. Эти отношения между записями должны быть установлены с точки зрения бизнеса. Они должны соответствовать **бизнес-правилам**. Реализация бизнес-правил в БД требует, чтобы операции модификации БД сначала проверялись на эти правила, а затем уже можно разрешить их выполнять. SQL имеет три операции модификации: `insert` (добавление), `update` (корректировка) и `delete` (удаление). Поэтому результат завершения каждой из этих операций должен быть проверен на соответствие бизнес-правилам, запрограммированным в БД.

Некоторые правила просты и применяются одинаково к разным записям в БД. Например: «Если запись о кинофильме удалена, то все связанные с ней записи таблицы `listed_as` (перечисление) должны быть также удалены». Другие правила более сложны и применяются исключительно к конкретным таблицам и к конкретным отношениям между записями. Например: «Брэд Питт — всегда ведущий актер (внесен в список под порядковым номером один) во всех фильмах Нейла Джордана, в которых он играет».

Декларативная ссылочная целостность может использоваться в реляционной БД, чтобы реализовать многие простые и повторяющиеся бизнес-пра-

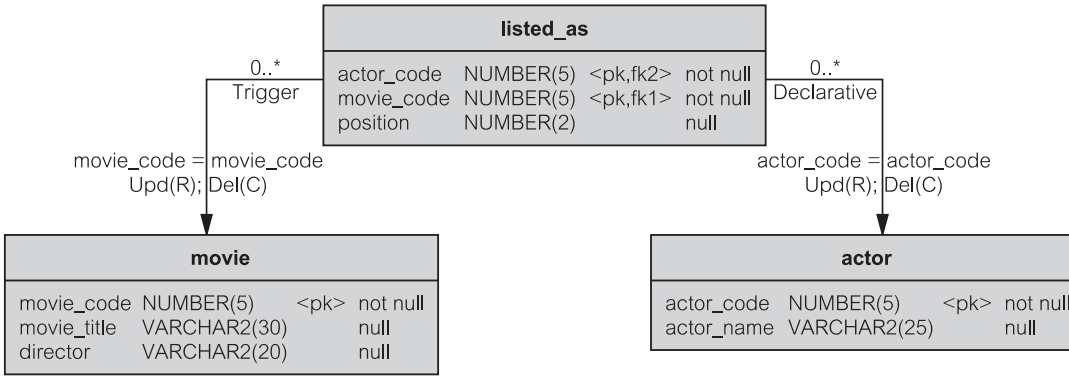


Рис. 10.5. Логическая схема БД с указателями бизнес-правил для MovieActor

вила. Декларативная ссылочная целостность реализуется как простое объявление в SQL-операторе `create table` — создать таблицу (или `alter table` — изменить таблицу), которое указывает, что делать, когда запись в этой таблице должна быть удалена, изменена или добавлена (последняя возможность недоступна во многих СУБД). Точная реализация декларативной ссылочной целостности зависит от СУБД.

Более сложные и исключительные бизнес-правила требуют реализации **процедурной ссылочной целостности**. Это включает использование языка программирования БД, уникального для каждой СУБД. Программы, написанные на таком языке и предписывающие бизнес-правила, называются **триггерами**. Отдельный триггер может быть написан для каждой из трех операций модификации каждой таблицы. В этом сценарии каждая таблица может иметь до трех триггеров. Некоторые СУБД (например, Oracle) позволяют разбить эти три триггера в соответствии с различными условиями, допуская, таким образом, больше, чем три триггера на таблицу.

Рис. 10.5 показывает, как логическая схема БД может быть графически расширена, чтобы указать декларативную или процедурную реализацию целостности. Модель показывает, что ссылочная целостность в правом отношении декларативная и что в левом отношении она является процедурной (триггером). Удаление актера или кинофильма должно каскадно удалить связанные записи с помощью подпрограммы `Del(C)` (удалить) в таблице `listed_as` (то есть, они также должны быть удалены). Корректировки ограничены подпрограммой `Upd(R)` (корректировать). Это означает, что попытка изменить `actor_code` (код актера) в таблице `actor` (актер) или `movie_code` (код фильма) в таблице `movie` (кинофильм) на другую величину будет запрещена в СУБД, потому что это, вероятнее всего, нарушит ссылочную целостность (например, изменение `movie_code` в `movie` без точно такого же обновления `movie_code` в таблице `listed_as` приведет к неправильному содержанию БД).

В большинстве СУБД поддержка операций удаления и корректировки для декларативной ссылочной целостности позволяет использовать три варианта: `cascade` (каскадный), `restrict` (ограниченный) и `nullify` (задание пус-

того указателя), иными словами — set null (установить пустой указатель). Последний вариант означает, что если первичный ключ изменен или запись удалена из первичной таблицы, то соответствующие значения внешнего ключа будут пустыми указателями. Некоторые СУБД допускают четвертый вариант — установить значения внешнего ключа вместо пустого указателя в некоторую величину default (значение по умолчанию).

Листинг 10.4 иллюстрирует декларативную ссылочную целостность. Целостность реализуется в операторе alter table (изменить таблицу). Хотя опция add constraint (добавить ограничение) и допустима в операторе create table, она приведет в этом случае к ошибке. Будет это потому, что таблица actor не существует, когда создается таблица listed_as, поэтому add constraint не может сослаться на нее.

Листинг 10.4. Декларативная ссылочная целостность с on delete cascade в MovieActor

```
Декларативная ссылочная целостность с on delete cascade
в MovieActor

SQL> CREATE TABLE listed_as (
  2    actor_code   NUMBER(5),
  3    movie_code   NUMBER(5),
  4    position     NUMBER(2),
  5    PRIMARY KEY  (actor_code, movie_code)
  6    ) ;

SQL> CREATE TABLE actor (
  2    actor_code   NUMBER(5),
  3    actor_name   VARCHAR2(25),
  4    PRIMARY KEY  (actor_code)
  5    ) ;

SQL> ALTER TABLE listed_as
  2    ADD CONSTRAINT fk_const3 FOREIGN KEY  (actor_code)
  3    REFERENCES actor
  4    ON DELETE CASCADE;
```

Листинг 10.5 иллюстрирует процедурную ссылочную целостность. Программа-триггер выполняет простое удаление всех записей таблицы listed_as с величинами movie_code, соответствующими величине movie_code в только что удаленной записи фильма. Этот триггер выполняет работу, подобную той, которую выполняет декларативная целостность, представленная на рис. 10.4, и мог бы быть заменен декларативной целостностью. CASE-средства способны автоматически формировать программы-триггеры как замену любой декларативной целостности. Для более сложных бизнес-правил триггеры должны быть запрограммированы вручную.

Листинг 10.5. Процедурная ссылочная целостность с триггером удаления в MovieActor

```
Процедурная ссылочная целостность с триггером удаления в
MovieActor

create trigger tda_movie after delete
on movie for each row
begin
   -- Удаление всех потомков в "listed_as"
   delete listed_as
   where movie_code = :old.movie_code;
end;
/
```

10.1.5. Программирование логики СУБД-приложения

Триггеры демонстрируют, что реляционные БД являются не только пассивными контейнерами данных. Реляционные БД активны — они могут быть запрограммированы, а программы размещены в БД и могут из нее выполняться. Это ставит важный вопрос разработки системы. Где следует программировать логику СУБД-приложения — в прикладной программе обслуживаемого процесса клиента или в БД обслуживающего процесса сервера?

Ответ типичен: смотря по обстоятельствам. Программирование логики СУБД-приложения в программе клиента требует использования программного варианта SQL (он называется **вложенным SQL**). Каждый SQL-оператор в программе (типа показанного в листинге 10.6) передается в БД, где он может быть проанализирован, проверен, оптимизирован, скомпилирован и выполнен. Если программа имеет много SQL-операторов и частый доступ к БД, обработка становится дорогостоящей и может воздействовать на функционирование всей системы. Однако имеется много случаев, когда SQL-операторы к БД создаются динамически во время выполнения программы и требуют ввода информации от пользователя прежде, чем они могут быть сформированы. Эти виды SQL-операторов, вероятно, придется оставить в программе клиента.

Листинг 10.6. Вложенный SQL-оператор select в прикладной программе MovieActor

```
Вложенный SQL-оператор select в прикладной программе MovieActor

-- Отобразить ведущего актера для каждого фильма
EXECUTE SELECT movie_title, actor_name
        FROM movie m, listed_as l, actor a
           WHERE m.movie_code = l.movie_code
             AND l.actor_code = a.actor_code
             AND l.position = 1;
```

В большинстве других ситуаций логика СУБД-приложения должна быть запрограммирована в сервере БД как **хранимая процедура** (то есть, программа, которая размещена в БД и может быть вызвана оттуда). Каждая СУБД имеет свой собственный вариант языка программирования для хранимых процедур и триггеров (триггеры — специальные виды хранимых процедур, которые не могут непосредственно вызываться, потому что они «инициализируются» автоматически, когда таблица изменяется).

Листинг 10.7 — пример хранимой процедуры Oracle, которая ищет фильмы с названиями, содержащими заданную строку в параметре `movie_title` (название фильма). После оператора `create procedure` (создание процедуры) имеются два запроса к этой процедуре с операторами `execute` (выполнить). В данном случае операторы выполнения запускаются из SQL-среды диалогового программирования, но обычно они будут частью прикладной программы клиента. Обратите внимание, что хранимые процедуры могут иметь входные и выходные аргументы. Обратите также внимание на использование функции `upper` (верхний регистр), объясняющей тот факт, что значения столбцов в БД являются чувствительными к регистру (но сами SQL-операторы нечувствительны к регистру, как упомянуто в разделе 10.1.1).

Хранимые процедуры могут использовать не только входные параметры, но также выходные и одновременно входные и выходные параметры. Хотя выходные параметры возвращают значения вызывающей программе, может быть более удобным использовать хранимую функцию вместо хранимой процедуры. **Хранимая функция** — блок кода, размещенный в БД, который всегда возвращает единственную величину указанного типа.

Листинг 10.7. Хранимая процедура в БД MovieActor

Хранимая процедура в БД MovieActor

```
SQL> -- Поиск строки в фильме, используя хранимую процедуру
SQL> CREATE OR REPLACE PROCEDURE string_search (string IN
     VARCHAR2) AS
  2        CURSOR c1 IS
  3            SELECT movie_title AS found
  4            FROM movie
  5            WHERE UPPER(movie_title) LIKE
               '%'||UPPER(string)||'%';
  6        BEGIN
  7          FOR c1rec IN c1 LOOP
  8              dbms_output.put_line
               ('Found movie title: '||LPAD(c1rec.found,30));
  9          END LOOP;
 10        END string_search;
 11  /

Procedure created.
```

```
SQL> EXECUTE string_search('vamp');
Found movie title:     Interview with the Vampire

SQL> EXECUTE string_search('the');
Found movie title:     Interview with the Vampire
Found movie title:     The Birdcage
```

В некоторых СУБД хранимые процедуры и функции могут быть сгруппированы в пакеты. **Пакет БД** состоит из двух частей: спецификация и тело. Спецификация содержит объявления и сигнатуры процедур и функций, включенных в пакет. Тело содержит реализации процедур и функций.

Листинг 10.8 показывает пакет БД по имени `MovieSearch` с двумя функциями. Функция `string_search` (поиск строки) имеет те же самые функциональные возможности, что и хранимая процедура `string_search` в листинге 10.7. Функция `leading_actors` (ведущие актеры) выполняет задачу оператора `select` из листинга 10.6.

10.1.6. Индексы

Реляционные БД не поддерживают основанные на указателях навигационные пути между записями. Как следствие этого, в большинстве реляционных БД индексы — единственные пути доступа к записям в таблице кроме последовательного просмотра всей таблицы.

Индекс — структура данных, отдельная от страниц данных, которые хранят сами записи таблицы, и которая состоит из иерархического дерева узлов индекса. Узлы содержат пары ключ — указатель. Ключи индекса содержат значения, извлеченные из одного или нескольких столбцов таблицы, для которых был построен индекс. Ключевые значения сохраняются в отсортированном порядке. Указатели индекса в узлах, не являющихся листьями (то есть в нижней части дерева[1]), определяют узлы в следующем более низком уровне дерева. Указатели индекса в узлах, являющихся листьями, определяют (или непосредственно содержат) записи таблицы, которые имеют то же самое значение ключа, что и в узле индекса.

Листинг 10.8. Пакет и функции в MovieActor

```
Пакет и функции в MovieActor

create or replace package MovieSearch as
    type ref_cursor is REF CURSOR;
    function string_search(string in VARCHAR2)
    return ref_cursor;
    function leading_actors return ref_cursor;
end MovieSearch;
/
```

[1] Напомним, что дерево изображается в «перевернутом» виде. (*Прим. перев.*)

```
create or replace package body MovieSearch is
    function string_search(string in VARCHAR2) return
    ref_cursor is cur ref_cursor;
    begin
        open cur for
            SELECT *
            from movie where UPPER(movie_title) like
                        '%'||UPPER(string)||'%';
        return cur;
    end;
    function leading_actors return ref_cursor is
    cur ref_cursor;
    begin
        open cur for
            SELECT *
            FROM movie m, actor a, listed_as l
           WHERE m.movie_code = l.movie_code AND l.actor_code =
                                a.actor_code AND l.position = 1
        return cur;
    end;
end MovieSearch;
/
```

Реляционные БД допускают динамическое создание и удаление индексов с помощью SQL-операторов `create index` (создать индекс) и `drop index` (удалить индекс). Процесс поддерживает **физическую независимость данных**. Прикладные программы могут выполнять свои задачи, связанные с БД, как с индексами, так и без них. Однако само выполнение программ может существенно изменяться в зависимости от текущего набора индексов в БД.

Вообще говоря, будут ли или нет приложения использовать индекс для данной операции с БД, полностью зависит от СУБД. Конечно же, СУБД может использовать индекс, только если программист БД создаст его. Однако это не означает, что должно быть создано большое количество индексов на всякий случай, чтобы СУБД могла их выбирать для использования. Использование индексов приводит к существенной стоимости сопровождения СУБД.

Изменения в записях БД требуют делать соответствующие изменения и в индексах. Эти изменения в индексах (перестройка индекса) выполняются автоматически самой СУБД на заднем плане, однако выполнение всех приложений, которые в настоящее время «говорят» с БД, может пострадать. Компромисс ясен. Интенсивные модификации БД с большим числом индексов могут замедлить время отклика приложений, но интенсивный поиск в БД со многими индексами может продемонстрировать улучшение выполнения функций. Задача разработчика БД состоит в том, чтобы создать такой набор индексов, который обеспечивает лучшее выполнение функций для всего набора прикладных программ, обращающихся к БД.

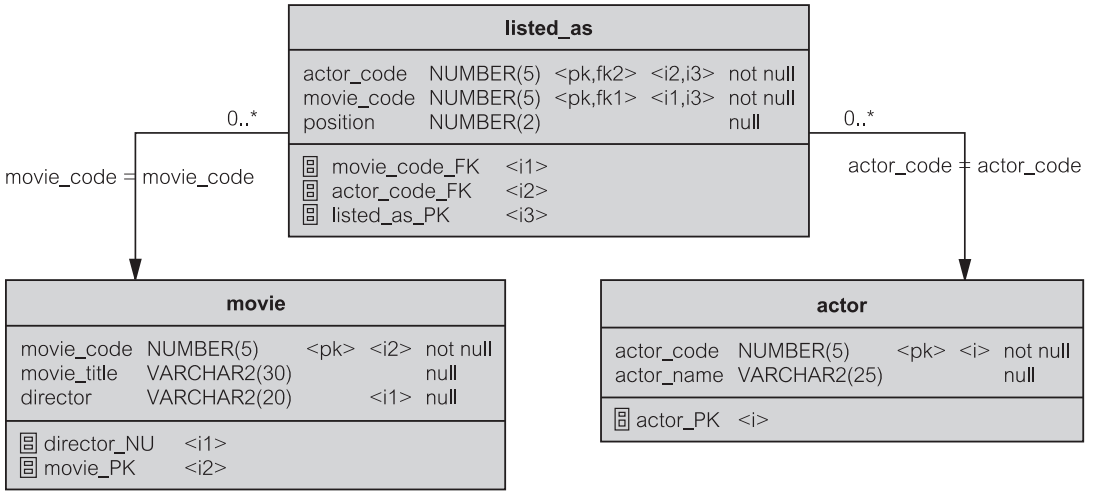


Рис. 10.6. Индексы в физической модели БД для MovieActor

Кроме выполнения своих непосредственных функций, индексы являются посредником СУБД, который гарантирует **уникальность** записей в таблицах. Так как такую уникальность определяет первичный ключ, СУБД строит индекс для каждого первичного ключа, определенного в БД. Если это не требуется явно программистом, индексы первичных ключей создаются неявно самой СУБД.

Как правило, программист должен создать индексы для внешних ключей, чтобы поддержать ссылочную целостность (раздел 10.1.2). В большинстве случаев индексы внешнего ключа **неуникальны** (то есть, таблица может иметь много записей с тем же самым значением внешнего ключа). Оператор `create index` позволяет выбрать вариант, что нужно обеспечить уникальный индекс. В противном случае индекс создается как неуникальный.

Рис. 10.6 показывает физическую модель БД, полученную добавлением индексов к логической модели БД, приведенной на рис. 10.2. Таблица `actor` (актер) имеет только индекс первичного ключа (`actor_PC` — первичный ключ таблицы «актер»). Таблица `movie` (кинофильм) имеет индекс первичного ключа (`movie_PK` — первичный ключ таблицы «фильм») и индекс, называемый `director_NU` (режиссер), указанный как неуникальный (NU) индекс столбца `director`. Таблица `listed_as` (перечисление) имеет индекс первичного ключа (уникальный) — `listed_as_PK` и два (неуникальных) индекса внешних ключей. Ключ `listed_as_PK` определен для двух столбцов первичных ключей одновременно: `actor_code` (код актера) и `movie_code` (код фильма).

Если физическая модель БД определена с помощью CASE-средства, это инструментальное средство может автоматически создать скрипт (сценарий) для формирования всех таблиц и индексов БД. Листинг 10.9 демонстрирует

такой скрипт для проекта, приведенного на рис. 10.6. Обратите внимание, что в зависимости от задания параметров генерации индексы первичных ключей могут создаваться явно (как для таблицы `actor`) или неявно (как для таблицы `movie`). Обычно предпочтителен второй вариант.

Листинг 10.9. SQL-операторы для создания таблиц и индексов в БД MovieActor

SQL-операторы для создания таблиц и индексов в БД MovieActor

```
/*==============================================================*/
/* Table: actor                                                 */
/*==============================================================*/
create table actor (
   actor_code           NUMBER(5)                 not null,
   actor_name           VARCHAR2(25)
)
/
/*==============================================================*/
/* Index: actor_PK                                              */
/*==============================================================*/
create unique index actor_PK on actor (
   actor_code ASC )
)
/
/*==============================================================*/
/* Table: movie                                                 */
/*==============================================================*/
create table movie (
   movie_code           NUMBER(5)                    not null,
   movie_title          VARCHAR2(30),
   director             VARCHAR2(20),
   constraint PK_MOVIE primary key (movie_code)
)
/
/*==============================================================*/
/* Index: director_NU                                           */
/*==============================================================*/
create index director_NU on movie (
   director ASC
)
/
/*==============================================================*/
/* Table: listed_as                                             */
/*==============================================================*/
create table listed_as (
   actor_code           NUMBER(5)                  not null,
   movie_code           NUMBER(5)                  not null,
```

```
   position             NUMBER(2),
   constraint PK_LISTED_AS primary key (actor_code,movie_code),
   constraint FK_LISTED_A_REFERENCE_ACTOR foreign key
        (actor_code) references actor (actor_code)
   on delete cascade
)
/
/*==============================================================*/
/* Index: movie_code_FK                                         */
/*==============================================================*/
create index movie_code_FK on listed_as (
   movie_code ASC
)
/
/*==============================================================*/
/* Index: actor_code_FK                                         */
/*==============================================================*/
create index actor_code_FK on listed_as (
   actor_code ASC
)
/
```

10.2. Отображение временных объектов в сохраняемые записи

Установление соответствия между **временными объектами** в прикладной программе и **сохраняемыми записями** в БД, используемыми приложением, имеет ряд измерений увеличивающейся сложности. Притом в зависимости от того, существует ли уже БД перед тем, как будет разработано приложение, установление соответствия выполняется или как прямое проектирование или как операция обратного проектирования.

Установление соответствия может иметь следующие формы:

1. Прямое соответствие концептуальных классов (раздел 8.3) таблицам.
2. Обратное соответствие таблиц концептуальным классам, которое обычно заменяется двумя параллельными шагами:
   а) обратное соответствие таблиц классам проекта (раздел 9.1.1) и
   б) прямое соответствие концептуальных классов классам проекта.
3. Прямое отображение классов проекта в таблицы. Поскольку классы проекта определяют и атрибуты, и операции, отображение включает также и активные аспекты таблиц, а именно триггеры и хранимые процедуры.
4. Обратное отображение таблиц в классы проекта. Оно включает решение проблем, наподобие того, как на приложение воздействуют активизации триггеров и обращения к хранимым процедурам.

5. Кэширование записей таблиц в виде объектов класса в памяти программы. Оно включает анализ выполнения транзакций БД в случае успешной работы или при отказах, а также фиксацию результатов или откат в изменениях БД.

На отображение может также влиять наличие промежуточного ПО и API-функций между прикладными программами клиента и сервером корпоративной реляционной БД. Промежуточное ПО может, в конечном счете, привести к трехуровневой структуре с сохраняемым информационным хранилищем объектов между приложением и реляционной БД. Однако модели объектных БД имеют небольшую историю.

10.2.1. Объектные БД, SQL:1999 и потеря соответствия

Модель объектной БД — конкурент реляционной модели. **Группа управления объектными данными** (Object Data Management Group — ODMG) разрабатывает стандарты для объектных БД. Объектные СУБД привлекли пристальное внимание в середине 1990-х годов, особенно в области СУБД-приложений, связанных с мультимедиа и функционированием рабочих групп.

В недавние годы стандарты ODMG сделали крен от определения объектных СУБД к определению **программного интерфейса хранилища объектов** (Object Storage API). Задача программного интерфейса хранилища объектов состоит в том, чтобы обеспечить уровень ПО между прикладной программой и реляционной (или другой) БД для связи объектов приложения с реляционными записями и наоборот. Отображение может включать использование сохраняемой объектной БД. Главная цель такой объектной БД состоит в том, чтобы установить отображение и зависимость от реляционной БД для получения большого количества обычных функций БД, типа управления транзакциями.

С точки зрения шаблона PCMEF (раздел 9.2.2) размещение программного интерфейса хранилища объектов между приложением и реляционной БД завершается перемещением многих задач, выполняемых классами уровней `domain` (предметная область) и `foundation` (основание) в программный интерфейс хранилища объектов. Проект этих классов должен теперь соответствовать программному интерфейсу хранилища объектов, и они отчасти относятся к API.

Подъем и упадок объектных БД были отмечены появлением нового объектно-реляционного стандарта БД, **SQL:1999**. Новый стандарт пытается добавить объектно-ориентированные особенности к реляционной модели. Задача очень трудна из-за основных философских различий между объектной и реляционной моделью, типа использования навигационных связей. Задача формирования таблиц напоминает создание объектов на сложном уровне поддержки ПО, из которого разработчики пока не сумели извлечь существенную пользу. Сюда следует включить объектно-ориентированные особенности в Oracle, начиная с Oracle8.

SQL:1999 не устраняет один из главных камней преткновения в связи между прикладными программами и реляционными БД — **потерю соответ-**

ствия (impedance mismatch)[2]. Это относится к невозможности использования прикладного языка (типа Java), чтобы непосредственно управлять данными в БД без использования SQL. Декларативный характер SQL управляет данными как множеством записей. Все популярные прикладные языки являются процедурными по своей природе и управляют данными как конкретными записями. Это языки, работающие в режиме последовательной обработки записей, в противоположность SQL, работающему в режиме последовательной обработки множеств. SQL обеспечивает механизм **курсора** (листинг 10.7), чтобы устранить несогласованность.

Потеря соответствия является причиной разработки стратегии отображения классов в таблицы и наоборот. Но это — не единственная причина. Другая причина — необходимо иметь различие между сохраняемыми записями в БД и временными объектами сущностей в кэше памяти прикладной программы. В особо крупных системах кэширование записей, извлеченных из БД, как объектов, доступных для программы, является основной обязанностью проекта. Эта проблема частично рассматривается в главе 12 и более глубоко в части 4. Следующий раздел обсуждает только отображение концептуальных классов в таблицы.

10.2.2. Объектно-реляционное отображение

Раздел 10.1.3 содержит пример отображения таблиц **логических моделей БД** в **концептуальные классы** UML-моделей классов. Вплоть до границ логического моделирования БД пример является весьма общим.

Данный раздел связан с отображением концептуальных классов (в объектно-ориентированной программе) в таблицы (в реляционной БД). Это часто называется **объектно-реляционным отображением** [66] — несколько «перегруженным» термином, который не следует путать с понятием объектно-реляционной БД. Отображение двунаправлено. Оно может быть *от объектов к таблицам*, когда БД еще не существует во время проектирования приложения. Оно может быть и *от таблиц к объектам*, когда БД уже существует и когда прикладная программа обслуживает находящийся в памяти кэш данных из БД (глава 12).

Семантическая сила логического моделирования БД ограничена по сравнению с семантической выразительностью моделирования классов. Это отражает основную простоту реляционной модели: таблица и ссылочная целостность — два единственных примитива моделирования. Объектно-ориентированные концепции, такие как наследование типов, интерфейсы, агрегирование и т. д., — не являются частями реляционной модели. Часто процедурные решения должны быть запрограммированы в реляционной БД, чтобы полностью выразить семантику декларативной модели, представленную в диаграмме классов.

[2] Здесь опять возникает уже не раз упоминавшаяся проблема. Термин impedance mismatch переводится различными авторами по-разному: «несоответствие объектной и реляционной модели», «несоответствие импеданса», «устойчивость к ошибкам», «потеря соответствия» и даже «полное сопротивление несоответствию». Все это говорит о том, что тематика книги настолько современна, что даже использует еще не устоявшуюся терминологию. В данном случае мы будем применять наиболее часто встречающийся термин — потеря соответствия. (*Прим. перев.*)

Концептуальные классы — просто контейнеры данных. Операции обычно не рассматриваются в концептуальных классах. Это делает отображение в таблицы более простым. Главная проблема заключается в отображении отношений между классами. Рассматриваемые отношения включают ассоциации с различными множественностями, агрегированиями и обобщениями.

За исключением самых простых моделей класса имеется более чем один возможный результат для отображения каждого отношения. Следующие разделы представляют наиболее вероятные результаты. Дается только одно отображение, предусмотренное для каждого сценария, но в дальнейшем обсуждаются также основные альтернативные решения. Чтобы облегчить понимание и сравнение механики отображения, все примеры обращаются к одним и тем же структурам классов, а также используют символический класс и имена атрибутов. Чтобы сформировать все результаты отображения, использовалось CASE-средство.

Отображение ассоциации и агрегирования «один ко многим»

Ассоциации «один ко многим» между концептуальными классами наиболее часто используются и являются самыми легкими для понимания и отображения в таблицы. Класс на стороне «один» ассоциации отображается в таблицу без каких-либо изменений, кроме использования в столбцах реляционных типов данных. Класс на стороне «многие» отображается другой таблицей с внешним ключом, добавленным, чтобы реализовать ассоциацию в терминах реляционной ссылочной целостности.

Рис. 10.7 дает пример, где в структуру таблиц отображены ассоциация «один ко многим» и эквивалентное агрегирование «один ко многим». Ясно, что любая дополнительная семантика агрегирования по сравнению с ассоциацией теряется при отображении. Любая такая семантика должна быть реализована отдельно в процедурной логике БД.

Отображение предполагает, что идентификаторы были определены в концептуальных классах, как обозначено в примечаниях на рис. 10.7. UML не имеет никакого визуального способа показать идентификаторы классов, кроме вспомогательных стереотипов или примечаний. Это совместимо с объектно-ориентированной философией, где объекты идентифицируются автоматически формируемыми неизменными идентификаторами объектов (Object Identifier — OID) [61] скорее, нежели используя определенные пользователем атрибуты. Однако большинство CASE-средств позволяет определять идентификаторы, вводя их как дополнительные свойства атрибутов класса.

`Attribute_1`, введенный в таблицу `YYY` как внешний ключ, допускает значения `null`. Это отражает необязательное участие объектов `YYY` в ассоциации с `XXX`. Если множественность была определена как `1..1`, показывая таким образом обязательное участие, то `Attribute_1` был бы определен как `not null`.

Отображение ассоциации «многие ко многим»

Модель реляционной БД неспособна непосредственно обеспечить отношение «многие ко многим» между таблицами. Это потому, что значение внешнего ключа в любой записи может быть связано только с одной величиной пер-

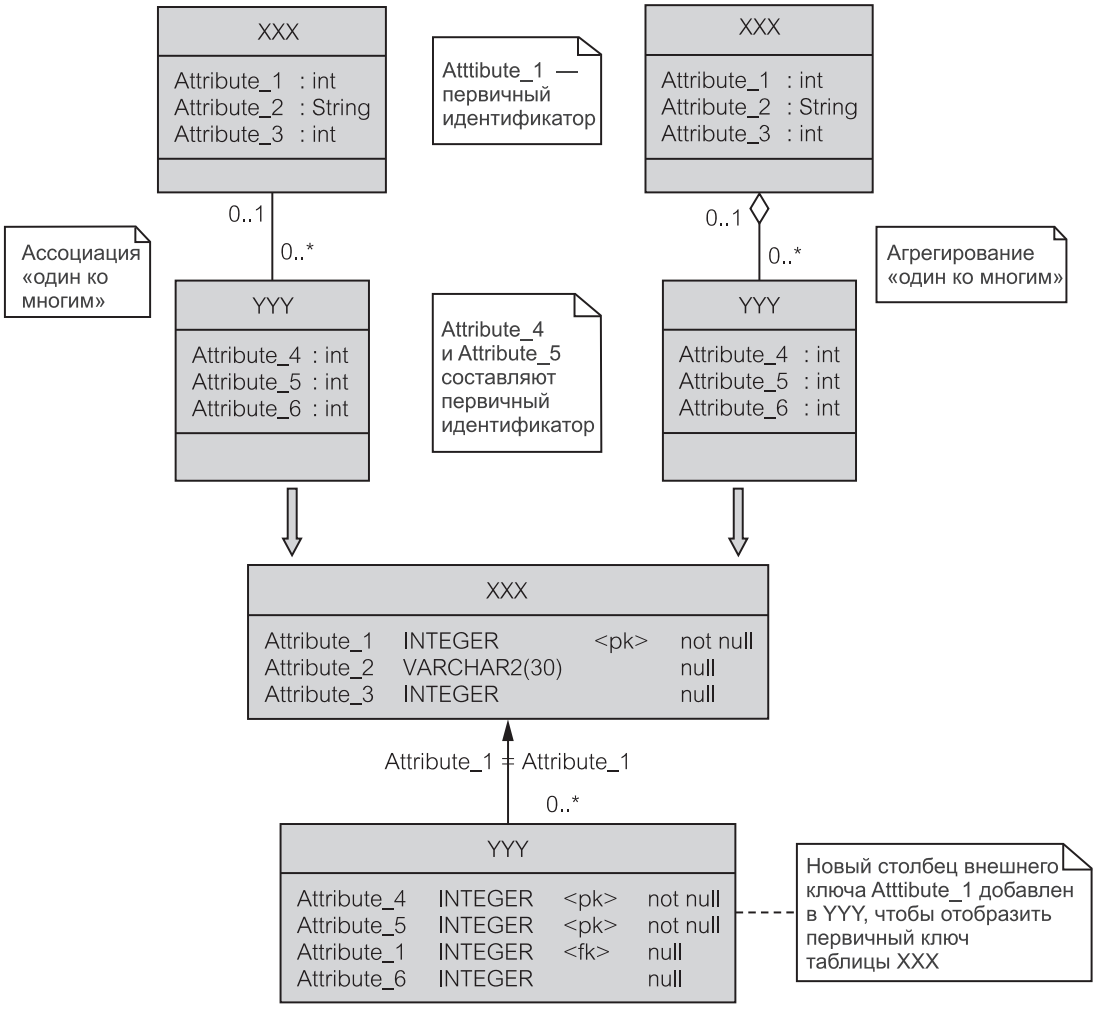


Рис. 10.7. Отображение ассоциации и агрегирования «один ко многим»

вичного ключа в другой записи. Чтобы обеспечить отношение «многие ко многим», реляционная модель вынуждена ввести «таблицу отношений», чтобы преобразовать одну ассоциацию «многие ко многим» для классов к двум отношениям «один ко многим» для таблиц.

Рис. 10.8 показывает такой пример. Добавленное примечание объясняет принцип отображения. Заметьте, что `Attribute_1` в `Association_1` определен как `not null`, несмотря на то, что это внешний ключ к `XXX`, как и на рис. 10.7, где допускались значения `null`. Простое объяснение этому заключается в том, что `Attribute_1` также является частью первичного ключа `Association_1`, а никакая часть первичного ключа не может принимать значение `null`. Более глубокое объяснение — это то, что любая запись `YYY` со значением `null` в `Attribute_1` на рис. 10.7 может все еще появляться

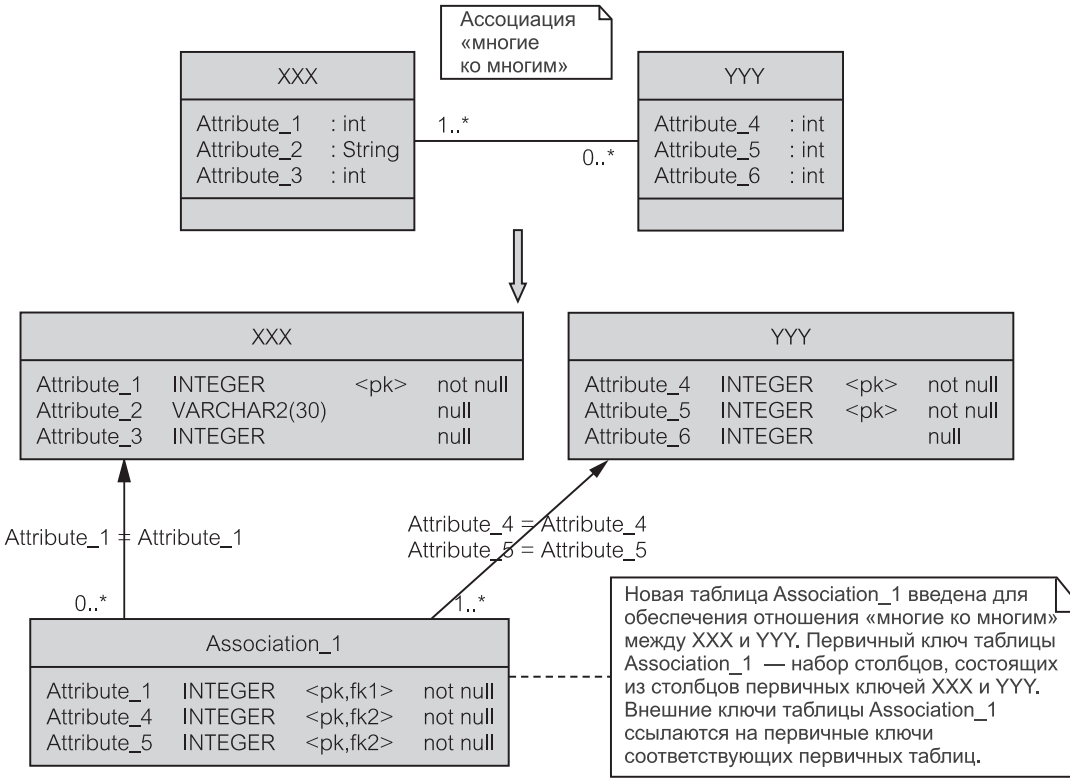


Рис. 10.8. Отображение ассоциации «многие ко многим»

как запись `YYY` на рис. 10.8, но `YYY` не содержит значения `Attribute_1`. `YYY`, который не связан с `XXX`, — просто не будет иметь никакой записи в `Association_1`.

Отображение ассоциации «один к одному»

Ассоциации «один к одному» между классами, вероятно, самые простые для понимания, но они оказываются не самыми простыми для преобразования в реляционную модель. Отображение, предлагаемое на рис. 10.9, самое очевидное, но вводит чрезмерное дублирование столбцов и циклические зависимости.

В зависимости от обстоятельств лучшим решением может быть объединение обоих классов в единственную таблицу. Альтернативное решение подобно приведенному на рис. 10.8, которое выделяет внешние ключи в «таблицу отношений», может быть предпочтительней. Заметьте также, что «таблица отношений» является необходимой во всех случаях, когда определен класс ассоциации (сравните с рис. 10.4) на любой ассоциации между классами, включая и ассоциацию «один к одному».

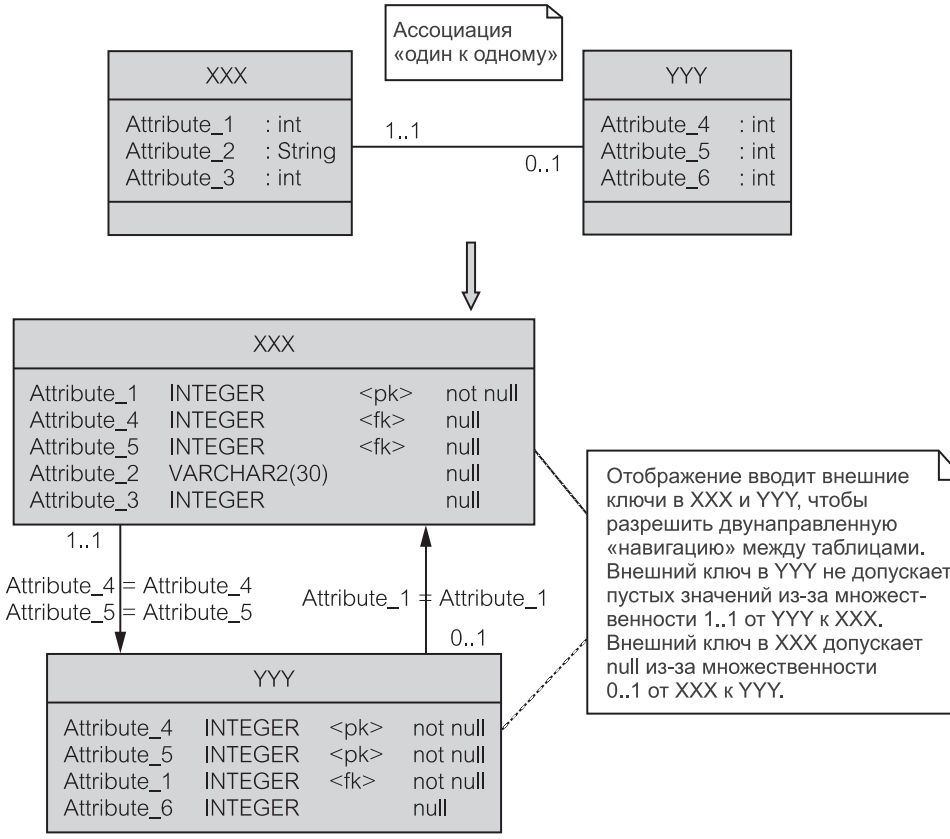


Рис. 10.9. Отображение ассоциации «один к одному»

Отображение рекурсивной ассоциации «один ко многим»

Рекурсивные (то есть **ссылающиеся на саму себя**) **ассоциации**, основанные на единственном классе, не так уж редки в моделировании. Классический учебный пример рекурсивной ассоциации «один ко многим» — ассоциация, охватывающая иерархическую структуру служащих в организации. В такой структуре служащий, являющийся менеджером, управляет другими служащими, являющимися подчиненными. Служащий-подчиненный ответствен только перед одним служащим-руководителем.

Рис. 10.10 показывает UML-нотацию для рекурсивных ассоциаций и отображение в реляционную структуру. Отображение вводит столбец внешнего ключа, который получен из определения первичного идентификатора. Значения внешнего ключа в `XXX` соответствуют значению первичного ключа в `XXX` (кроме случаев, когда значение внешнего ключа — `null`).

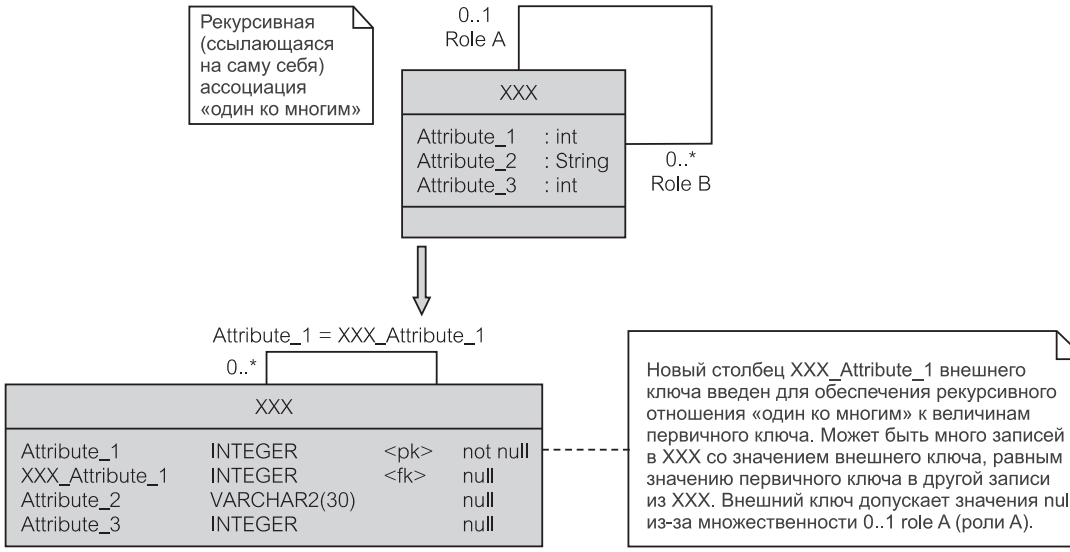


Рис. 10.10. Отображение рекурсивной ассоциации «один ко многим»

Отображение рекурсивной ассоциации «многие ко многим»

Классический учебный пример для рекурсивной ассоциации «многие ко многим» — задача о «списке материалов», известная также как задача о разбиении или разбиении на части. Изделия состоят из частей (компьютер состоит из процессора, памяти и т. д.). Каждая часть может состоять из меньших частей (подчастей). Одна и та же подчасть (как концепция, а не экземпляр) может использоваться в производстве различных изделий. Фактически, само изделие в этом смысле является частью. Ясно, что ассоциация частей рекурсивна и к тому же относится к категории «многие ко многим».

Как показано на рис. 10.11, отображение рекурсивной ассоциации «многие ко многим» напоминает отображение ассоциации «многие ко многим» для двух классов (рис. 10.8). Чтобы отобразить такую ассоциацию, требуется «таблица отношений».

Отображение обобщения

Как показано в разделе 7.4.2, использование **обобщения** для моделирования концептуальных классов не поощряется. Кроме того, цель отображения, то есть, модель реляционной БД, не понимает наследование атрибутов, которое подкрепляет понятие обобщения.

Наследование может очень несовершенно представляться в реляционной модели. Одно решение — прямое дублирование атрибутов суперкласса в «таблице подкласса». Другое решение состоит в том, чтобы не иметь «таблицы суперкласса» вообще, и создать «таблицу подкласса», чтобы представить и базовый (суперкласс), и конкретный (подкласс) типы. Решение, которое кажется наиболее привлекательным, показано на рис. 10.12. Первичный иден-

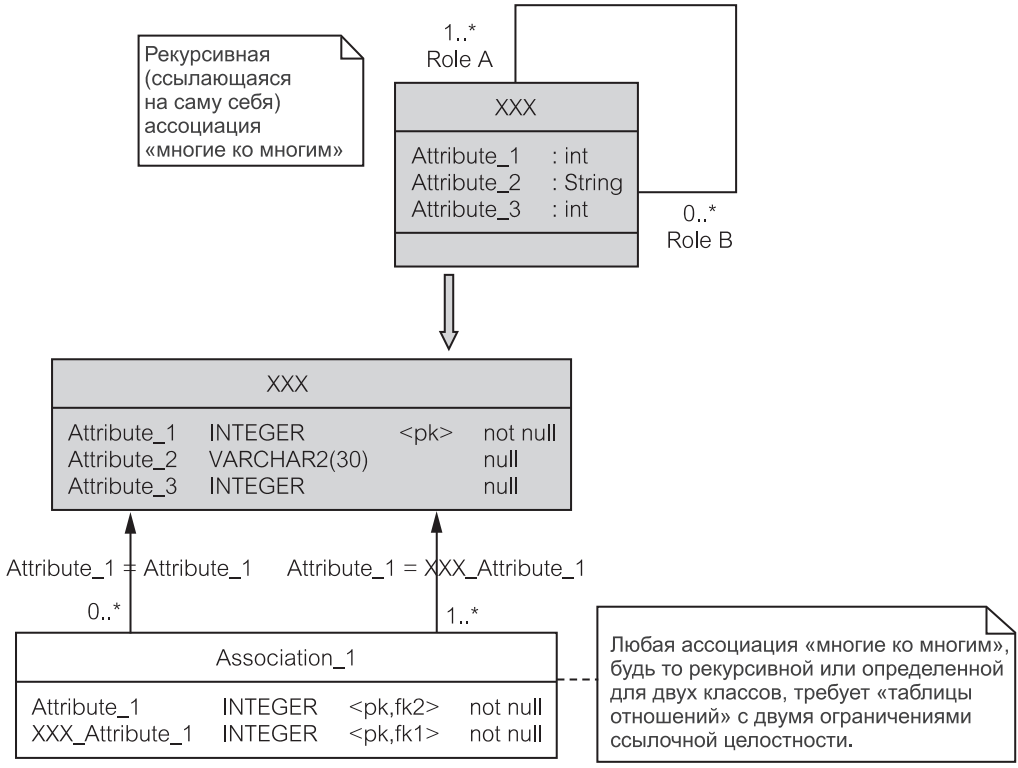


Рис. 10.11. Отображение рекурсивной ассоциации «многие ко многим»

тификатор суперкласса дублирован в «таблице подкласса» и используется как ее первичный ключ (или часть ее первичного ключа) и как внешний ключ к «таблице суперкласса».

10.3. Проектирование и создание БД для управления электронной почтой

Итерация 1 из учебного примера ЕМ предполагает, что БД существует заранее и загружена данными относительно служащих, деловых партнеров и исходящих сообщений. Прикладная программа ЕМ извлекает эту информацию из БД, готовит и посылает электронные сообщения (исходящие сообщения) и корректирует БД, чтобы отметить, какие исходящие сообщения уже были посланы. Следующие итерации позволят помещать в БД исходящие сообщения и связанную с ними информацию из прикладной программы.

Концептуальная диаграмма классов для ЕМ была определена как часть требований итерации 1 в главе 8 (рис. 8.8). Диаграмма повторена ниже на рис. 10.13 для удобства.

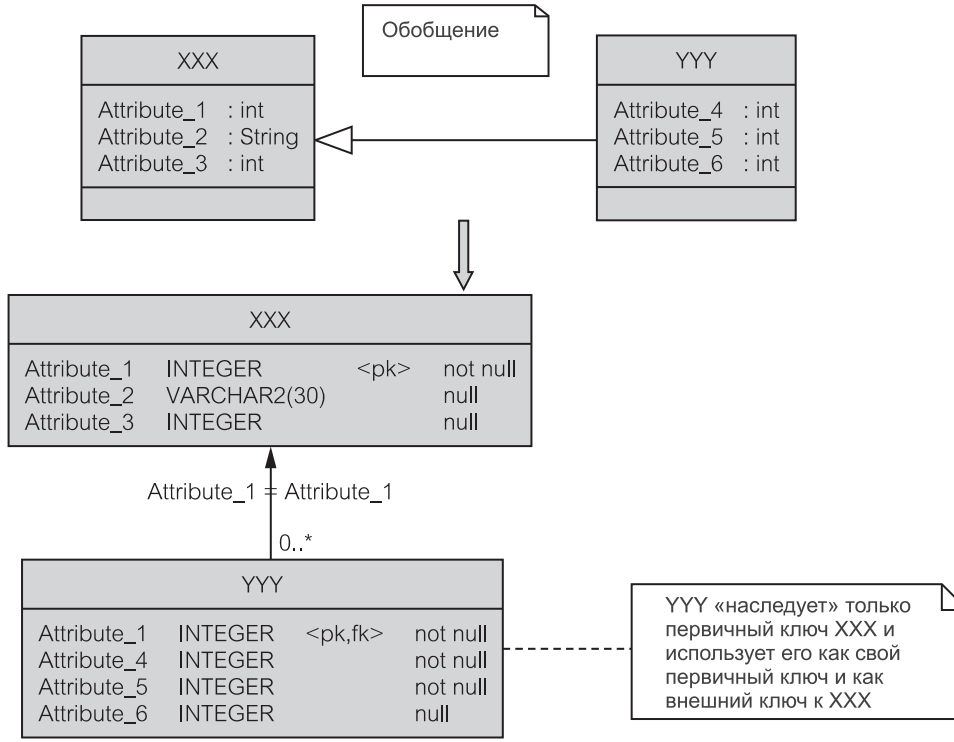


Рис. 10.12. Отображение обобщения

10.3.1. Модель БД

Рис. 10.14 представляет физическую модель БД, полученную из концептуальной модели, изображенной на рис. 10.13. Модель физическая, а не только логическая, потому что она определяет физические характеристики (индексы) и использует типы данных конкретной СУБД (Oracle). Отображение концептуальной модели достаточно прямое и очевидное.

Кроме неявных индексов на первичных ключах определен дополнительный уникальный индекс (`login_UN` — уникальное регистрационное имя) на `login_name` (регистрационное имя) в таблице `Employee` (служащий). Этот индекс используется, чтобы засвидетельствовать, что пользователь программы работает под `login_name` БД, который внесен в список таблицы `Employee`. Такой пользователь рассматривается как законный служащий, уполномоченный использовать подсистему EM.

Нет никаких индексов, определенных на внешних ключах. Считается, что таблицы будут содержать относительно небольшое число записей (скорее тысячи, чем миллионы) и индексы внешнего ключа будут редко использоваться в СУБД, но увеличат стоимость сопровождения.

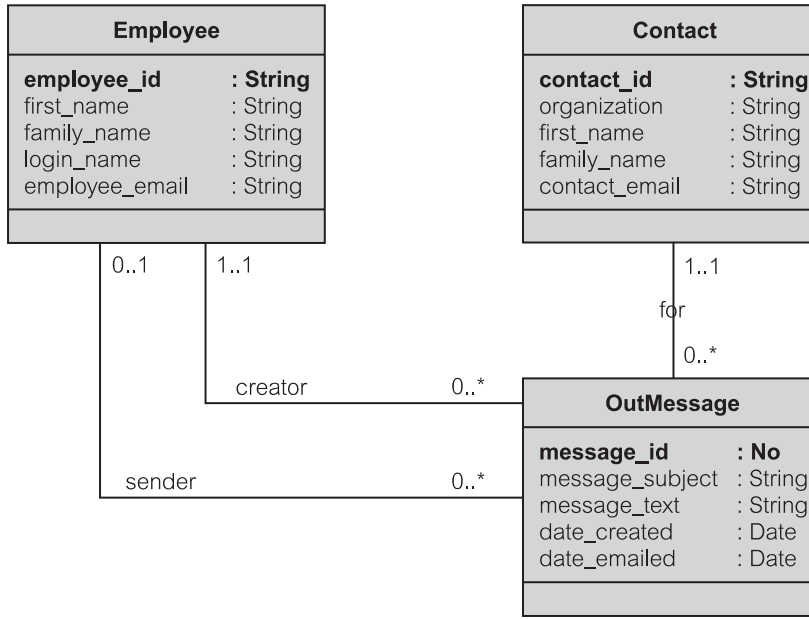


Рис. 10.13. Диаграмма концептуальных классов для EM

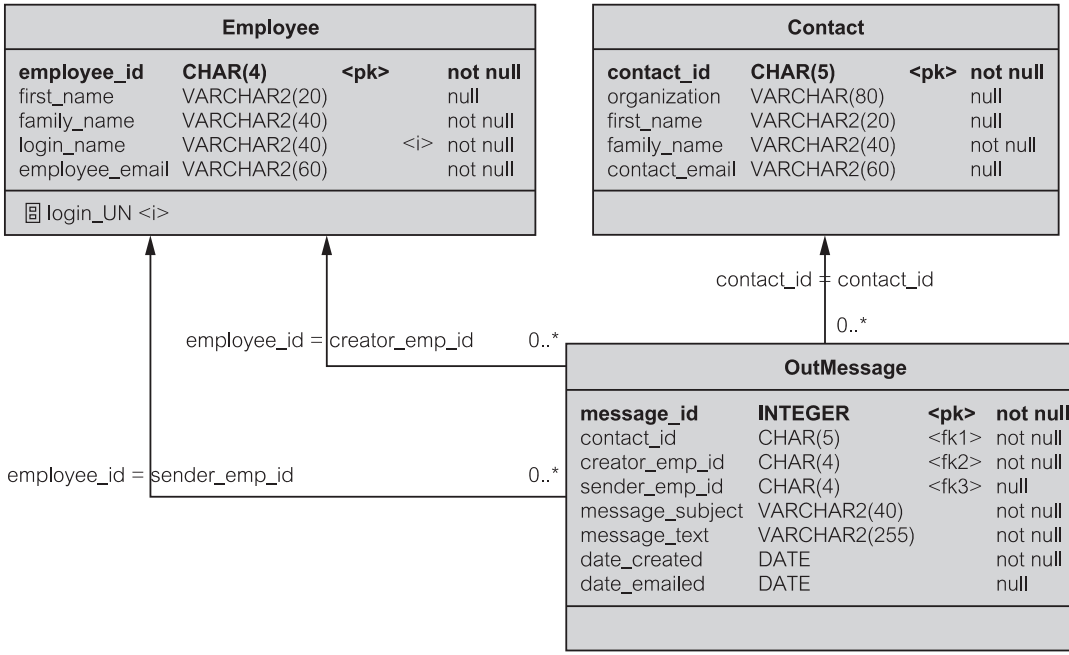


Рис. 10.14. Логическая схема БД для EM

10.3.2. Создание схемы БД

CASE-средство может автоматически формировать SQL-операторы создания из физической модели БД. Эти операторы могут запускаться в БД, чтобы создать схему БД (структуру таблиц БД). Листинг 10.10 представляет такой автоматически произведенный SQL-скрипт для создания схемы БД EM.

Листинг 10.10. SQL-скрипт для создания схемы БД EM

SQL-скрипт для создания схемы БД EM

```
/*==============================================================*/
/*Table: Contact                                                */
/*==============================================================*/
create table Contact (
   contact_id          CHAR(5)                       not null,
   organization        VARCHAR(80),
   first_name          VARCHAR2(20),
   family_name         VARCHAR2(40)                  not null,
   contact_email       VARCHAR2(60),
   constraint PK_CONTACT primary key (contact_id)
)
/
/*==============================================================*/
/*Table: Employee                                               */
/*==============================================================*/
create table Employee (
   employee_id         CHAR(4)                       not null,
   first_name          VARCHAR2(20),
   family_name         VARCHAR2(40)                  not null,
   login_name          VARCHAR2(40)                  not null,
   employee_email      VARCHAR2(60)                  not null,
   constraint PK_EMPLOYEE primary key (employee_id)
)
/
/*==============================================================*/
/* Index: login_UN                                              */
/*==============================================================*/
create unique index login_UN on Employee (
   login_name ASC
)
/
/*==============================================================*/
/* Table: OutMessage                                            */
/*==============================================================*/
create table OutMessage (
   message_id          INTEGER                       not null,
   contact_id          CHAR(5)                       not null,
```

```
   creator_emp_id      CHAR(4)                        not null,
   sender_emp_id       CHAR(4),
   message_subject     VARCHAR2(40)                   not null,
   message_text        VARCHAR2(255)                  not null,
   date_created        DATE                           not null,
   date_emailed        DATE,
constraint PK_OUTMESSAGE primary key (message_id),
constraint FK_OUTMES_REF_CONTACT foreign key (contact_id)
      references Contact (contact_id),
constraint FK_OUTMES_REF_CREATOREMP foreign key
      (creator_emp_id)
      references Employee (employee_id),
constraint FK_OUTMES_REF_SENDEREMP foreign key (sender_emp_id)
      references Employee (employee_id)
)
/
```

10.3.3. Пример содержимого БД

После того как схема БД будет создана, структуры таблиц могут быть загружены (населены) данными. Итерация 1 ЕМ предполагает, что БД загружена и что загрузка происходит из многих пунктов всей системы АЕМ. Для целей этой книги SQL-скрипт с операторами `insert` (добавление), чтобы загрузить БД, достаточен (скрипт, как и весь остальной код для учебного примера, можно получить с Web-сайта книги).

Листинг 10.11 представляет пример содержимого БД для ЕМ после того, как скрипт размещения был запущен, чтобы загрузить данные. Анализ содержимого БД может быть полезен для лучшего понимания требований работы подсистемы ЕМ.

Листинг 10.11. Пример содержимого БД ЕМ

```
Пример содержимого БД ЕМ

SQL> select * from Employee;

EMPLOYEE_ID FIRST_NAME FAMILY_NAME LOGIN_NAME EMPLOYEE_EMAIL
----------- ---------- ----------- ---------- --------------
E011        Leszek     Maciaszek   leszek11   leszek@ics.
                                              mq.edu.au
E012        Bruc Lee   Liong       bruc11     bliong@ics.
                                              mq.edu.au
E013        Stephen    Bills       steve11    sbills@acnielsen.
                                              com.au

SQL> select * from Contact;
```

```
CONTACT_ID ORGANIZATION FIRST_NAME FAMILY_NAME CONTACT_EMAIL
---------- ------------ ---------- ----------- -------------
C1234      SBS Sydney   Pablo      Romero      promero@sbs.
                                               com.au
C4321      Channel 9    Vincent    Buckley     vbuckley@chnine.
                                               com.au
C2233      ABC Radio    Dorothy    Norris      dnorris@abc.
                                               com.au
C5887      Ford         Agatha     Sommer      asommer@ford.
                                               com.au
           Australia
```

```
SQL> select contact_id, creator_emp_id, sender_emp_id,
message_subject, message_text
  2  from OutMessage
  3  order by contact_id;
```

```
CONTACT_ID CREATOR_EMP_ID SENDER_EMP_ID MESSAGE_SUBJECT
---------- -------------- ------------- ---------------
MESSAGE_TEXT
-------------
C1234      E011                         Product missing
The Product name for your ad CI223375XY is missing. Can you
please supply it?

C2233      E012                         Log incomplete
Log LG02JUN20 does not contain any ad instances between 1 and 4
pm. Please re-send a corrected log.

C2233      E011                         Log incorrect
Log SZ020620B contains ad instances that overlap with program
times between 9:30 and 11:00 am. Looks like durations of some
ad instances are incorrect.

C4321      E011                         Agency missing
What is the agency which booked the advertisement DL5FG88779,
please?

C5887      E013                         Contract renewal
Your contract with us is due for renewal next month. We will
mail the new contract to you for your signing at the beginning
of next month. Please inform us if you require changes to the
existing
contract details.
C5887      E011                         Are you satisfied?
This is just a courtesy email to ask if you are satisfied with
the quality and timeliness of the advertising expenditure re-
ports that we supply to you as per our current contract. Please
let me know if our service can be improved in any way.
```

Резюме

1. *База данных (БД)* является большой, сохраняемой, многопользовательской, восстанавливаемой, последовательной, безопасной и расширяемой.
2. Доминирующая модель БД для информационных систем бизнеса — *реляционная модель*. Реляционная модель представляет данные как *записи* (строки) в *таблицах* (отношениях).
3. *SQL* — язык для обработки данных в реляционных БД. Имеются два основных варианта SQL: *интерактивный* SQL и *вложенный* SQL.
4. *Схема БД* (реляционной) представляет собой множество определений таблиц для конкретной предметной области. Таблицы связаны и объединены посредством ограничений *ссылочной целостности*.
5. Схема БД может быть представлена на различных уровнях абстракции — как *концептуальная* модель БД, *логическая* модель БД или *физическая* модель БД.
6. Бизнес-правила, которым БД должна следовать, могут быть запрограммированы посредством *декларативной ссылочной целостности* и *процедурной ссылочной целостности* (*триггеров*).
7. Логика приложения может быть запрограммирована с использованием *вложенного SQL* и с помощью *хранимых процедур*.
8. *Индексы* — технология БД, которая улучшает выполнение функций системы и гарантирует уникальность записей в таблицах.
9. Важная задача разработки ПО состоит в том, чтобы отобразить *временные объекты* прикладной программы в *сохраняемые записи* БД. Это называется *объектно-реляционным отображением*. Отображение выполняется тяжело из-за возможности *потери соответствия*.

Ключевые термины

Обзорные вопросы

Вопросы для обсуждения

1. Каково наиболее характерное различие между структурами реляционных данных и структурами объекта? Объясните последствия.
2. Объясните, как используются диалоговый и вложенный SQL в разработке СУБД-приложения.
3. Можно ли позволить пустые значения для столбца внешнего ключа? Объясните на примере.
4. Какова наиболее популярная технология для моделирования концептуальных структур БД?
5. Сравните декларативную и процедурную ссылочную целостность. Являются они конкурирующими или сотрудничающими технологиями?
6. Сравните использование вложенного SQL и хранимых процедур в разработке СУБД-приложения.
7. Обсудите основные «за» и «против» индексов БД.
8. Являются ли технологии объектов и реляционных БД конкурирующими или сотрудничающими? Почему вы думаете, что дело обстоит именно так?

Вопросы учебного примера

1. Рассмотрите схему БД EM на рис. 10.14. Почему `sender_emp_id` (идентификатор служащего, посылающего сообщение) в `OutMessage` (исходящее сообщение) допускает значение `null`?

2. Проект таблицы `Employee` (служащий) в схеме БД ЕМ на рис. 10.14 создал бы проблемы, если приложения, отличные от ЕМ, захотели бы получить доступ к той же самой таблице. Какие здесь проблемы? Как они могут быть исправлены для использования другими приложениями, и как проблема может быть переадресована в ЕМ?
3. Проект таблицы `Contact` (деловой партнер) в схеме БД ЕМ на рис. 10.14 допускает значения `null` в столбцах `organization` и `contact_email`. Каковы причины для разрешения пустых значений в этих двух столбцах?

Примеры задач

Упражнения учебного примера

1. Рассмотрим схему БД ЕМ на рис. 10.14. Необходимо различать служащих, которые намечены для рассылки исходящих сообщений и которые являются фактическими отправителями их. Измените схему, чтобы учесть это новое требование.
2. Рассмотрим схему БД ЕМ на рис. 10.14. Необходимо различать `PersonContact` (партнер-личность) и `OrganizationContact` (партнер-организация) как две категории `Contact` (деловой партнер). Хотя `OutMessage` (исходящее сообщение) всегда относится к `Contact`, деловым партнером может быть основной электронный адрес организации. В некоторых случаях это может быть единственный электронный адрес, который содержит ЕМ для каждого делового партнера в этой организации. Возможно также, что `PersonContact` может быть индивидуумом, чья организация неизвестна. Измените схему, чтобы учесть эти новые требования.

Небольшой проект — управление информацией о партнерах

Обратимся к CIM (Contact Information Management — управление информацией о партнерах) небольшого проекта в главе 9. Ответьте на следующие вопросы независимо от того, имеете ли вы решение вопросов небольшого проекта главы 9.

1. Разработайте логическую модель БД для CIM.
2. Напишите код триггера для следующего бизнес-правила. Если деловой партнер «завершен», информация об его почтовом адресе и адресе для курьера должна быть удалена из БД. Помещение величин в столбцы `end_date` (дата завершения) и `emp_id_ended` (завершение работы служащего) «завершает» делового партнера. Деловой партнер не удаляется, он только отмечается как завершенный. Код триггера может быть псевдокодом.
3. Триггер, требуемый в вопросе 2, не проверяет, имеет ли `emp_id_ended` или `date_ended` (дата завершения) значения. Следовательно, возможно изменить `emp_id_ended` и/или `date_ended` на пустые значения. Это означает, что `CourierAddress` (адрес для курьерской почты) и `PostalAddress` (почтовый адрес) могут быть удалены, даже если кор-

ректировка неправильна (то есть изменение значения столбца в `null`). Обеспечьте улучшенную версию кода триггера, которая решает проблему работы с пустым значением.

4. В некоторых СУБД типа Oracle действие триггера (например, удаление `PostalAddress` и `CourierAddress`) может быть выполнено *прежде* или *после* обработки события триггера (например, изменение `emp_id_ended` и `date_ended`). Когда оно назначается триггеру, запускаемому *до* или *после* возникновения события?

Глава

11

Проектирование классов и взаимодействия

Проектирование классов является таким процессом, который должен обеспечить их поведение, заданное в модели сценариев использования (глава 8) при соответствии структурному шаблону, выбранному для системы (глава 9). **Проектирование классов** начинается с идентификации классов (и интерфейсов) и с их распределения по пакетам. Это сопровождается определением особенностей классов (раздел 9.1.1), необходимых для реализации сценариев использования. Далее устанавливаются зависимости и отношения классов.

Практически, проектирование классов неотделимо от **проектирования взаимодействия**, так что имеет смысл обсуждать их в одной главе. Как только исходные классы будут выбраны, модели взаимодействия обеспечивают полезный руководящий принцип для определения особенностей классов и в первую очередь операций класса. Эта глава начинается с проектирования классов, которое в дальнейшем уточняется, чтобы приспособить возможности проекта взаимодействия.

Проектирование классов связано с **классами проекта** (раздел 9.1.1) в противоположность классам анализа. Проектирование классов учитывает платформу реализации. Это включает определение БД (глава 10) и среды программирования.

Понятие проектирования классов включает также проектирование **интерфейсов** (раздел 9.1.7). Создание многих интерфейсов является последствием принципов и механизмов, требуемых структурным шаблоном. Они увеличивают понимание программы, удобство сопровождения и масштабируемость.

Во время выполнения объектно-ориентированная система представляет собой набор взаимодействующих объектов. Каждый объект обеспечивает сервисы, определенные его классом. Следовательно, определение классов является ключевой задачей объектно-ориентированного анализа и проектирования. Анализ системы концентрируется на определении *концептуальных классов* (раздел 8.3). Другие **прикладные классы** (раздел 8.3) во время анализа могут быть (а могут и не быть) определены на любом уровне детализации. Во время проектирования прикладные классы получают название *классов проекта*, чтобы лучше отразить их место в жизненном цикле ПО. Прилагательные (концептуальный, прикладной, проекта и т. д.) могут быть опущены, когда они очевидны из контекста.

Определение классов проекта использует две главные движущие силы: требования реализации сценария использования и технические характеристи-

ки, определяемые структурным проектированием. Потоки событий в документации сценариев использования (раздел 8.2) могут заставить проектировщика сосредоточить каждый класс на единственную цель (класс должен делать что-то одно). Структурный шаблон PCMEF+ в этой книге (раздел 9.2.2) диктует распределение классов по пакетам, соглашения насчет наименований и структурные принципы, которым должны следовать классы.

Классы должны иметь причину для своего существования и обязанности выполнять определенные функции. UML определяет **обязанность** как «контракт или обязательство классификатора» [106]. Концепция классификатора включает как классы, так и интерфейсы. Понятно, что интерфейсы могут и должны быть выбраны в процессе определения классов.

Главное правило назначения обязанностей классу — это то, что класс, который имеет соответствующие данные, должен и выполнять соответствующие обязанности. Если все данные находятся в одном классе, проблема решена. Если данные распределены по многим классам, возможны следующие решения [88, 55]:

- Поместить обязанности в один класс (часто в класс-представление) и делегировать части работы другим классам, содержащим данные.
- Создать класс «третьего лица» (часто класс пакета `control` — управление или `mediator` — посредник), назначить ему обязанности и при необходимости заставить его делегировать части работы другим классам.
- Использовать существующий класс как класс «третьего лица».

Следующее рассмотрение делит процесс на определение классов, исходя из требований сценария использования, и разработку исходного проекта, используя структурные рассмотрения. В действительности, этими двумя рассмотрениями, так же, как и любыми другими соображениями относительно проектирования классов, занимаются в параллель. Последовательность, принятая здесь, — от конкретного к абстрактному. Характеристики сценария использования требуют конкретных функциональных возможностей с конкретными классами. Абстрактные классы и интерфейсы — не главное соображение. Акцент изменяется, когда учитываются структурные требования. Нефункциональные характеристики требуют интерфейсов.

Проектирование взаимодействия обеспечивает две цели: проверку проекта существующих классов и дополнение их дальнейшими деталями. В частности, могут быть определены **сигнатуры** (списки аргументов) операций (методов) классов. Две главные UML-технологии моделирования, поддерживающие проектирование взаимодействия, — это **диаграммы последовательности действий** и **диаграммы связей** (известные до UML 2.0 как **диаграммы сотрудничества**).

11.1. Определение классов из требований сценария использования

Классы должны удовлетворять и обеспечивать поведение, указанное в требованиях сценария использования. Это **функциональные требования**, опреде-

ленные в документе сценария использования (раздел 8.2). Определение классов по требованиям сценария использования включает выделение этих требований из документа сценария и выбор классов и их совместной работы, необходимых для выполнения этих требований.

Процесс определения классов из требований сценария использования должен гарантировать, что классы и их совместная работа твердо придерживаются структурного шаблона проекта. Проблемы, которые следует рассмотреть, включают взаимодействия между акторами и уровнем `presentation` (представление), логику управления системы и информационные потребности системы (БД). Вспомните, что акторы могут связываться с системой только через классы уровня `presentation`.

Концептуальные классы, которые определяют бизнес-объекты в БД, должны быть выделены заранее (раздел 8.3). Концептуальные классы станут в проекте классами пакета `entity` (сущность). Нет никакой явной потребности пересматривать их сейчас. Обратите также внимание на то, что большинство классов `entity` определяется не одним сценарием использования, а относится к системе предприятия в целом. Проблема пакета `entity` — и его обязанность *кэшировать* бизнес-объекты в памяти программы — будет позже переадресована (начиная со следующего шага определения классов из структурного проекта).

Систематический подход к определению классов из требований сценария использования может быть представлен в виде таблицы, чтобы перечислить требования и идентифицировать классы. Таблица может состоять из пяти столбцов, названных следующим образом:

- *Номер требования* — это последовательный номер, используемый для простоты ссылки. Если управление требованиями в проекте выполнено должным образом, тогда требования уже перечислены и размещены в CASE-хранилище проекта [61].
- *Определение требования* — это текст требования, содержащий ссылки на модели, диаграммы, рисунки, составляющие полное определение требования.
- *Пакет размещения и имя класса* — этот столбец «определяет класс» для требования. Он задает имя пакета и имя класса, ответственного за выполнение требования. Если имя класса имеет префикс в виде буквы, которая определяет его пакет, то указание имени пакета не обязательно. Альтернативно имена пакетов могут быть помещены в отдельный столбец.
- *Отличительное имя операции* — этот столбец содержит имена операций класса, ответственных за выполнение требования. Идентифицируются только основные операции. Вспомогательные операции, предназначенные для завершения услуг, будут определены во время проектирования взаимодействия (глава 12).
- *Сопутствующий пакет и класс или интерфейс* — этот столбец определяет главных партнеров основной операции, если операция не может сама обеспечить требуемую услугу. Столбец может содержать только имя сотрудничающего пакета, но часто также приводится класс или имя интерфейса.

11.1.1. Определение классов из требований сценария использования для управления электронной почтой

Таблица 11.1 помогает в определении классов из документа сценария использования для управления электронной почтой (раздел 8.2). Требования были скопированы из документа сценария использования в столбец «Определение требования». Так как требования EM для итерации 1 весьма просты, окончательные требования могут быть объединены в небольшое число классификаторов. В некоторых случаях достаточен единственный класс проекта на пакет для первоначальных итераций (это особенно верно для пакетов `control` — управление и `mediator` — посредник). Несомненно, шаги определения классов из структурных требований и из первого пробного проекта предложат большее количество классов и интерфейсов.

Требование R1 занимается сервисом `displayLogin` (отобразить регистрационное имя) класса `PConsole` (консоль). Сервис требует, чтобы пользователь ввел регистрационное имя, которое будет помещено в `getUserInput` (получить введенную пользователем информацию). Эта информация передается операции `login` (регистрационное имя) класса `CActioner` (класс «исполнитель» пакета `control`). Класс `FConnection` (класс «соединение» пакета `foundation`) уровня `foundation` (основание) выполняет операции, связанные с регистрационным именем.

Требование R2 использует операцию `getConnection` (получить соединение) класса `FConnection`. Операция `getConnection` использует интерфейс `IAConstants` (интерфейс «константы» пакета `acquaintance`), чтобы определить БД, к которой программа должна подключиться. `IAConstants` представляет все *постоянные величины*, используемые в приложении. Операция `readEmployee` (прочитать информацию о служащем) проверяет, является ли связываемый пользователь одним из служащих, которым позволено использовать подсистему EM (то есть, регистрационное имя служащего и адрес его электронной почты находятся в таблице `Employee` — служащий).

Вся ответственность за отображение списка меню согласно требованию R3 возлагается на операцию `displayMenu` (отобразить меню) класса `PConsole`. Если пользователь выбирает на любой стадии выход из приложения, то согласно требованию R4 запрос пользователя на выход читается операцией `getUserInput` (получить введенную пользователем информацию) класса `PConsole` и передается классу `CActioner`, чтобы выполнить операцию `exit` (выход). Операция `exit` требует взаимодействия с уровнем `foundation`.

Требование R5 подпотока *S1 — View Unsent Messages* (просмотр непосланных сообщений) — отражено визуально на рис. 8.5. Запрос пользователя показать непосланные сообщения в консольном окне перехватывается операцией `getUserInput` класса `PConsole` и передается к своей собственной операции по имени `viewMessages` (просмотр сообщений). `viewMessages` использует интерфейс `IAConstants`, чтобы получить информацию о максимальном числе сообщений, которые могут быть показаны одновременно. Затем происходит обращение к операции `retrieveMessages` (извлечь сообщения) класса `MBroker` (класс «посредник» пакета `mediator`). `MBroker`

делегирует эту задачу далее классу FReader (класс «чтение» пакета foundation), который обращается к БД с операциями readMessages (читать сообщения) и readContact (читать информацию о деловом партнере).

Подпоток *S2 — Display Text of a Message* (отображение текста сообщения) — показан как эквивалент требования R6. Он начинается с операции getUserInput класса PConsole. Вход пользователя, распознанный как запрос показать текст исходящего сообщения, передается к displayMessageText (отобразить текст сообщения) в том же самом классе (PConsole). Чтобы показать текст исходящего сообщения, приложение должно выяснить, находится ли исходящее сообщение в кэше памяти программы (как экземпляр EOutMessage — класс «исходящее сообщение» пакета entity) или оно должно быть извлечено из БД, используя класс FReader пакета foundation. Задача выяснения дальнейшего пути требует сотрудничества с классом MBroker. Поскольку она требует больше информации от структурного проектирования, исходя из требований сценария использования, ее объяснение оставлено до следующего шага определения классов, исходя из структурных требований.

Требования R7 и R8 связаны с передачей по электронной почте исходящих сообщений согласно подпотоку *S3 — Email a Message* (передача сообщений по электронной почте). Чтобы начать эти операции, getUserInput запрашивает сотрудничество операции prepareMessage (подготовить сообщение), которая в свою очередь передает работу операции sendMessage (послать сообщение) класса CActioner. С этого момента требуется использовать библиотеку JavaMail™ API (называемую на жаргоне как MAPI — Mail API — интерфейс API электронных сообщений), которая обеспечивает функциональные возможности создания и отсылки электронных сообщений (MAPI обсуждается в разделе 13.4.1). Операция sendMessage требует сотрудничества с интерфейсом IAConstants, чтобы получить адрес почтового сервера. Убедившись, что сообщение по электронной почте было послано успешно, операция updateMessage (скорректировать сообщение) класса FWriter (класс «запись» пакета foundation) корректирует таблицу OutMessage (исходящее сообщение) — рис. 8.8, задавая значение параметру date_emailed (данные отосланы).

Требование R8 включает подпоток S3. Обязанность отображения информационного сообщения относительно успешно посланного сообщения по электронной почте возложена на операцию displayConfirmation (отображение подтверждения) класса PConsole.

Оставшиеся требования относятся к потокам исключений. Поток *E1 — Incorrect username or password* (неправильное имя пользователя или неправильный пароль) представлен как требование R9. Он начинается с обязанностей, определенных в требовании R1. При неправильном вводе регистрационного имени операция displayLoginError (отобразить сообщение об ошибке ввода регистрационного имени) класса PConsole показывает информационное сообщение и позволяет пользователю повторить ввод до трех раз. Если пользователь неспособен правильно ввести регистрационное имя, операция exit класса CActioner обеспечивает изящную зачистку и выход.

Таблица 11.1. Определение классов из требований сценария использования

№ треб.	Определение требования	Пакет размещения и имя класса	Отличительное имя операции	Сопутствующий пакет и класс или интерфейс
R1	Система отображает информационное сообщение и запрашивает, чтобы служащий ввел имя пользователя и пароль	presentation PConsole control CActioner	displayLogin getUserInput login	presentation PConsole control CActioner foundation FConnection
R2	Система пытается связать служащего с БД EM	foundation FConnection foundation FReader	getConnection readEmployee	acquaintance IAConstants, foundation FReader
R3	После успешной связи приложение отображает список возможных опций меню, которые служащий может запросить	presentation PConsole	displayMenu	
R4	Если служащий выберет выход из EM-приложения, нажав 4, сценарий использования завершается	presentation PConsole control CActioner	getUserInput exit	control CActioner foundation FConnection
R5	*S1 — View Unsent Messages* (просмотр непосланных сообщений): информация отображается в консольном окне, как в примере на рис. 8.5	presentation PConsole mediator MBroker foundation FReader	getUserInput viewMessages retrieveMessages readMessages readContact	presentation PConsole acquaintance IAConstants, mediator MBroker foundation FReader
R6	*S2 — Display Text of a Message* (отображение текста сообщения): служащий подтверждает message_id, прежде чем будет отображен текст этого сообщения. Текст сообщения отображается так, как показано на рис. 8.6	presentation PConsole	getUserInput displayMessageText	presentation PConsole mediator MBroker
R7	*S3 — Email a Message* (передача сообщения по электронной почте): служащий задает message_id сообщение, которое следует передать по электронной почте. Служащий печатает message_id, сообщение отсылается и корректируется БД	presentation PConsole control CActioner foundation FWriter	getUserInput prepareMessage sendMessage updateMessage	presentation PConsole control CActioner MAPI, acquaintance IAConstants, foundation FWriter
R8	*S3 — Email a Message*: информационное сообщение отображается в консольном окне после успешной передачи сообщения, как показано на рис. 8.7	presentation PConsole	displayConfirmation	

Таблица 11.1. (продолжение)

№ треб.	Определение требования	Пакет размещения и имя класса	Отличительное имя операции	Сопутствующий пакет и класс или интерфейс
R9	*E1 — Incorrect username or password* (неправильное имя пользователя или неправильный пароль): если в основном потоке актор вводит неправильное имя пользователя или неправильный пароль (рис. 8.3), система выводит сообщение об ошибке. Система разрешает актору повторно ввести имя пользователя и пароль, либо покинуть приложение. Актору дают три возможности, чтобы ввести правильные имя пользователя и пароль. Если все три раза будут неудачны, система отменяет регистрацию и сценарий использования заканчивается	ref. R1 presentation PConsole control CActioner	 displayLoginError exit	presentation PConsole ref. R1, control CActioner foundation FConnection
R10	*E2 — Incorrect option* (неправильная опция): если в основном потоке актор выберет неправильную опцию (рис. 8.4), система игнорирует введенное значение и заново показывает список опций. Если будут подряд три неудачных выбора, система прекращает работу с актором и повторно начинает сценарий использования (запрашивая регистрационное имя — рис. 8.3)	presentation PConsole control CActioner ref. R1	getUserInput displayMenu logout	presentation PConsole foundation FConnection
R11	*E3 — Too many messages* (слишком много сообщений): если в подпотоке S1 число непосланных сообщений, запланированных актору, превышает заранее установленное число сообщений, которые могут рассматриваться одновременно, система отображает информационное сообщение о том, что в БД имеется большее количество сообщений, чем допустимо. Информационное сообщение отображается после того, как актору показывается заранее установленное число непосланных сообщений, и перед тем, как будет заново показан список меню.	ref. R5 presentation PConsole	tooManyMessages	
R12	*E4 — Email could not be sent* (сообщение электронной почты не может быть послано): если в подпотоке S3 почтовый сервер возвращает ошибку о том, что сообщение не могло быть послано, система выдает актору информацию, что сообщение не было послано, и сценарий использования продолжается, показывая список меню	ref. R7 control CActioner presentation PConsole	 handleEmailException displayEmailFailure displayMenu	 presentation PConsole

Ответственность за требование R10 (*E2 — Incorrect option* — неправильный выбор варианта) возложена на `PConsole`. Если операция `getUserInput` обнаруживает, что введенная опция меню неправильна, она просит операцию `displayMenu` заново показать список меню. Если три раза ввод будет неудачен, операция `logout` (завершение работы с зарегистрированным пользователем) класса `CActioner` отменяет регистрацию пользователя и обеспечивает возможность другой регистрации согласно требованию R1.

Поток исключений *E3 — Too many messages* (слишком много сообщений) представлен как требование R11. Он ограничивает число выводимых сообщений, которые могут быть показаны одновременно на экране. Это важно из-за ограниченного размера памяти программы и из-за требований удобства пользовательского интерфейса (см. FURPS в разделе 8.4). Операция `viewMessages` (см. выше R5) получает предельное число исходящих сообщений от интерфейса `IAConstants`. Если имеются еще не извлеченные из БД исходящие сообщения, `viewMessages` информирует операцию `tooManyMessages` (слишком много сообщений) класса `PConsole`, как обстоит дело. Операция `tooManyMessages` отображает информацию об этом.

Требование R12 определяется потоком исключений *E4 — Email could not be sent* (сообщение электронной почты не может быть послано). Если сообщение электронной почты не может быть послано по какой-либо причине, операция `sendMessage` класса `CActioner` (см. выше R7) просит операцию `handleEmailException` (управление исключениями электронной почты) объяснить проблему. Операция `displayEmailFailure` (отобразить ошибку электронной почты) класса `PConsole` в этом случае отображает соответствующую информацию. Последующая операция `displayMenu` позволяет продолжить сценарий использования.

11.1.2. Проектирование исходных классов для управления электронной почтой

Таблица 11.1 дает достаточно информации, чтобы попытаться изобразить исходную модель классов для управления электронной почтой. Поскольку таблица перечисляет ответственные операции, их можно показать в модели. Однако в таблице не предполагалось показать сценарий пошаговой передачи сообщений и все сотрудничающие операции. Исходная диаграмма классов может заполнить некоторые пустые места, но все же модель не будет полной.

Рис. 11.1 представляет наглядное отображение таблицы 11.1 в виде диаграммы классов. Здесь имеются некоторые дополнительные части информации, но там также можно наблюдать «пропущенные связи» из-за пробелов в таблице (в частности, отсутствуют многие операции). Здесь показаны ассоциации для нисходящей связи, однако модель не объясняет, как организована восходящая связь. Из-за простоты итерации 1 обязанности трех пакетов (`presentation` — представление, `control` — управление и `mediator` — посредник) сосредоточены в трех отдельных классах — по одному на каждый пакет. И все же проект классов сущностей — не подарок.

Константы в интерфейсе `IAConstants` (интерфейс «константы» пакета `acquaintance`) имеют имена. Они идентифицируют БД (используемую

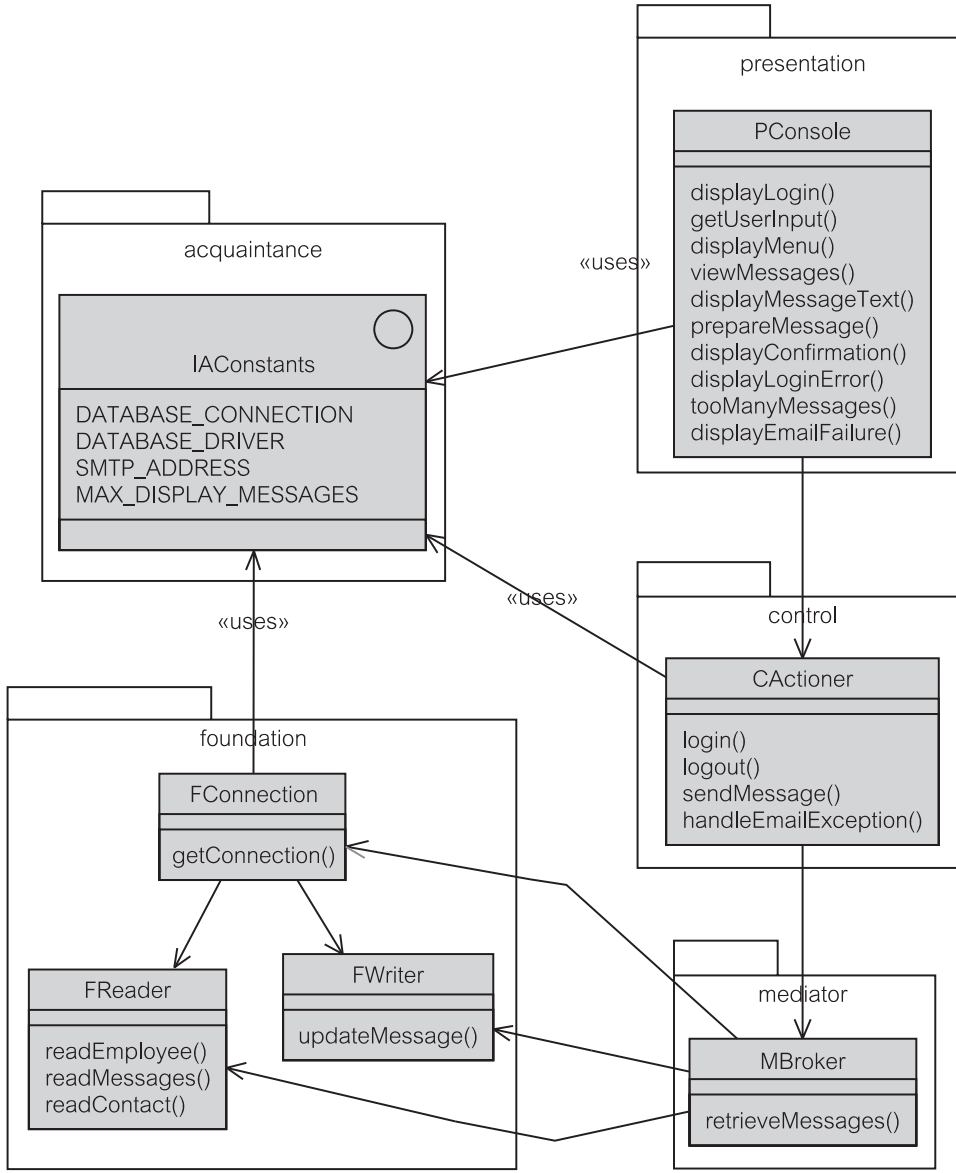


Рис. 11.1. Диаграмма классов управления электронной почтой

классом `FConnection` — класс «соединение» пакета `foundation`) и почтовый сервер (используемый классом `CActioner` — класс «исполнитель» пакета `control`), с которым соединяется приложение. Последняя константа сообщает приложению, каково максимальное число исходящих сообщений, которые могут быть показаны на экране (используется классом `PConsole` — класс «консоль» пакета `presentation`) — см. ниже.

Модель включает ассоциации между сотрудничающими классами. Согласно PCMEF-структуре все ассоциации являются нисходящими. Большинство ассоциаций могло бы быть выявлено из сотрудничества, непосредственно видимого в таблице 11.1. Однако в связи с тем, что таблица не показывает пошаговую передачу сообщений, некоторые виды сотрудничество (которые проанализированы в таблице), по-видимому, имеющиеся между несоседними пакетами, удалены из PCMEF-структуры. Практически все такие виды сотрудничества будут использовать ассоциации (как показано на рис. 11.1) и другие средства связи, допустимые в PCMEF-структуре.

Константы в интерфейсе

Интерфейс Java — удобный способ хранения *констант*. Элементы данных в интерфейсе Java, в противоположность UML-интерфейсу (раздел 9.1.7), неявно заданы как public (внешний), static (статический) и final (заключительный) [25]. Следовательно, они являются константами. Классы, которые используют интерфейс `IAConstants`, должны только импортировать интерфейс:

```
import acquaintance.IAConstants;
```

Величины констант в диаграмме не показываются, но, конечно же, каждая константа должна иметь значение, когда программа скомпилирована, и эта величина не может быть изменена. Поскольку интерфейс не имеет экземпляров, среда программы во время выполнения размещает постоянные величины в статической области хранения, находящейся в интерфейсе.

Следовательно, константы могут использоваться для *параметризации* операций программы, использующих параметры. Параметризация задается во время компиляции, а не во время выполнения, но это имеет то преимущество, которое ясно видно в исходном коде, что она ограничена одним интерфейсом в коде. Когда нужна более сложная параметризация, требующая определения во время выполнения, но, тем не менее, видимая в исходном коде, может использоваться одноэлементный класс. *Одноэлементный класс* может инициализировать только один объект.

11.2. Структурная разработка проекта классов

Классы (и интерфейсы) должны удовлетворить структурным ограничениям, накладываемым структурным шаблоном, выбранным для проекта. Ограничения включают **нефункциональные требования** (FURPS+), определенные в документе дополнительной спецификации (раздел 8.4).

Структурные ограничения задают требования к **разработке** исходного проекта классов. Разработка является процессом усовершенствования, основанным на структурных решениях. Разработка проекта классов — более узкая концепция, чем та, которая определена в **Unified Process** (UP) — объединенном процессе. В UP разработка занимается одной из четырех основных стадий разработки системы (глава 1). UP-стадии следующие [12, 53]:

1. начало;
2. разработка;

3. конструирование;
4. переход.

Разработка проекта на основе структурных требований завершается модификациями и дополнениями к классам, определенным из требований пользователя. Поэтому процесс может начаться с таблицы, используемой для определения классов, на основании требований сценария использования, и затем таблица должна быть расширена, чтобы включить столбцы, которые представляют модификации и дополнения. Новая таблица (таблица 11.2) может состоять из следующих столбцов:

- *Номер требования* — как в таблице 11.1.
- *Ответственный класс и операция* — этот столбец объединяет столбцы 2 и 3 из таблицы 11.1.
- *Сотрудничающий пакет, класс или интерфейс* — этот столбец первоначально тот же самый, что и последний столбец в таблице 11.1. Однако если изменения сделаны в строке именно в этом столбце, то это может изменить рассматриваемый столбец по сравнению с таблицей 11.1.
- *Структурный принцип и/или паттерн и/или другая причина изменения* — этот столбец поясняется в главе 9 и в особенности в разделах 9.2 и 9.3, где определены принципы и паттерны PCMEF-шаблона.
- *Новый/обновленный ответственный класс и операция* — этот столбец перечисляет новые или скорректированные классы и операции, чтобы выполнить или завершить обязанности, не полностью решенные на шаге определения классов из требований сценариев использования.
- *Новый/обновленный сотрудничающий пакет, класс или интерфейс* — этот столбец определяет сотрудников для новых представленных или скорректированных классов и операций.

11.2.1. Структурная разработка проекта классов для управления электронной почтой

Таблица 11.2 представляет проект классов для управления электронной почтой после структурной проработки. Первые три столбца повторяют результаты таблицы 11.1. Оставшиеся три столбца — результат разработки. Разработка привела к добавлению классов пакета `entity` (сущность), расширению пакета `acquaintance` (знакомство) и дополнению методов к классам во всех PCMEF-пакетах, кроме классов уровня `foundation` (основание).

Класс `MBroker` (класс «посредник» пакета `mediator`) получил операцию `login` (регистрационное имя) как результат разработки требования R1. Это происходит из-за того, что PCMEF-шаблон запрещает прямое сотрудничество между уровнем управления — `control` и уровнем основания — `foundation` (из-за принципа нисходящей зависимости и паттерна Цепочки обязанностей). Следовательно, `CActioner` (класс «исполнитель» пакета `control`) делегирует регистрационное имя к `MBroker`.

Требование R2 используется, когда операция `login` класса `MBroker` вызывает операцию `connect` (связь) объекта `FConnection` (класс «соединение» пакета `foundation`). Операция `connect` посылает сообщение

Таблица 11.2. Структурные уточнения проекта классов

№ треб.	Ответственный класс и операция	Сотрудничающий пакет и класс или интерфейс	Структурный принцип и/или паттерн и/или другая причина изменения	Новый/обновленный ответственный класс и операция	Новый/обновленный сотрудничающий пакет и класс или интерфейс
R1	PConsole displayLogin PConsole getUserInput CActioner login	presentation PConsole control CActioner mediator MBroker	DDP и Chain из Responsibility	MBroker login	foundation FConnection
R2	FConnection connect	acquaintance IAConstants, foundation FReader			
	FReader readEmployee		employee, извлеченный из БД	MBroker createEmployee	entity EEmployee
R3	PConsole displayMenu				
R4	PConsole getUserInput CActioner exit	control CActioner mediator MBroker	DDP и Chain из Responsibility	MBroker logout	foundation FConnection
R5	PConsole getUserInput PConsole viewMessages	presentation PConsole acquaintance IAConstants	DDP и Chain из Responsibility, когда инициализируются объекты EOutMessage и EContact (см. последнюю строку R5); NCP и APP	CActioner retrieveMessages PConsole displayMessages	mediator MBroker acquaintance IAEmployee IAOutMessage IAContact, EEmployee EOutMessage EContact, entity
	MBroker retrieveMessages		детали партнера нужно извлечь вместе с сообщениями	MBroker retrieveContacts	foundation FReader
	FReader readMessages FReader readContact		сообщения и партнеры извлекаются из БД	MBroker createContacts MBroker createMessages	entity EContact entity EOutMessage
R6	PConsole getUserInput PConsole displayMessageText	presentation PConsole	DDP и Chain из Responsibility, чтобы решить, где искать сообщения	CActioner retrieveMessage MBroker isInCache	mediator MBroker entity EOutMessage foundation FReader

Таблица 11.2. (продолжение)

№ треб.	Ответственный класс и операция	Сотрудничающий пакет и класс или интерфейс	Структурный принцип и/или паттерн и/или другая причина изменения	Новый/обновленный ответственный класс и операция	Новый/обновленный сотрудничающий пакет и класс или интерфейс
			Если экземпляр EOutMessage в кэше; NCP и APP	PConsole displayMessageText	acquaintance IAEmployee IAOutMessage IAContact, entity EEmployee EOutMessage EContact
R7	PConsole getUserInput PConsole prepareMessage CActioner sendMessage FWriter updateMessage	presentation PConsole control CActioner MAPI, acquaintance IAConstants, foundation FWriter	DDP и Chain из Responsibility Mediator; установить в 1 флаг кэша	MBroker updateMessage MBroker flagCache	foundation FWriter
R8	PConsole displayConfirmation				
R9	см. R1 PConsole displayLoginError CActioner exit	presentation PConsole см. R1, control CActioner	DDP и Chain из Responsibility	MBroker logout	foundation FConnection
R10	PConsole getUserInput PConsole displayMenu CActioner logout см. R1	presentation PConsole	DDP и Chain из Responsibility	MBroker logout	foundation FConnection
R11	см. R5 PConsole tooManyMessages				
R12	см. R7 CActioner handleEmailException PConsole displayEmailFailure PConsole displayMenu	presentation PConsole			

readEmployee (прочитать информацию о служащем) классу FReader (класс «чтение» пакета foundation). Если пользователь ЕМ-приложения определен в БД как авторизованный служащий, его характеристики возвращаются классу MBroker. MBroker использует операцию createEmployee (создание объекта «служащий»), чтобы инициализировать объект EEmployee (класс «служащий» пакета entity) в кэше памяти программы.

Требование R3 не приводит к каким-либо изменениям из-за структурной проработки. Однако MBroker обращается к операции exit (выход) в требовании R4 на пути от CActioner к FConnection.

Совершенствование требования R5 (*View Unsent Messages — просмотр непосланных сообщений*) существенно. Принцип нисходящей зависимости и паттерн Цепочка обязанностей приводят к передаче операции retrieveMessages (извлечь сообщения) в класс CActioner. CActioner делегирует эту операцию классу MBroker. MBroker распознает, что детали рассматриваемого делового партнера должны быть извлечены вместе с исходящими сообщениями. С этой целью он использует операцию retrieveContact (извлечь делового партнера), а она делегирует работу по извлечению исходящих сообщений и деталей делового партнера операциям readMessages (читать сообщения) и readContact (читать делового партнера) класса FReader.

Исходящие сообщения и деловые партнеры возвращаются классу MBroker, который затем последовательно использует операции createContact (создать делового партнера) и createMessages (создать сообщения), чтобы инициализировать объекты EContact (класс «деловой партнер» пакета entity) и EOutMessage (класс «исходящее сообщение» пакета entity). MBroker обеспечивает также то, что во время формирования экземпляров EOutMessage будут созданы связи ассоциации между объектами EOutMessage и EContact.

В этот момент операция viewMessages (просмотр сообщений) класса PConsole (класс «консоль» пакета presentation) может вызвать операцию displayMessages (отобразить сообщения), чтобы получить детали каждого исходящего сообщения и связанного с ним делового партнера и представить эту информацию на экране компьютера. Согласно принципу соседней связи и принципу использования пакета знакомств, операция displayMessages использует соответствующие интерфейсы в пакете знакомств, чтобы получить доступ к объектам EOutMessage и EContact, которые реализуют эти интерфейсы.

Проект классов в соответствии с требованием R6 (*Display Message Text — отображение текста сообщения*) также экстенсивно дополняется в процессе структурной разработки. Во-первых, операция displayMessageText (отобразить текст сообщения) класса PConsole направляется через операцию retrieveMessage (извлечь сообщение) класса CActioner к объекту класса MBroker. MBroker может получить текст исходящего сообщения от соответствующего объекта класса EOutMessage, если этот объект находится в кэше памяти и он не «изменен» (то есть, его информационное содержание то же самое, что и в БД). В противном случае MBroker должен использовать FReader, чтобы поместить объект класса EOutMessage в кэш. Опе-

рация isInCache (наличие в кэше) класса MBroker решает, каким путем нужно идти.

Если экземпляр EOutMessage находится в кэше, операция displayMessageText класса PConsole может обратиться к этому экземпляру через интерфейс IAOutMessage, который реализует класс EOutMessage. Практически все остальные интерфейсы пакета entity — IAEmployee (интерфейс «служащий» пакета acquaintance) и IAContact (интерфейс «деловой партнер» пакета acquaintance) — должны также использоваться, потому что доступ к EOutMessage следует связям ассоциации от объектов классов EEmployee и EContact к объекту класса EOutMessage.

Требование R7 (*Email Message — передача сообщения по электронной почте*) дополнено двумя операциями в классе MBroker: updateMessage (скорректировать сообщение) и flagCache (установка флага кэша). Операция updateMessage обеспечивает принцип нисходящей зависимости и патерн Цепочка обязанностей. Как только операция updateMessage класса FWriter (класс «запись» пакета foundation) скорректирует таблицу OutMessage (исходящее сообщение), MBroker использует операцию flagCache, чтобы «изменить» флаг кэша. Однако детали того, как это сделано, все еще не решены. Особенностью данной операции является то, что объекты в пакете entity станут «измененными», как только исходящее сообщение будет передано по электронной почте, и БД обновляется. Все это известно программе.

На требование R8, которое завершает работу, начатую в соответствии с требованием R7, структурная проработка не влияет. Точно так же структурная проработка не вызвала существенных изменений в проекте классов для требований R9–R12, которые определяют потоки исключений сценария использования. Только операция logout (завершение работы с зарегистрированным пользователем) класса MBroker используется в требованиях R9 и R10, чтобы позволить классу CActioner делегировать операцию logout классу FConnection (класс «соединение» пакета foundation).

11.2.2. Проектирование классов для управления электронной почтой после структурной проработки

Результаты таблицы 11.2 приводят к созданию улучшенной модели классов для управления электронной почтой, как показано на рис. 11.2. К модели добавлен пакет entity (сущность). Пакеты entity и mediator (посредник) составляют пакет domain (предметная область) как один из PCMEF-уровней. Пакет acquaintance (знакомство) имеет три новых интерфейса, реализованные классами в пакете entity. Новые операции, определенные во время структурной проработки, показаны в соответствующих классах.

11.2.3. Инициализация классов

Важным побочным эффектом структурной проработки является определение того, кто (то есть, какой класс) должен быть ответственным за создание нового экземпляра другого класса. Другими словами, кто должен послать сообще-

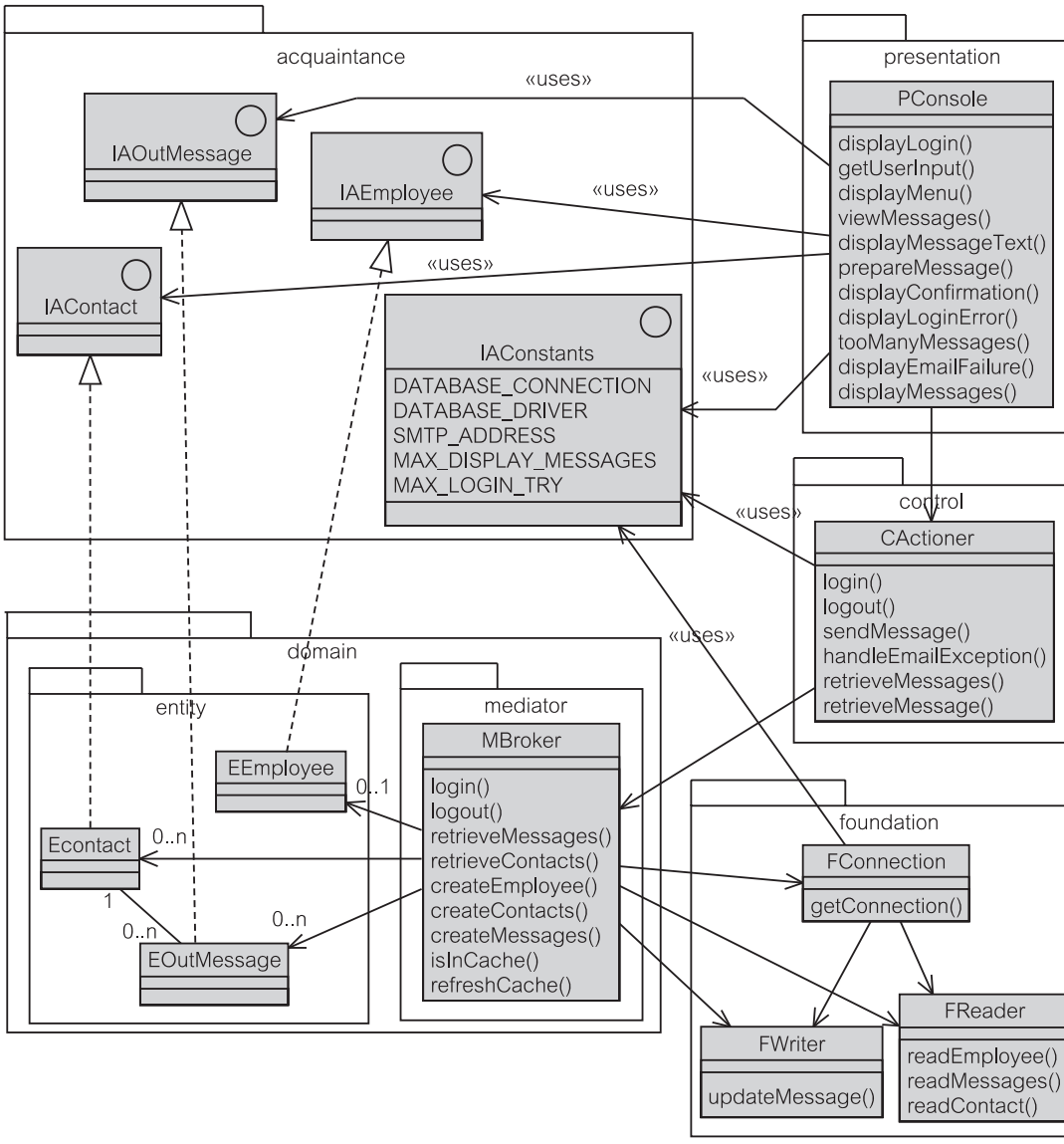


Рис. 11.2. Диаграмма классов для управления электронной почтой

ние методу-конструктору другого класса, чтобы инициализировать новый объект этого класса.

Некоторые классы могут быть инициализированы при запуске программы каким-то методом инициализации. Другие классы создаются динамически во время выполнения, однако их создатели известны статически (во время компиляции). В более сложных случаях точный создатель объекта определяется во время выполнения. Однако для большей части объектов в программе их созда-

тели могут быть известны во время проектирования. Это означает, что инициализация экземпляра класса может быть тщательно отработана, и структурные соображения могут быть в центре внимания этих проектных усилий.

Инициализация класса должна твердо придерживаться принципов структурного шаблона, выбранного для проекта. Инициализация может включать разные структурные уровни, пока обеспечивается принцип нисходящей зависимости PCMEF-шаблона. Это означает, что классы в более высоких уровнях инициализируют классы в (соседних) более низких уровнях, но не наоборот.

Кто инициализирует первый объект?

Объектно-ориентированная система во время своего выполнения походит на футбольный матч. Игроки передают мяч подобно классам, которые передают сообщения. Игрок решает, кому передать мяч «во время выполнения». Обязанности распределены среди игроков, и никто из игроков не может управлять игрой. Точно так же никакой объект не управляет ходом игры в объектно-ориентированной программе.

Лишь единственная «передача мяча», которой можно управлять с некоторой степенью уверенности, — в начале матча. Футбольный матч имеет начало, и объектно-ориентированная программа должна каким-то образом начаться. Первый объект должен быть инициализирован в начале программы. Начальная точка компьютерной программы находится в **главном методе**. Главный метод инициализирует первый объект — это объект класса, который содержит главный метод.

В управлении электронной почтой класс, содержащий метод `main` (главный), находится в пакете `presentation` (представление) и называется `PMain` (класс «основной» пакета `presentation`). Java требует массив объектов `String` (строка) в качестве аргумента метода `main` (рис. 11.3). Параметр `args` (аргументы) содержит аргументы, если таковые вообще имеются, помещаемые в командную строку при запуске программы (в итерации 1 не используется). В Java класс, который содержит главный метод, должен иметь то же самое имя, что и запускаемый программный файл. Метод-конструктор, инициализирующий объект `PMain`, то есть `PMain()`, обеспечивает второй «удар ногой», инициализируя один или несколько других классов и передавая «мяч» одному из этих только что созданных объектов.

Диаграмма инициализации для управления электронной почтой

Технология создания диаграмм классов UML может использоваться, чтобы показать, кто создает новые экземпляры других классов. Этого можно достигнуть, рисуя отношения зависимости между классами со стереотипом **«instantiate»** (инициализация).

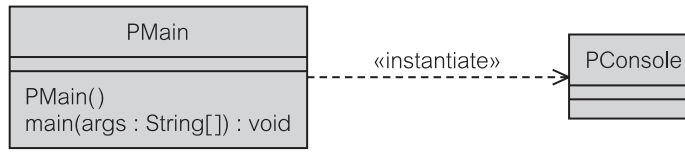


Рис. 11.3. Класс PMain в EM

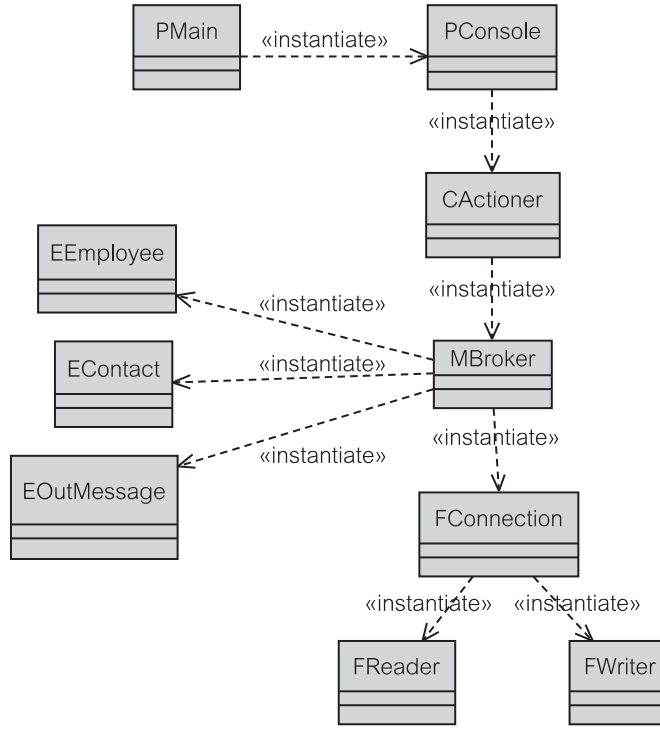


Рис. 11.4. Диаграмма инициализации для EM

Когда класс `A` инициализирует класс `B`, объекту `A` необходима ссылка на объект `B`. Если эта ссылка в будущем должна использоваться для передачи сообщений от `A` до `B` (как часто и бывает), то должна быть установлена однонаправленная связь ассоциации.

Рис. 11.4 изображает диаграмму инициализации для управления электронной почтой. Связи ассоциации на рис. 11.4 не смоделированы. Они показаны на рис. 11.2. Обратите внимание, что нет никакой необходимости связать `PMain` и `PConsole` (класс «консоль» пакета `presentation`), потому что управление уже не возвращается к `PMain` после выхода из него в результате «удара».

11.3. Взаимодействия

UML 2.0 определяет **взаимодействие** как «единицу поведения, которое сосредоточено на наблюдаемом обмене информацией между отдельными составными частями» [106]. «**Составная часть** представляет множество экземпляров, которые принадлежат экземпляру, содержащему классификатор» [106]. В действительности же составная часть — это объект, который существует во время взаимодействия. Существование объекта в определенное время называется **линией жизни**.

Взаимодействие реализуется как последовательность сообщений между линиями жизни. Сообщения могут быть синхронными (раздел 9.1.6) или асинхронными (раздел 9.1.8). **Сообщение** представляет собой связь от *возникновения посылающего события* на одной линии жизни до *возникновения получаемого события* на другой линии жизни.

UML 2.0 обеспечивает ряд графических технологий изображения взаимодействия. Технологии включают диаграммы последовательности действий, диаграммы связей (до UML 2.0 они назывались диаграммами сотрудничества) и диаграммы просмотра взаимодействий.

11.3.1. Диаграммы последовательности действий

Диаграмма последовательности действий определяет последовательность сообщений и возникновения их событий на линиях жизни. Так как линии жизни представляют составные части, а последние представляют объекты, диаграмма последовательности действий показывает обмен сообщениями между объектами в процессе взаимодействия. Область взаимодействия отдается на усмотрение проектировщика; это может быть сценарий использования, часть его, ряд сценариев использования, отдельное требование, действие пользователя и т. д.

Прямоугольная рамка с названием в отделении, расположенном в верхнем левом углу, изображает **взаимодействие** (рис. 11.5). Пункты на рамке могут служить **шлюзами**, чтобы обозначить начало или конец сообщений. Шлюзы представляют концептуальный интерфейс взаимодействия. Они могут моделировать события от/к актора(у). Шлюз на рис. 11.5 сообщает нам, что взаимодействие начинается, когда вызывается сообщение `doit()` (выполнить) на линии жизни `:Client` (клиент).

Сообщения изображаются в виде горизонтальных стрелок между линиями жизни. Имя составной части для линии жизни содержит три элемента: `partName:ClassName[multiplicity]` (имя составной части: имя класса [множественность]). Множественность может использоваться, чтобы представить коллекцию объектов. Если множественность не указана, она считается равной единице.

Имя `partName` или имя `ClassName` может быть опущено. Если оба имени присутствуют, объект (`partName`) является конкретным экземпляром класса (`ClassName`). Если `partName` отсутствует, объект инициализируется и представляет любой или все экземпляры класса `ClassName`. Если имя `ClassName` опущено, объект не инициализируется.

Акцент в диаграмме последовательности действий делается на последовательность сообщений. Размещение сообщений одно под другим показывает это. Необязательная нумерация сообщений также указывает последовательность. Объект, получающий сообщение, активизирует соответствующий метод. Время, когда поток управления сосредоточен на объекте, называется **активизацией**. Активизации показываются в виде узких прямоугольников на линиях жизни объектов.

Иерархическая нумерация сообщений показывает зависимости активизации между сообщениями. В частности, как показано на рис. 11.5, сообщение к самому себе (то есть, метод обращается к тому же самому объекту) внутри

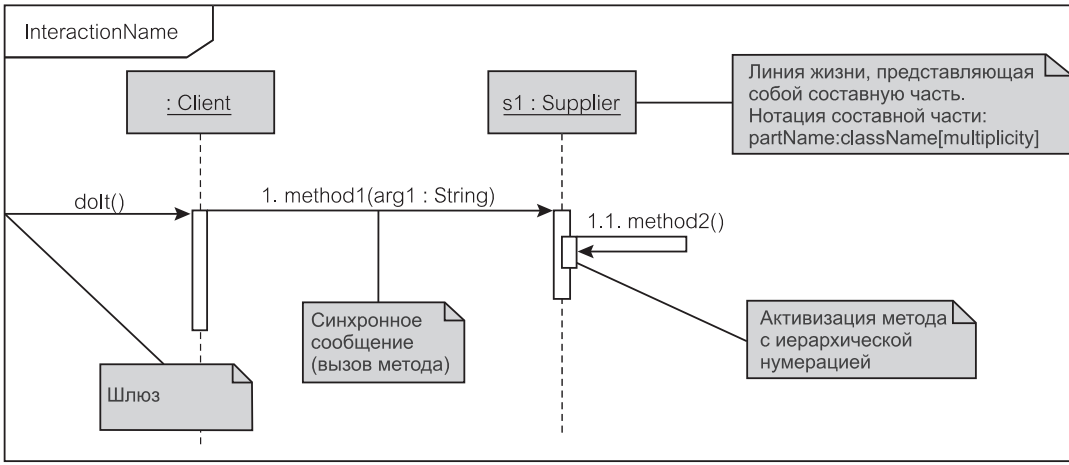


Рис.11.5. Диаграмма последовательности действий — нотация

одной активизации приводит к новой активизации. Это показано и иерархической нумерацией, и графически добавлением нового узкого прямоугольника. Как показано на рис. 11.6, UML 2.0 делает различия между следующими сообщениями:

- **синхронное сообщение** — представляется закрашенной стрелкой;
- **сообщение обратной связи** — представляется пунктирной линией с незакрашенной стрелкой; в UML 2.0 оно называется **откликом на метод** (чтобы не путать с **возвратом** от вызова сообщения, который реализуется неявно в конце активизации и обычно не показывается на диаграммах взаимодействия);
- **асинхронное сообщение** — представляется незакрашенной стрелкой;
- **сообщение создания объекта** — представляется пунктирной линией с незакрашенной стрелкой, направленной к прямоугольнику объекта (составной части).

Взаимодействия сосредотачиваются на последовательностях сообщений, а не на данных, которые сообщения передают. Поэтому сигнатуры сообщений не нужно показывать и типы возвращаемых величин обычно не визуализируются. Типы возвращаемых величин часто понимаются из контекста.

Говорят, что неспособность диаграмм взаимодействия представить манипуляции над данными может препятствовать пониманию более сложных взаимодействий, которые передают объекты (или коллекции объектов), и что можно использовать связи ассоциации для получения этих объектов. Например, считая, что на рис. 11.5 объект `s1` содержит ссылку на объект `x1` некоторого другого класса, сообщение от `:client` (клиент) к `s1`, требующее, чтобы `s1` возвратил `x1`, не приведет к какому-либо сообщению, реализующему связь от `s1` до `x1` (см. раздел 11.4.3 для более конкретного примера).

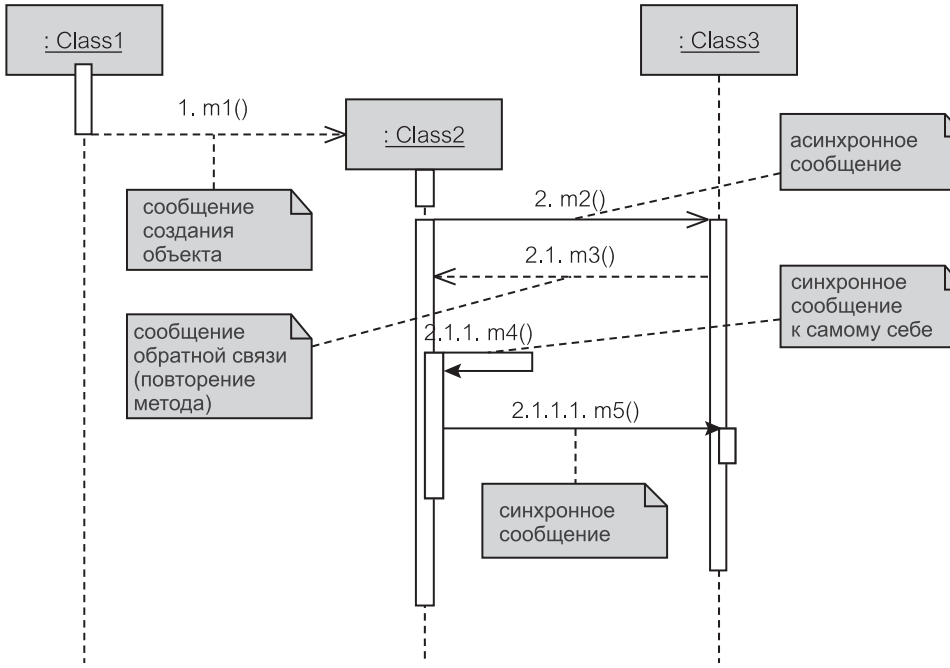


Рис. 11.6. Сообщения в диаграмме последовательности действий

11.3.2. Диаграммы связей

Диаграммы связей до UML 2.0 были известны как диаграммы сотрудничества. Изменение названия произошло потому, что в UML понятие сотрудничества означает нечто большее, чем семантика, формируемая диаграммой связей.

Сотрудничество «описывает, как операция или классификатор реализованы набором классификаторов, используемых и связанных определенным способом» [106]. Определение сотрудничества имеет два аспекта: структурный и поведенческий [61]. UML 2.0 предоставляет новую диаграмму, называемую **диаграммой внутренней структуры**. Диаграмма внутренней структуры рассматривает сотрудничество как своего рода классификатор и представляет структуру и поведение этого классификатора.

Диаграммы внутренней структуры и диаграммы взаимодействия задают концепцию составных частей. Диаграмма внутренней структуры показывает, как составные части обеспечивают поведение классификатора, который содержит их. Эта цель диаграммы внутренней структуры не отличается от основной цели диаграмм взаимодействия.

Если не касаться терминологии, диаграмма связей — просто визуальная разновидность диаграммы последовательности действий. Большинство CASE-средств может свободно преобразовывать эти две диаграммы друг в

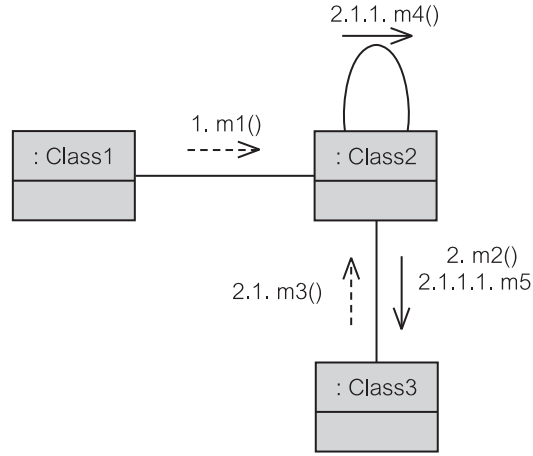


Рис. 11.7. Сообщения в диаграмме связей

друга. Рис. 11.7 представляет диаграмму связей, которая соответствует диаграмме последовательности действий на рис. 11.6 с удаленными примечаниями.

Сообщения в диаграмме связей изображаются непрерывной линией со стрелкой и именем сообщения. Непрерывная линия также может использоваться, чтобы показать, как объект, посылающий сообщение, «протягивает руку» объекту-получателю (это может быть, например, связью ассоциации или знакомством). Текущая версия UML 2.0 не реализует эту возможность, но многие CASE-средства делают это.

Диаграмма связей не изображает линии жизни (но сами объекты — составные части — представляют линии жизни). Активизации неявно проявляются в иерархической нумерации сообщений. Сообщение создания объекта также неявно (до некоторой степени) проявляется в нумерации — вновь созданный объект продолжает нумерацию со следующей последовательной величины. Кроме этого нет никакого внешнего различия между сообщениями создания объекта и обратной связи.

На практике разработчики используют диаграммы последовательности действий намного чаще, чем диаграммы связей. С точки зрения наглядного представления диаграммы последовательности действий имеют преимущество для более сложных моделей, в которых явная визуализация последовательностей сообщений является существенной даже при том, что они могут потребовать печати на большом формате бумаги, используя принтеры (графопостроители), способные обеспечить большие форматы.

Диаграммы связей могут быть более полезны для анализа сообщений от/к конкретного(му) объекта(у). Они могут быть более удобны при изображении исходного проекта взаимодействий и выполнения «итеративного» моделирования методом проб и ошибок. По этой причине они весьма удобны для организации «мозгового штурма».

11.3.3. Диаграммы просмотра взаимодействий

Взаимодействия могут содержать более мелкие взаимодействия, называемые **фрагментами взаимодействия**. Фрагмент взаимодействия — семантическое понятие, которое использует ту же самую синтаксическую нотацию, что и само взаимодействие — рамку с именем в отделении, расположенном в верхнем левом углу.

Взаимодействие, которое было полностью определено (возможно, диаграммой взаимодействия), называется **экземпляром взаимодействия**. Экземпляры взаимодействия имеют свои разрешенные шлюзы, и поэтому они могут быть объединены, чтобы показать поток управления между ними и в пределах больших взаимодействий. Эта возможность применяется в UML 2.0 в верхней части диаграмм деятельности (однако в другой UML-технологии моделирования), чтобы представить диаграмму просмотра взаимодействий.

Рис. 11.8 показывает, как диаграмма последовательности действий может ссылаться на взаимодействия и экземпляры взаимодействий. Эта диаграмма является расширением диаграммы, представленной на рис. 11.5. Взаимодействие `inter1` использует функциональные возможности экземпляра взаимодействия `InterOcc1` для регистрации БД. Затем это взаимодействие выполняет свою собственную работу, прежде чем выполнить объединение сообщений, называемых `method3()` и `InterOcc2`, в `Inter2`. `Inter2` полностью содержится в `Inter1`. `Inter2` посылает сообщение `exit` (выход) экземпляру взаимодействий `InterOcc2`, который завершается отменой регистрации в БД. После отмены регистрации шлюз `Inter1` посылает сообщение `exit` объекту `:Client` (клиент).

Диаграмма просмотра взаимодействий является ограниченной разновидностью **диаграммы деятельности** [61], в которой узлы-объекты заменены взаимодействиями и экземплярами взаимодействий. Рис. 11.9 представляет простой пример. Переход из начального состояния (черный круг) «инициализирует» `InterOccl`. При выходе из `interOccl` принимается решение, закончить ли взаимодействие, либо, если удовлетворяется ограничение защиты, начать `InterOcc2`.

11.4. Взаимодействия для управления электронной почтой

Идентификация взаимодействий для ЕМ весьма обычна. Итерация 1 ЕМ является консольным, управляемым с помощью меню приложением с ограниченным числом действий, которые может выбрать пользователь. Поэтому самый простой способ идентифицировать взаимодействия — связать взаимодействие с каждым основным действием, которое может выбрать пользователь. Они являются «удачными путями» взаимодействия (раздел 8.2). После этого могут быть добавлены «неудачные пути» взаимодействий, соответствующие потокам исключений в спецификации сценария использования.

Нет никакой необходимости создавать диаграмму просмотра взаимодействий. Список взаимодействий (это показано ниже) будет содержать:

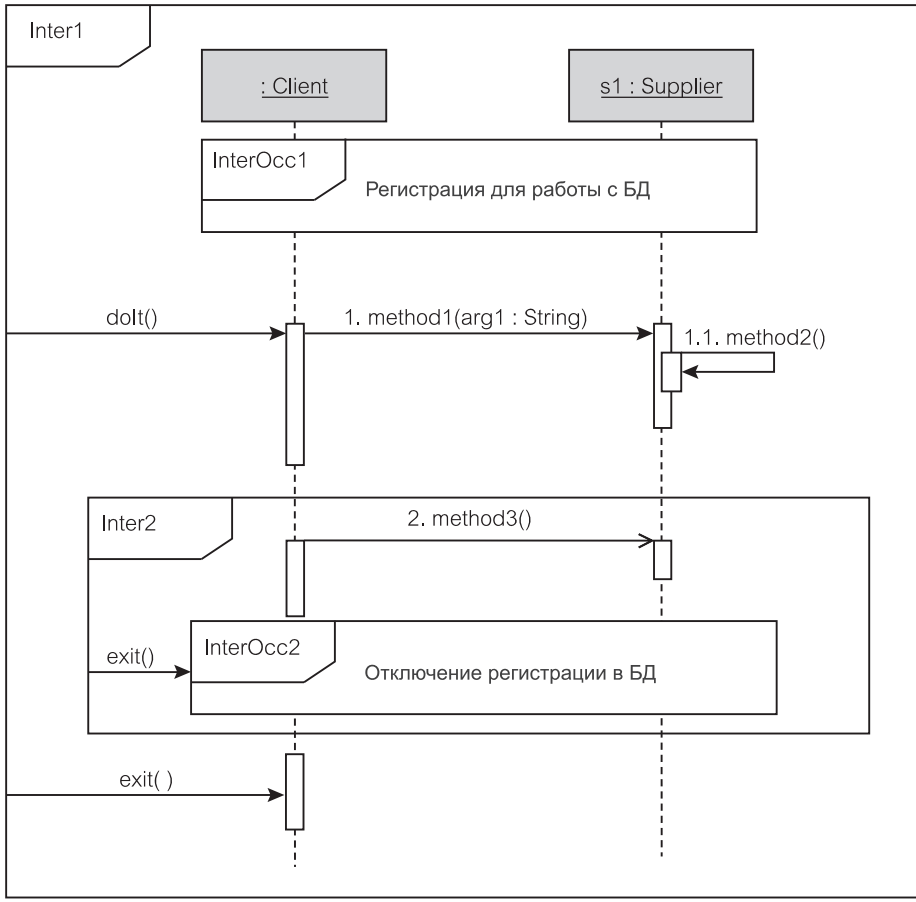


Рис. 11.8. Диаграмма последовательности действий с взаимодействиями и экземплярами взаимодействий

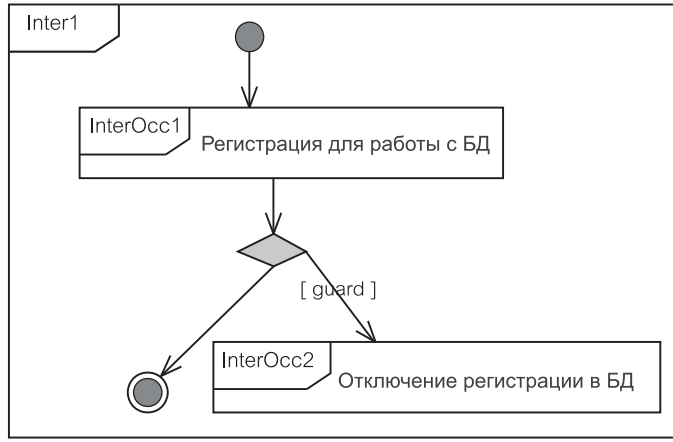


Рис. 11.9. Диаграмма просмотра взаимодействий

- «удачный путь» взаимодействия:
  - регистрационное имя (login);
  - выход (exit);
  - просмотр непосланных сообщений (view unsent messages);
  - отображение текста сообщения (display message text);
  - сообщение, передаваемое по электронной почте (email message);
- «неудачный путь» взаимодействия:
  - неправильное имя пользователя или неправильный пароль (incorrect username or password);
  - неправильная опция (incorrect option);
  - слишком много сообщений (too many messages);
  - сообщение не может быть послано по электронной почте (email could not be sent).

Ниже будут представлены диаграммы взаимодействия для EM. «Удачные пути» взаимодействий представлены как диаграммы последовательности действий. Они представляют различные экземпляры взаимодействий. Эти экземпляры взаимодействий ссылаются на взаимодействия «неудачных путей», которые смоделированы как диаграммы связей.

В соответствии с хорошей практикой моделирования законченность моделей стоит на втором месте после абстрактных требований. Диаграммы выражают важные моменты и опускают менее значимые решения моделирования, если они могут затенить модели. Некоторые из опущенных аспектов тривиальны или очевидны, однако другие могут быть достаточно хитрыми. Последние переадресованы непосредственному программированию и рефакторингу (глава 12).

Диаграммы в значительной степени очевидны, если они проанализированы в контексте предыдущего обсуждения этой главы. Это «удачные пути» взаимодействий, для которых применение потоков исключений представляет фрагменты взаимодействий в диаграммах, относящихся к «неудачным путям» взаимодействий. Однако границы взаимодействий (рис. 11.8) не представлены, чтобы сэкономить место.

В существенной степени диаграммы являются визуальным представлением сообщений, показанных в таблице 11.2. Сигнатуры сообщений дают дополнительное объяснение. В большинстве случаев предполагается формирование объектов, как это было обсуждено ранее (рис. 11.4).

11.4.1. Взаимодействие «Регистрационное имя»

Взаимодействие «Регистрационное имя» (Login) начинается с операции `displayLogin()` (отобразить регистрационное имя) от объекта `:PMain` (класс «основной» пакета `presentation`) к объекту `:PConsole` (класс «консоль» пакета `presentation`) — см. рис. 11.10. Это сообщение является частью метода-конструктора класса `PMain()`, который инициализирует объект класса `PConsole` перед тем, как к нему будет послана операция `displayLogin()` (рис. 11.3). Операция `getUserInput()` (получить вве-

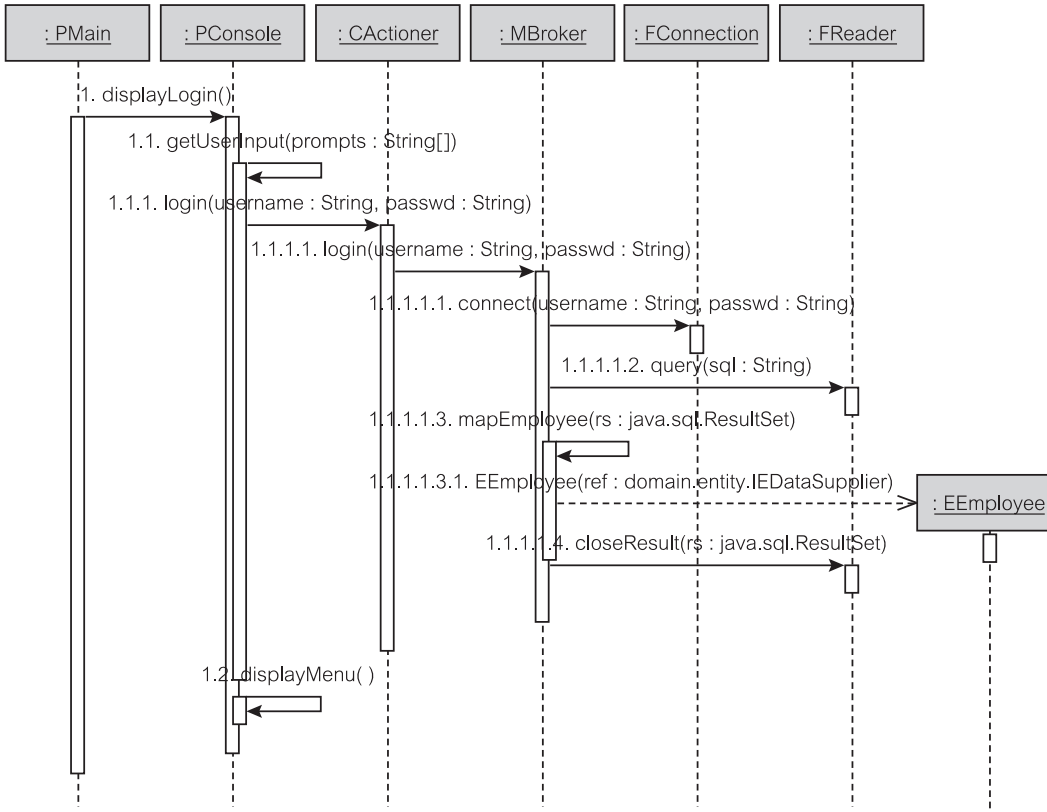


Рис. 11.10. Взаимодействие «Регистрационное имя»

денную пользователем информацию) объекта `:PConsole` получает регистрационное имя и пароль от пользователя и посылает запрос `login()` (регистрационное имя) к объекту `:CActioner` (класс «исполнитель» пакета `control`).

Объект `:CActioner` делегирует запрос `login()` объекту `:MBroker` (класс «посредник» пакета `mediator`), который в свою очередь делегирует его в сообщении `connect()` объекту `:FConnection` (класс «соединение» пакета `foundation`). Объект `:FConnection` инициализирует объект `:FReader` — класс «чтение» пакета `foundation` (здесь не показано), и два объекта связываются в этом процессе.

Объект `:MBroker` формирует SQL-строку и передает ее в сообщении `query()` (запрос) объекту `:FReader`. Объект `:FReader` выполняет SQL-поиск в БД, посылая следующий оператор `select` (выбор) через свое соединение с БД (на практике звездочка (`*`) предназначена для замены имен всех столбцов таблицы `employee` — служащий):

```
select * from employee where login_name = ?;
```

Операция `loginName` (идентификационное имя) обеспечивает однозначную идентификацию служащего в БД. Объект :`FReader` возвращает набор результатов объекту :`MBroker`. Объект :`MBroker` использует набор результатов, чтобы поместить строковые данные о служащих в только что созданный объект :`EEmployee`. Затем он запрашивает объект :`FReader`, чтобы тот закрыл набор результатов.

В результате у :`EEmployee` будут заполнены значениями все его элементы данных (здесь не показано). Взаимодействие «Регистрационное имя» после этого завершается, и :`PConsole` использует `displayMenu()` — отобразить меню, чтобы показать меню приложения пользователю.

11.4.2. Взаимодействие «Выход»

Взаимодействие «Выход» (Exit) — простая последовательность делегирований от объекта :`PConsole` (класс «консоль» пакета `presentation`) до объекта :`FConnection` (класс «соединение» пакета `foundation`) — см. рис. 11.11. Как только объект :`FConnection` закроет связь с БД, объект :`CActioner` (класс «исполнитель» пакета `control`) обеспечивает выход из системы.

11.4.3. Взаимодействие «Просмотр непосланных сообщений»

Взаимодействие «Просмотр непосланных сообщений» (View Unsent Messages) начинает с запроса пользователя из объекта :`PConsole` (класс «консоль» пакета `presentation`) — см. рис. 11.12. Операция `viewMessages()` (просмотр сообщений) создает новый (пустой) объект Java-коллекции (например, `ArrayList` — список-массив) и передает его в операции `retrieveMessages()` (извлечь сообщения) объекту

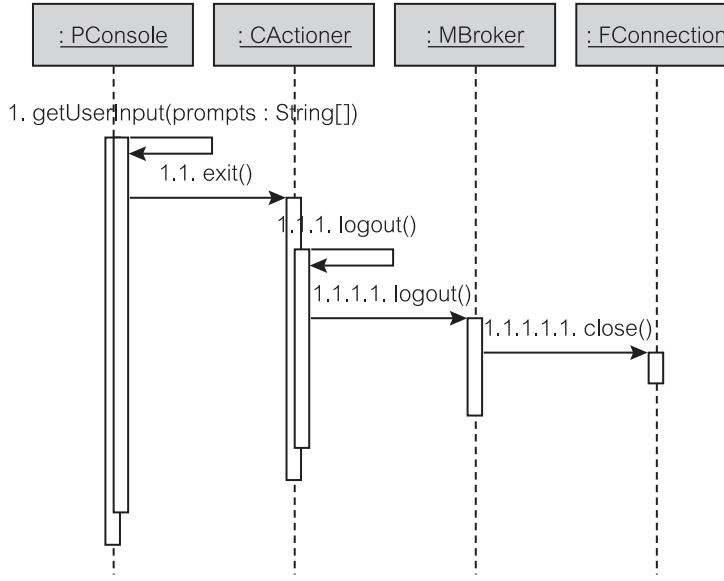


Рис. 11.11. Взаимодействие «Выход»

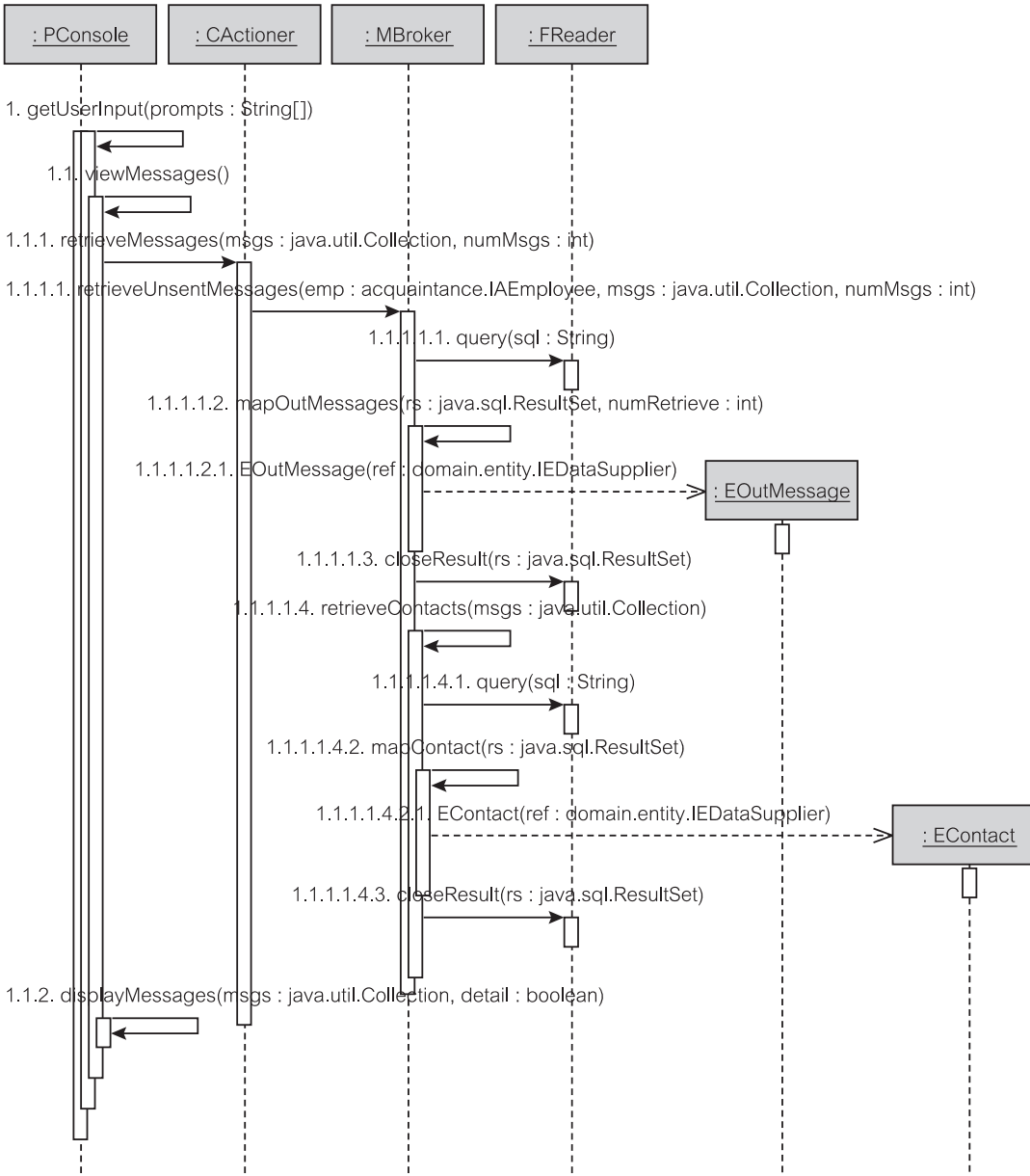


Рис. 11.12. Взаимодействие «Просмотр непосланных сообщений»

`:CActioner` (класс «исполнитель» пакета `control`). Объект `:CActioner` делегирует запрос объекту `:MBroker` (класс «посредник» пакета `mediator`) после помещения его объекта `:EEmployee` (класс «сотрудник» пакета `entity`) в список аргументов (объект `:CActioner` связан с объектом `:EEmployee` во взаимодействии «Регистрационное имя» — см. выше).

Объект :MBroker запрашивает :FReader (класс «чтение» пакета foundation), чтобы тот выполнил следующую операцию query() (запрос) над БД:

```
select * from OutMessage
         where sender_emp_id = ? and date_emailed is null;
```

Результат, полученный операцией query(), используется для того, чтобы создать в кэше коллекцию объектов :EOutMessage (класс «исходящее сообщение» пакета entity) и установить соответствующие хэш-массивы[1]. Конструктор объектов класса :EOutMessage использует интерфейс IEDataSupplier (интерфейс «источник данных» пакета entity). Этот интерфейс определяет коллекцию, в которую может быть помещен объект :EOutMessage.

Тот факт, что коллекция исходящих сообщений возвращается к объекту :MBroker и что инициализированные объекты класса :EOutMessage помещены в эту коллекцию, нельзя соответствующим образом показать в UML-диаграммах взаимодействия (раздел 11.3.1). Диаграммы взаимодействия не предназначены для отображения манипуляций с данными. Сообщения на диаграммах не показывают типы возвращаемых данных. Кроме того, на диаграммах последовательности действий невозможно ясно указать, что сообщение использует ассоциативную связь (переменную, содержащую ссылку на объект или коллекцию объектов), чтобы возвратить объект (или коллекцию объектов) клиенту. Диаграммы связей могут быть более гибки в этом отношении.

Каждый объект :EOutMessage содержит id (идентификатор) делового партнера для данного исходящего сообщения. Эта информация используется объектом :MBroker, чтобы извлечь из БД деловых партнеров для соответствующих исходящих сообщений. Данная операция завершается формированием объектов класса :EContact (класс «деловой партнер» пакета entity) и размещением их в Java-коллекции. В конечном счете :PConsole оказывается в состоянии отобразить исходящие сообщения вместе с информацией о деловых партнерах, используя операцию displayMessages() (отобразить сообщения).

11.4.4. Взаимодействие «Отображение текста сообщения»

«Отображение текста сообщения» (Display Message Text) является прямым взаимодействием. Сущность этого объясняется в примечании к рис. 11.13. Объект :MBroker (класс «посредник» пакета mediator) проверяет, находится ли объект :EOutMessage (класс «исходящее сообщение» пакета entity) уже в памяти, и если его там нет, он должен поместить этот объект в память, используя query() (запрос) объекта :FReader (класс «чтение» пакета foundation). Формирование объекта :EOutMessage (класс «исходящее сообщение» пакета entity) подразумевается и на рис. 11.13 не показано. Как только :EOutMessage будет размещен в памяти, :MBroker воз-

[1] Хэш-массив или ассоциативный массив представляет собой массив данных со специальной адресацией через ключи. (*Прим. перев.*)

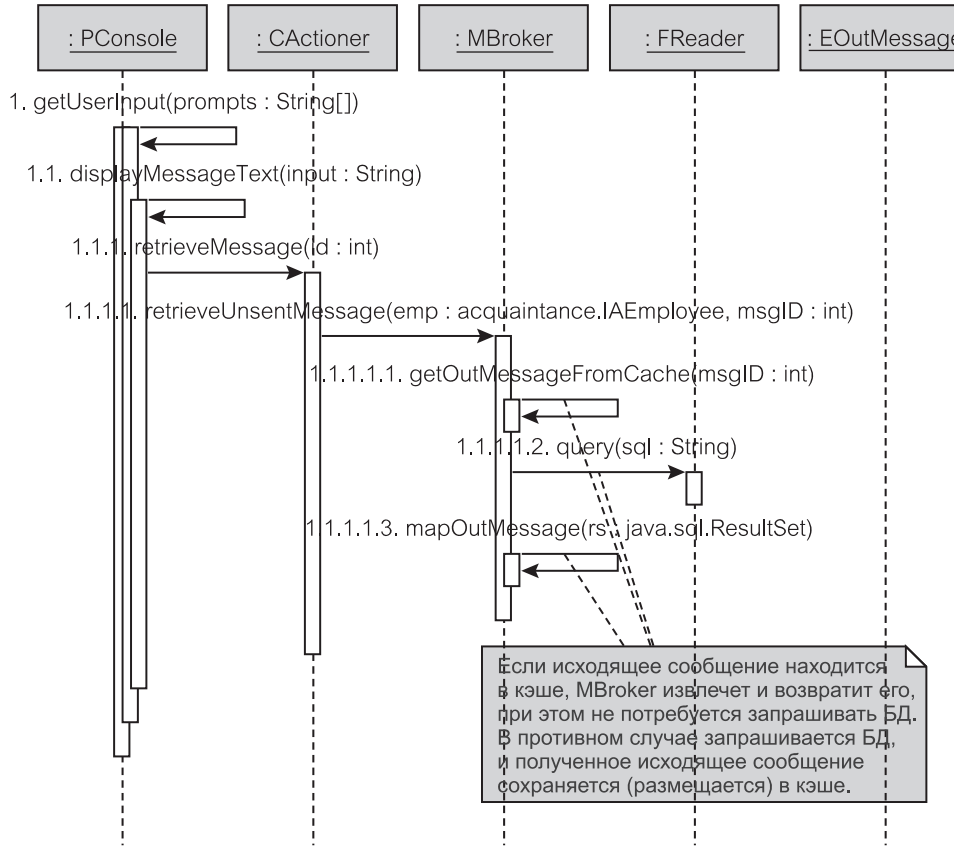


Рис. 11.13. Взаимодействие «Отображение текста сообщения»

вращает его окончательному клиенту, то есть, `:PConsole` (класс «консоль» пакета `presentation`). Согласно объяснению относительно несоответствий диаграмм взаимодействия в предыдущем разделе, на рис. 11.13 нет никакого очевидного сообщения к объекту `:EOutMessage`.

11.4.5. Взаимодействие «Сообщение, передаваемое по электронной почте»

Взаимодействие «Сообщение, передаваемое по электронной почте» (Email Message) начинается, когда пользователь хочет послать по электронной почте текст исходящего сообщения, извлеченного из БД (рис. 11.14). Как только запрос на это взаимодействие сделан, объект `:PConsole` (класс «консоль» пакета `presentation`) начинает готовить сообщение для отсылки, используя операцию `prepareMessage()` (подготовить сообщение). Сначала `prepareMessage()` обращается к объекту `:CActioner` (класс «исполнитель» пакета `control`), чтобы использовать операцию `retrieveMessage()` (извлечь сообщение), как показано во взаимодей-

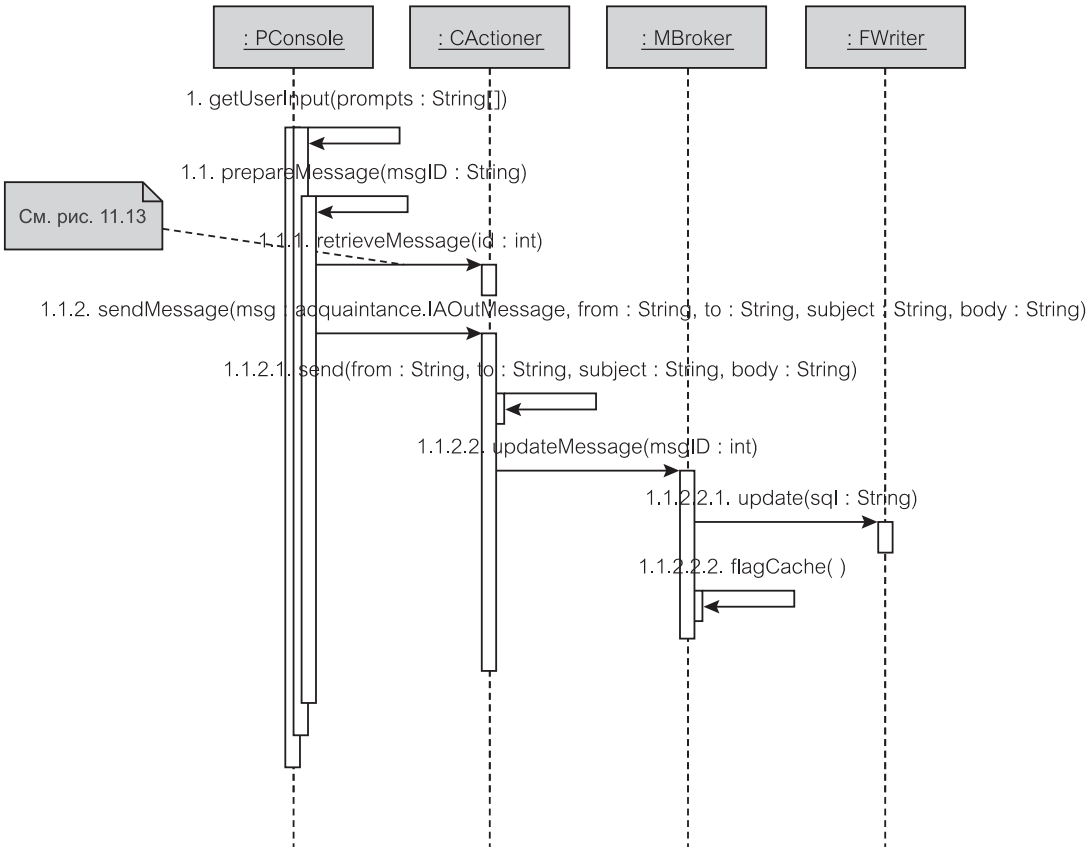


Рис. 11.14. Взаимодействие «Сообщение, передаваемое по электронной почте»

ствии «Отображение текста сообщения». Как только детали исходящего сообщения будут получены, объект `:PConsole` обращается к объекту `:CActioner` для использования операции `sendMessage()` (послать сообщение).

`:CActioner` использует функциональные возможности библиотеки JavaMail, чтобы послать исходящее сообщение по электронной почте с помощью метода `send()` (послать). Если `send()` отработает успешно, метод `sendMessage()` активизирует операцию `updateMessage()` (скорректировать сообщение) класса `:MBroker` (класс «посредник» пакета `mediator`). Класс `:MBroker` делегирует работу классу `:FWriter` (класс «запись» пакета `foundation`), который формирует и выполняет в БД следующий SQL-оператор `update` (корректировка):

```
update OutMessage set date_emailed = ? where message_id = ?;
```

Оператор `update` изменяет запись в таблице `OutMessage` (исходящее сообщение). Это означает что соответствующий объект `:EOutMessage` (класс «исходящее сообщение» пакета `entity`) в кэше программы содержит

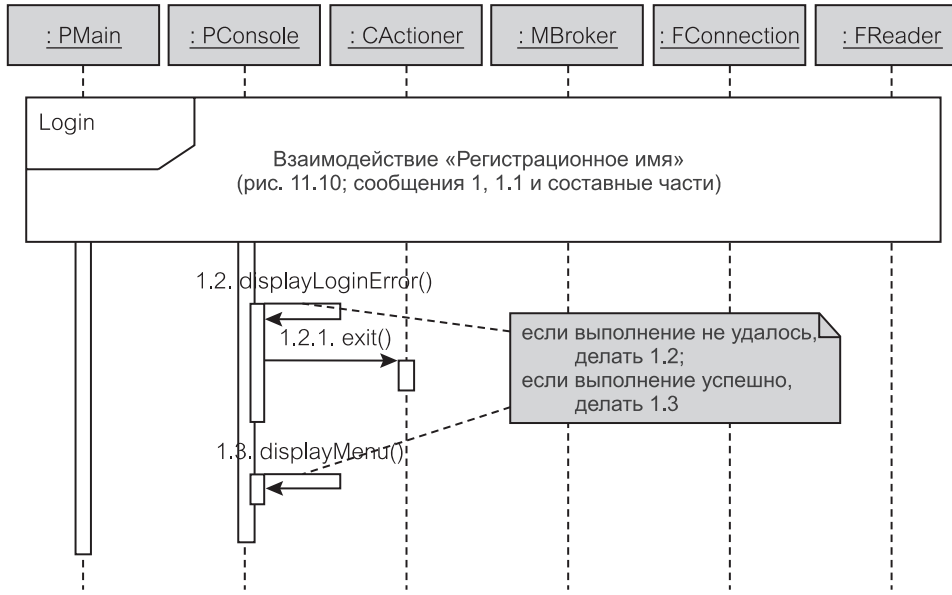


Рис. 11.15. Взаимодействие «Неправильное имя пользователя или неправильный пароль»

неправильные данные. Для этого объект `:MBroker` использует операцию `flagCache()` (флаг кэша), чтобы установить кэш в «измененное» состояние. Далее `:MBroker` запрашивает использование объекта класса `entity` (сущность) с логикой программы, чтобы заново извлечь данные из БД.

11.4.6. Взаимодействие «Неправильное имя пользователя или неправильный пароль»

Взаимодействие «Неправильное имя пользователя или неправильный пароль» (Incorrect User Name or Password) — альтернативный путь выполнения взаимодействия «Регистрационное имя», как показано на рис. 11.15. Если объект `:PConsole` (класс «консоль» пакета `presentation`) возвратит «недопустимое регистрационное имя» с помощью последовательности операций, активизированных операцией `getUserInput()` — получить введенную пользователем информацию (рис. 11.10), он использует операцию `displayLoginError()` (отобразить сообщение об ошибке ввода регистрационного имени), чтобы вывести пользователю ответ: «Неправильное регистрационное имя». Если это происходит третий раз подряд, программа посылает сообщение `exit()` (выход) объекту `:CActioner` (класс «исполнитель» пакета `control`) — см. рис. 11.11.

11.4.7. Взаимодействие «Неправильная опция»

Если регистрационное имя пользователя введено правильно, объект `:PConsole` (класс «консоль» пакета `presentation`) использует операцию `displayMenu()` (отобразить меню), чтобы представить опции (команды)

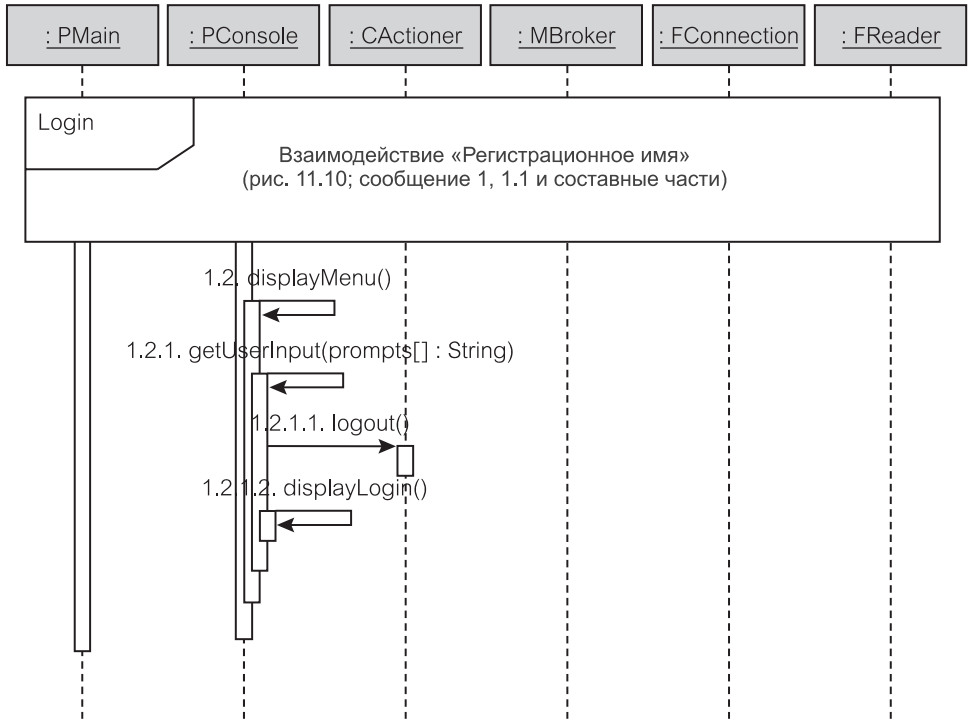


Рис. 11.16. Взаимодействие «Неправильная опция»

меню пользователю (рис. 11.16). Операция `getUserInput()` (получить введенную пользователем информацию) читает введенный пользователем номер опции. Если введенное значение неправильно, программа игнорирует величину и заново показывает меню (это неявно прослеживается на рис. 11.16).

Пользователю позволены три попытки ввести допустимый номер опции (на рис. 11.16 это не показано). Программа получает число разрешенных попыток из константы `MAX_LOGIN_TRY` (максимальное число попыток ввода регистрационного имени), находящейся в интерфейсе `IAConstants` — интерфейс «константы» пакета `acquaintance` (рис. 11.2). После трех ошибок объект `:PConsole` инициализирует операцию `logout()` (завершение работы с зарегистрированным пользователем), и когда управление возвращается обратно, приглашает пользователя заново зарегистрироваться (использует `displayLogin()` — отобразить регистрационное имя).

11.4.8. Взаимодействие «Слишком много сообщений»

Значение константы `MAX_DISPLAY_MESSAGES` (максимальное число отображаемых сообщений) в интерфейсе `IAConstants` — интерфейс «константы» класса `acquaintance` (рис. 11.2) — контролирует максимальное число исходящих сообщений, которые программа подготовила для отображения на консольном экране пользователя. Это необходимо по причинам ограничения возможности использования и из-за ограниченной памяти, доступной программе.

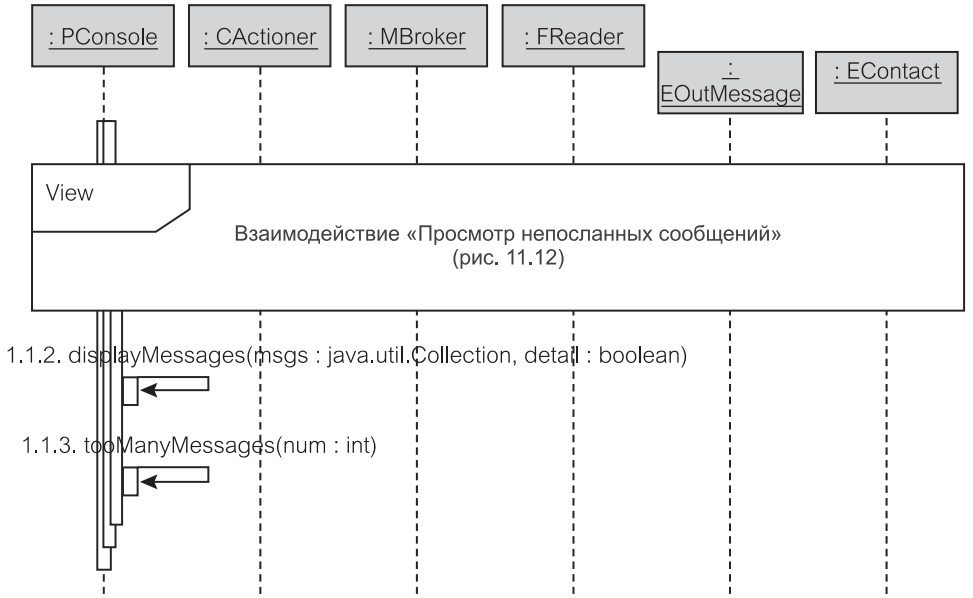


Рис. 11.17. Взаимодействие «Слишком много сообщений»

Как показано на рис. 11.17, взаимодействие «Слишком много сообщений» (Too Many Messages) является расширением взаимодействия «Просмотр непосланных сообщений». Как можно заметить на рис. 11.12, сообщение от объекта `:PConsole` (класс «консоль» пакета `presentation`) к объекту `:CActioner` (класс «исполнитель» пакета `control`), называемое `retrieveMessages()` (извлечь сообщения), содержит число исходящих сообщений (`numMsgs`) в своем списке аргументов. Это гарантирует, что только такое количество исходящих сообщений будет извлечено из БД и возвращено объекту `:PConsole` в контейнере, передаваемом также в виде аргумента (`msgs` — сообщения).

Исходящие сообщения отображаются операцией `displayMessages()` (отобразить сообщения). После отображения метод `tooManyMessages()` (слишком много сообщений) сообщит пользователю, сколько исходящих сообщений не было извлечено из БД. В итерации 1 пользователь может видеть эти оставшиеся исходящие сообщения только после передачи по электронной почте некоторых из исходящих сообщений, в настоящее время видимых на экране.

11.4.9. Взаимодействие «Сообщение не может быть послано по электронной почте»

Взаимодействие «Сообщение не может быть послано по электронной почте» (Email Could Not Be Sent) делит взаимодействие «Сообщение, передаваемое по электронной почте» на два фрагмента, как показано на рис. 11.18. Когда объект `:CActioner` (класс «исполнитель» пакета `control`) получает Java-исключение, означающее, что сообщение по электронной почте не было

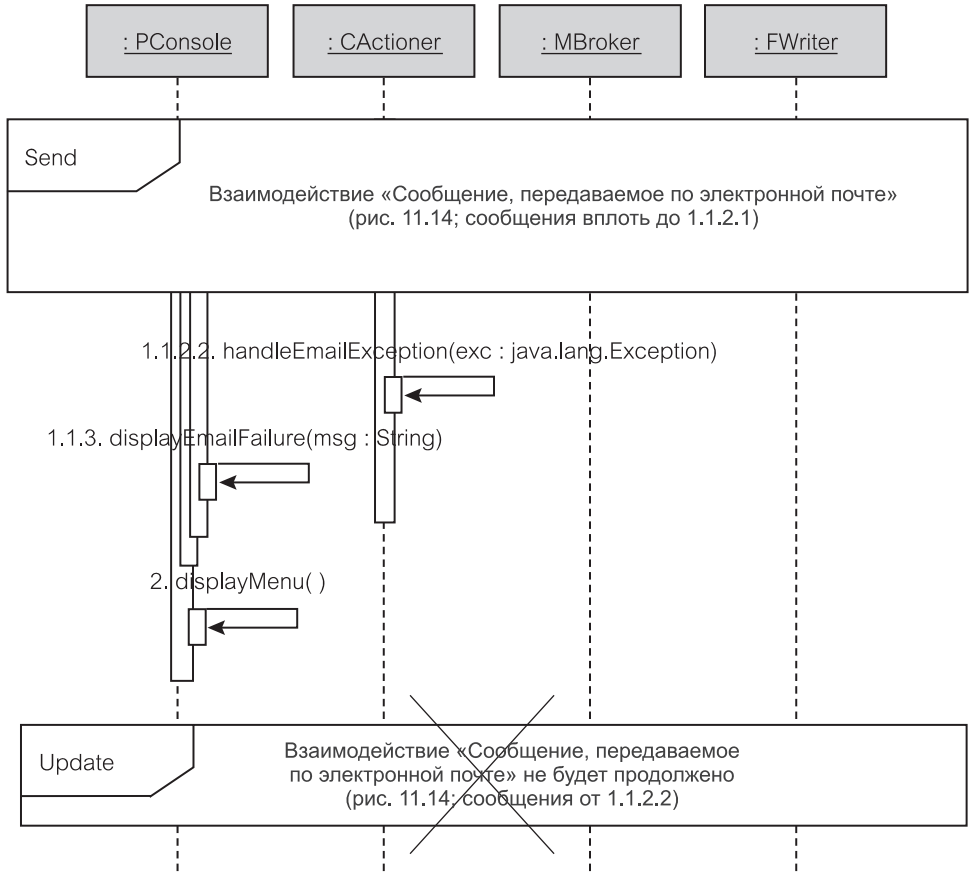


Рис. 11.18. Взаимодействие «Сообщение не может быть послано по электронной почте»

послано, метод `handleEmailException()` (управление исключениями от электронной почты) интерпретирует проблему таким образом, что операция `displayEmailFailure()` (отобразить ошибку электронной почты) объекта `:PConsole` (класс «консоль» пакета `presentation`) может сообщить пользователю о том, что передача по электронной почте не удалась. После этого `:PConsole` заново отображает меню с помощью операции `displayMenu()` (отобразить меню). Фрагмент взаимодействия, связанный с корректировкой БД, в этом случае не будет выполняться.

Резюме

1. *Проект классов* и *проект взаимодействия* — две стороны одной и той же медали. Они должны выполняться в параллель.
2. *Определение классов*, исходя из требований сценария использования, включает выделение требований из документа сценария использования и формирование классов и сотрудничества между классами, которые необходимы для выполнения этих требований.

3. Классы должны соответствовать выбранному структурному шаблону. Структурные ограничения вызывают потребность в *разработке* исходного проекта классов.
4. *Java-интерфейс* является удобным способом хранения констант и средством параметризации поведения программы.
5. Структурная разработка включает решения относительно *формирования экземпляров* классов.
6. *Взаимодействия* моделируются в диаграммах последовательности действий и диаграммах связей.
7. *Диаграмма последовательности действий* определяет последовательность сообщений и возникновение их событий на линиях жизни.
8. *Диаграмма связей* (до UML 2.0 известная как диаграмма сотрудничества) — просто визуальная разновидность диаграммы последовательности действий.
9. При моделировании взаимодействий следует различать сообщения синхронные, асинхронные, обратной связи и создания объекта.
10. Взаимодействия сосредотачиваются на последовательностях сообщений, а не на данных, которые эти сообщения передают. Поэтому сигнатуры сообщений не нужно показывать, типы возвращаемых данных обычно также не визуализируются.
11. Взаимодействие, определенное в диаграмме взаимодействия, называется *экземпляром взаимодействия*. *Диаграмма просмотра взаимодействий* — ограниченная разновидность *диаграммы деятельности*, в которой узлы-объекты заменены взаимодействиями и экземплярами взаимодействий.

Ключевые термины

Обзорные вопросы

Вопросы для обсуждения

1. Проект классов включает проект интерфейсов. Какие требования — функциональные или нефункциональные — являются «движущей силой» проекта интерфейсов? Объясните. (Предупреждение — проект интерфейсов в этом вопросе касается проектов интерфейсов UML/Java, а не проекта пользовательского интерфейса.)
2. Структура таблицы определения классов по требованиям сценария использования (таблица 11.1) не является полной и правильной в отношении структурного шаблона и в соответствии со всеми PCMEF-принципами (раздел 9.2.2). Какие PCMEF-принципы не подтверждаются, если изучить таблицу 11.1? Приемлемо ли это?
3. Диаграммы взаимодействия и диаграммы последовательности действий в частности, широко используются, чтобы представить детальную логику поведения программы. Однако они не способны выразить некоторый вид взаимосвязи объектов, и это может привести к необъяснимым пробелам в логических рассуждениях. Что это за ограничение диаграмм взаимодействия? Оправдано ли оно?

Вопросы учебного примера

1. Обратимся к требованию R11 в таблице 11.1, которое ссылается на поток исключений *E3 — слишком много сообщений*. Как это исключение реализовано в итерации 1? Гарантирует ли эта реализация, что память программы не будет переполнена при извлечении исходящих сообщений из БД?
2. Обратимся к диаграмме последовательности действий «Отображение текста сообщения» на рис. 11.13. Какое здесь сделано предположение, и как вы могли бы выполнить это по-другому, если такое предположение не было бы сделано? Намек: предположение касается взаимодействия «Просмотр непосланных сообщений».
3. Одним из подходов к очистке объектов в памяти — использование Java-метода `finalize()` (завершить) во время «сборки мусора». Объясните, когда это можно использовать в учебном примере EM-программы.
4. Можно ли рассматривать взаимодействие «Слишком много сообщений» как тип программирования исключений? Может ли оно быть реализовано с помощью Java-исключения?

Примеры задач

Упражнения учебного примера

1. Обратимся к проекту класса `PMain` — основной (рис. 11.3) и к диаграмме формирования экземпляра для EM (рис. 11.4). Может ли `PMain` инициализировать больше, чем один класс и все еще удовлетворять PCMEF-шаблону? Изобразите соответствующий фрагмент диаграммы формирования

экземпляра. По возможности напишите код для `PMain.java`. Дайте необходимые объяснения.

2. Обратимся к проекту класса `PMain` (рис. 11.3) и к диаграмме формирования экземпляра для EM (рис. 11.4). Предположим наличие так называемого «энергичного формирования экземпляров», когда `PMain` имеет возможность инициализировать в своем конструкторе несколько (или даже большинство) классов приложения. Это не будет соответствовать PCMEF-шаблону, но все-таки такую операцию можно допустить как обоснованное исключение. Изобразите соответствующий фрагмент диаграммы формирования экземпляра. По возможности напишите код для `PMain.java`. Дайте необходимые объяснения.
3. Рассмотрим взаимодействие «Выход» (раздел 11.4.2). Преобразуйте диаграмму последовательности действий на рис. 11.11 в диаграмму связей.
4. Рассмотрим взаимодействие «Сообщение, передаваемое по электронной почте» (раздел 11.4.5). Преобразуйте диаграмму последовательности действий на рис. 11.14 в диаграмму связей.

Небольшой проект — система использования временного протокола

Этот небольшой проект предполагает, что вы имеете решение вопросов, связанных с системой использования временного протокола (Time Logging System — TLS), рассмотренных в главах 7 и 8. Эти решения включены в руководство для преподавателя, использующего данный учебник, и доступны преподавателям на Web-сайте книги.

1. Найдите классы проекта для TLS, используя табличный подход, как рассмотрено в главе 11, таблица 11.1. Опишите входные требования таблицы в стиле, подобном используемому в главе 11. Для простоты предположите, что достаточно иметь два класса сущностей — `EEmployee` (класс «служащий» пакета `entity`) и `ETimeLogRecord` (класс «запись временного протокола» пакета `entity`).
2. Разработайте диаграмму классов проекта для TLS, отражающую ваше решение вопроса 1. Покажите классы, интерфейсы и названия методов (сигнатуры методов определять не нужно). Объясните ваши основные предположения и упрощения.
3. В вопросах 1 и 2 мы предполагали упрощенную версию уровня `entity` (сущность). В действительности объектная модель предметной области для TLS (глава 7, небольшой проект — временной протокол, вопрос 2) выявила бы и другие сущности типа `Activity` (операция), `Task` (задача), `Calendar` (календарь) и `Contact` (деловой партнер). На основе модели объектов предметной области для TLS (предложенной вашим преподавателем) представьте более полную диаграмму классов проекта для уровня `entity` (сущность). Добавьте интерфейсы и объясните, почему они необходимы. Покажите основные методы. Элементы данных показывать не надо.
4. Разработайте диаграмму последовательности действий для взаимодействия «Отобразить окно временной регистрации».

5. Разработайте диаграмму связей для взаимодействия «Регистрация и получение вхождений временного протокола». Модель не должна показывать вхождения временного протокола, а только размещать их в памяти. Дайте краткое объяснение.
6. Разработайте диаграмму последовательности действий для взаимодействия «Отображение вхождения временного протокола». Модель должна предполагать, что вхождения временного протокола уже были размещены в памяти. Дайте краткое объяснение.
7. Разработайте диаграмму последовательности действий для взаимодействия «Отображение вхождения временного протокола для корректировки». Это взаимодействие должно представлять диалоговое окно для отобранного вхождения временного протокола, возможно потому, что служащий хочет скорректировать вхождение. Дайте краткое объяснение.
8. Разработайте диаграмму последовательности действий для взаимодействия «Сохранение вхождения временного протокола после корректировки». Это взаимодействие сохраняет в БД изменения, сделанные в диалоговом окне корректировки. Дайте краткое объяснение.
9. Разработайте диаграмму последовательности действий для взаимодействия «Синхронизация временного протокола». Это взаимодействие необходимо, чтобы позволить служащему входить в протокол TLS с различных рабочих станций в одно и то же время. TLS должна быть способна периодически синхронизировать отображенные вхождения временного протокола с самыми последними данными, доступными в БД.
10. Разработайте диаграмму связей для взаимодействия «Создание нового вхождения во временной протокол».
11. Разработайте диаграмму связей для взаимодействия «Удаление вхождения во временной протокол».

Небольшой проект — управление информацией о деловых партнерах

Этот небольшой проект предполагает, что вы имеете решения вопросов, связанных с CIM, рассмотренных в главах 9 и 10. Эти решения включены в руководство для преподавателя, использующего этот учебник, и доступны преподавателям на Web-сайте книги.

1. Найдите классы проекта для CIM, используя табличный подход, как рассмотрено в главе 11, таблица 11.1. Опишите входные требования таблицы в стиле, подобном используемому в главе 11.
2. Разработайте диаграмму классов проекта для CIM, отражающую ваше решение вопроса 1. Покажите классы и интерфейсы (методы показывать не нужно). Объясните ваши основные предположения и упрощения.
3. Разработайте диаграмму последовательности действий для взаимодействия «Отображение окна регистрационного имени».
4. Разработайте диаграмму связей для взаимодействия «Создание регистрационного имени».

5. Разработайте диаграмму последовательности действий для взаимодействия «Список всех деловых партнеров».
6. Разработайте диаграмму последовательности действий для взаимодействия «Отображение деталей делового партнера».
7. Разработайте диаграмму связей для взаимодействия «Создание нового PersonContact (делового партнера)».
8. Разработайте диаграмму последовательности действий для взаимодействия «Корректировка информации о деловом партнере».

Глава 12

Программирование и тестирование

Постоянная обратная связь между *программированием и тестированием* является центральным моментом любой разработки системы. Цель разработки системы — программный продукт — программа, которая работает и отвечает требованиям заинтересованных сторон. Анализ требований необходим для их документирования и обобщения. Проектирование системы необходимо, чтобы «разделять и властвовать» над сложностью проблем предметной области и обеспечить надлежащее управление проектом. Но реальный решающий момент заключается в программировании и тестировании.

В быстрой разработке ПО [66] тестирование занимает ведущее место и управляет программированием. Такая ситуация известна как **управляемое тестированием программирование**. Код тестирования пишется перед кодом приложения. Задачей прикладного программиста является создание кода, который пропускается через тестовую программу. Тесты определяют, что было запрограммировано. Они также определяют детальный проект классов. По этой причине управляемое тестированием программирование иногда считают **управляемой тестированием разработкой**.

Программа является ответом на результаты тестирования. Программа обеспечивает и подтверждает свое тестирование. Если это так, то что подтверждает само тестирование? Ответ, однако, находится в другом виде тестирования — приемочных испытаниях. **Приемочные испытания** проверяют, удовлетворяет ли код приложения (и по этой причине они являются тестированием, пропускающим через себя приложение) требованиям заинтересованных сторон.

Никакое тестирование не может быть полным и исчерпывающим. Приемочные испытания не могут доказать правильность работы программы. Даже в небольшой системе число возможных путей выполнения программы слишком велико, чтобы проверить их все. Точно так же в управляемой тестированием разработке написание тестовой программы, проверяющей каждую функцию и каждую комбинацию функций, невыполнимо и даже может быть бесполезным.

Имеется компромисс между стоимостью и возможностями тестирования. Компромисс определяет количество тестовых трудозатрат по отношению к сумме трудозатрат на программирование. Подобный компромисс, хотя бы с учебной точки зрения, должен быть рассмотрен в книге при обсуждении программирования и тестирования. Данная глава объясняет управляемую тести-

рованием разработку, программирование и приемочные испытания. Учебный пример EM служит для иллюстрации технологии и процессов.

До того, как будет представлен учебный пример, быстрое обучение языку Java подготавливает читателя к более сложному коду в реализации учебного примера. Обучение проводится с точки зрения разработки ПО. Оно подчеркивает важность поддержания ассоциаций классов в программах, обработки коллекций, управления кэшами объектов и доступа к БД.

12.1. Быстрое обучение языку Java с точки зрения разработки программного обеспечения

Раздел 10.1 содержал быстрое обучение реляционным БД с точки зрения разработки ПО. Обучение использовало простую БД `MovieActor` («Фильм-Актер»), чтобы объяснить основные принципы проектирования БД и концепции программирования. Обучение языку Java, представленное ниже, развивает простую прикладную программу `MovieActor`, которая обеспечивает доступ к БД `MovieActor`.

Обучение включает отобранные концепции Java-языка, фундаментальные для разработки и создания Java-систем. В отличие от учебного примера EM здесь не делается никаких попыток разместить обсужденные решения в рамках проекта и структурного шаблона. Весь код размещен только в пяти классах, три из которых — классы сущностей, соответствующих трем таблицам в БД `MovieActor`.

Полные исходный код Java и Oracle. а также UML-модель для этого цикла обучения могут быть взяты с Web-сайта книги. Фрагменты кода, представленные ниже, сохраняют номера строк для операторов программы такими же, что и в полном решении, находящемся в документации программы на Web-сайте. По этим причинам имеются промежутки в номерах строк фрагментов кода, представленных ниже.

12.1.1. Класс

Класс в Java является шаблоном для создания объекта. В соответствии с этим он определяет следующие концепции [56]:

- его имя и видимость;
- элементы-переменные (элементы данных);
- элементы-функции (методы);
- конструкторы;
- видимость элементов;
- его суперкласс (если он есть) и интерфейсы (если они имеются).

Классу и его элементам должен быть назначен уровень **видимости**. Видимость класса определяет, как другие объекты программы могут обращаться к его экземплярам и свойствам. Видимость класса обычно объявляется как *public* (общедоступный), чтобы все другие объекты могли иметь свободный доступ к экземплярам класса.

Видимость элементов может быть public (общедоступный), protected (защищенный), private (приватный) или package (доступный в пакете). Большинство элементов данных объявляются как private (доступные только экземплярам класса), а большинство методов — как public (доступно для экземпляров любого класса). Множество всех методов класса категории public определяет **интерфейс класса**. Поскольку термин «интерфейс класса» представляет разные концепции для *интерфейса Java* и *UML-интерфейса*, он иногда называется **протоколом класса**, чтобы избежать путаницы в терминологии [56]. Видимость рассмотрена более глубоко далее в книге.

Имеются две категории **элементов данных** (или **полей**): переменные экземпляра и переменные класса. **Переменная экземпляра** принадлежит объекту (экземпляру). Она может быть простого типа данных или типа-класса. Переменная **простого типа данных** хранит непосредственно величину этого типа. **Тип-класс** — определенный пользователем тип, указывающий на предварительно определенный в программе класс. Переменная этого типа хранит ссылку (указатель) на объект класса. Поэтому данный тип иногда называют **ссылочным типом данных**.

В объектно-ориентированном мире «все является объектами». Хранение данных может быть реализовано только в объекте. Выполнение метода может быть осуществлено только на объекте. Поскольку «все является объектами», с самим классом нужно иногда обращаться как с объектом, который может хранить данные и выполнять методы. *Переменная класса* принадлежит классу и доступна для всех объектов этого класса. Все объекты обладают одной и той же переменной класса. Переменные класса определяются ключевым словом `static` (статический).

Локальная переменная, используемая «локально» в теле метода класса, не является элементом данных. Элементы данных могут хранить постоянные или переменные величины. Локальные переменные не могут быть определены как константы. Константы определяются с ключевым словом `final` (конечный).

Как и элементы данных, **методы** могут быть методами экземпляра или методами класса. **Метод экземпляра** принадлежит объекту. **Метод класса** принадлежит классу (то есть всем объектам этого класса). Методы класса определяются с ключевым словом `static`. К ним можно обращаться или через объект класса (например, `objectVariable.ClassMethod()` — <переменная-объект>.<метод_класса()) или без использования объекта через имя класса (например, `className.ClassMethod()` — <имя_класса>.<метод_класса>()).

Java поддерживает **перегрузку методов**, позволяя нескольким методам класса иметь одно и то же имя. Перегружаемые методы отличаются различными сигнатурами. **Сигнатура метода** — это список аргументов (параметров) метода. Тип величины, которая возвращается методом объекту-клиенту, называют возвращаемым типом. **Типы аргументов** и **возвращаемые типы** могут быть простыми типами или типами-классами. **Прототип метода**, иногда называемый *заголовком* [56], состоит из имени метода, его сигнатуры и возвращаемого типа. Прототип метода может или использоваться в интерфейсах или быть объявлен как абстрактный, потому что он не имеет реализации.

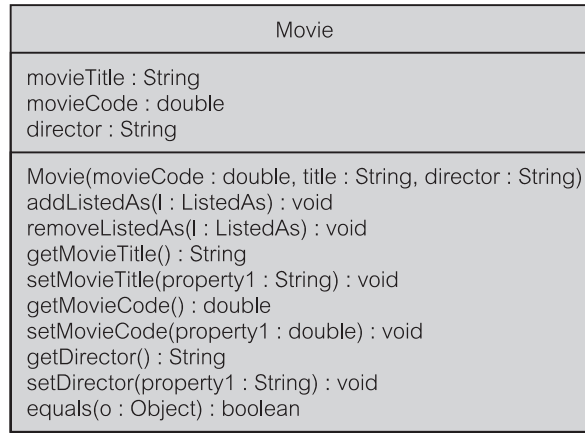


Рис. 12.1. Проект класса Movie

Конструктор — это специальный метод, который используется, чтобы инициализировать класс (то есть создать объект класса). Конструктор может иметь аргументы и может быть перегружен таким образом, чтобы объекты можно было создать различными путями. Имя конструктора то же самое, что и имя класса, для которого он был определен.

Рис. 12.1 показывает UML-проект класса `Movie` (кинофильм). Класс `Movie` — это класс-сущность, который соответствует (отображает) таблице `movie` на рис. 10.1. Класс состоит из трех элементов данных (все они являются переменными экземпляра), конструктора и ряда методов экземпляра со своими сигнатурами. Элементы данных, имеющие типы-классы (ссылки), не видны в средней части UML-класса, потому что они представляются как имена ролей на линиях ассоциации (раздел 12.1.2).

Листинг 12.1. Movie.java[1]

Movie.java

```
7:   public class Movie {
8:      private String movieTitle;
9:
10:     private double movieCode;
11:
12:     private String director;
13:
14:     private Collection listedAs;
15:
```

[1] В данном и последующих листингах пронумерованы отдельные операторы, в том числе и пустые. Если оператор не умещается на одной строке книги, он переносится на следующую сроку или последующие строки. Все эти строки идут под одним и тем же номером. Следует учитывать, что при конкретном написании таких операторов каждый из них должен занимать только одну строку, сколь бы длинной она ни была. В противном случае при компиляции программы может возникнуть сообщение об шибке. (*Прим. перев.*)

```
16:     public Movie(double movieCode,String title,String
        director){
17:        this.movieCode = movieCode;
18:        this.director = director;
19:        this.movieTitle = title;
20:        listedAs = new ArrayList();
21:     }
22:
23:     public void addListedAs(ListedAs l){
24:        listedAs.add(l);
25:     }
26:
27:     public void removeListedAs(ListedAs l){
28:        listedAs.remove(l);
29:     }
30:
35:     public String getMovieTitle() {
36:        return movieTitle;
37:      }
38:
43:     public void setMovieTitle(String title) {
44:        this.movieTitle = title;
45:     }
46:
51:     public double getMovieCode() {
52:        return movieCode;
53:     }
54:
59:     public void setMovieCode(double code) {
60:        this.movieCode = code;
61:     }
62:
67:     public String getDirector() {
68:        return director;
69:     }
70:
75:     public void setDirector(String director) {
76:        this.director = director;
77:     }
78:
79:     public boolean equals(Object o){
80:        try{
81:           Movie m = (Movie) o;
82:           if(m.movieCode == movieCode) return true;
83:        }catch(Exception exc){}
84:        return false;
85:     }
86:
87:  }
```

Листинг 12.1 представляет полное определение (реализацию) Java-класса `Movie.java` для проекта на рис. 12.1. Определение включает свойства видимости (скрытые в визуальном отображении на рис. 12.1). Тип параметра `listedAs` (перечисление) — `Collection` (коллекция), рассмотренный в разделе 12.1.2.

12.1.2. Ассоциации и коллекции классов

Как обсуждалось раньше, начиная с главы 9, понятность, удобство сопровождения и масштабируемость систем являются предпосылками создания путей сотрудничества объектов, ясно видимых в коде. Одним из способов обеспечения этой видимости является задание связей ассоциации между классами, которые сотрудничают при передаче сообщений. Структурный PCMEF-шаблон способствует этому.

Как было указано раньше, ассоциации между классами обычно представляются переменными экземпляра, которые хранят ссылки на связанные объекты — только одна ссылка в случае ассоциации с множественностью «один» или коллекция ссылок в случае множественности «многие».

От концептуальной модели к модели проектирования классов

Большинство крупных систем реализует то или иное кэширование записей таблицы в виде объектов класса в памяти программы (раздел 10.2). Рисунки 10.2 и 10.4 показывают отображение реляционной модели БД в концептуальную модель классов для приложения `MovieActor` («Фильм-Актер»). Концептуальная модель классов может служить отправной точкой для проекта классов в пакете `entity` (сущность) программы `MovieActor`, который соответствует PCMEF-шаблону.

Концептуальная модель классов для `MovieActor` использовала класс ассоциации `listed_as` (перечисление), чтобы хранить атрибут `position` (позиция). Классы `movie` (фильм) и `actor` (актер) связаны ассоциацией «многие ко многим». Это была сжатая и выразительная модель.

Преобразование концептуальной модели `MovieActor` к модели проекта должно явно сохранить три класса-сущности, как показано на рис. 12.2. В соответствии с соглашением Java имена классов начинаются с заглавных букв, а в именах атрибутов знаки подчеркивания заменяются заглавными буквами последовательных слов. Добавляются имена методов, поскольку проект классов-сущностей должен быть помещен в полный контекст проекта взаимодействий (раздел 11.3).

Что может быть сюрпризом на рис. 12.2, так это то, что ассоциация «многие ко многим» не была преобразована к двум ассоциациям «один ко многим», следуя принципу, рассмотренному в разделе 10.2.2 (хотя и рассмотренному в контексте отображения таблиц). Вместо этого появился несколько замысловатый проект с четырьмя однонаправленными ассоциациями с множественностями «один». Чтобы разобраться с этим проектом, необходимо объяснить понятие Java-коллекции, что будет сделано ниже.

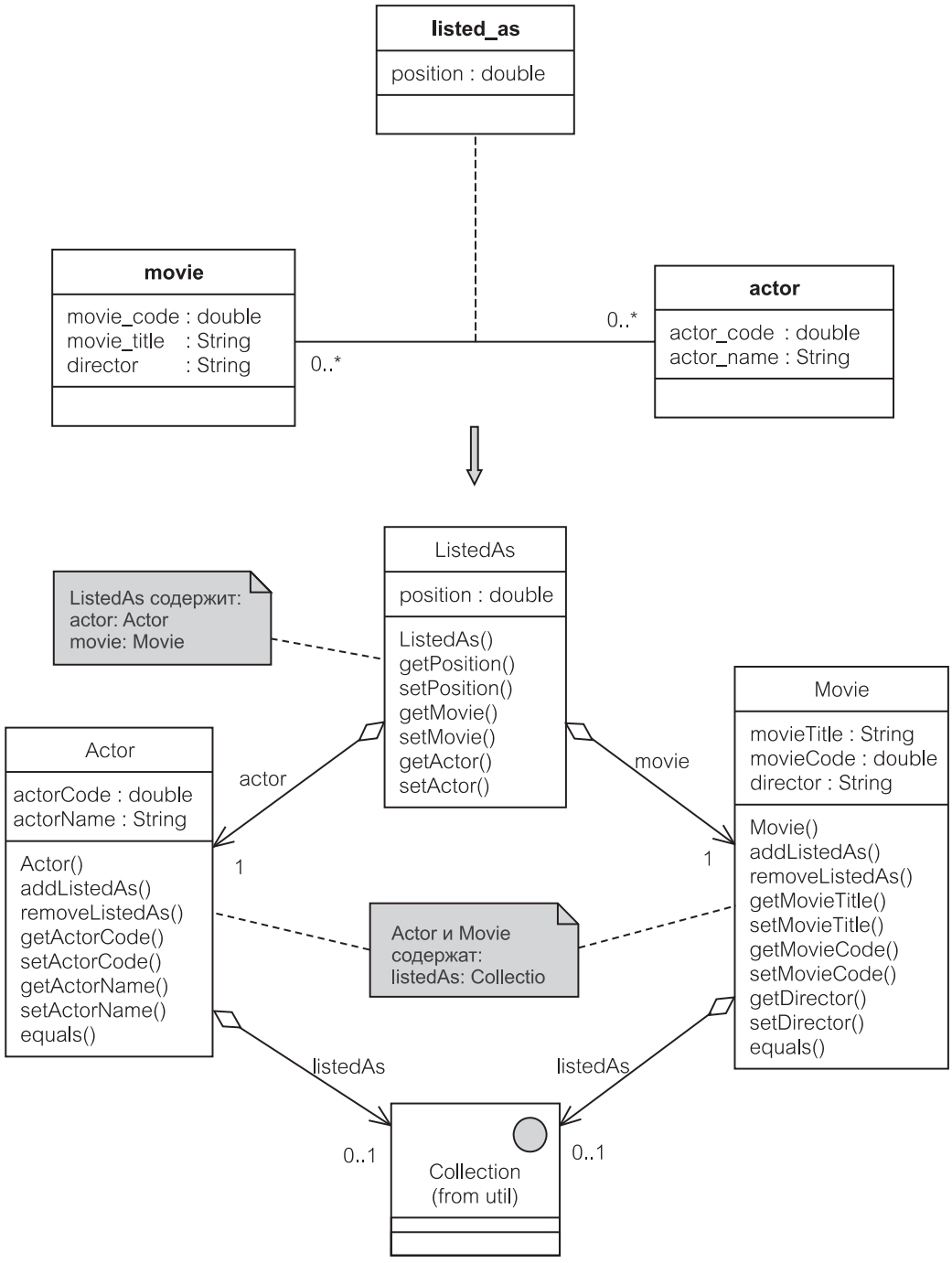


Рис. 12.2. От концептуальной модели к модели проектирования классов

Коллекции Java

В отличие от объектно-ориентированных БД объектно-ориентированные языки программирования не обеспечивают явную поддержку для реализации и поддержания ассоциаций между объектами. Это обязанность программиста запрограммировать ассоциации в классах и обеспечить целостность связей ассоциации. Задача может стать весьма хитрой, в особенности, в случае двунаправленных ассоциаций, в которых модификация связи в одном объекте требует соответствующей модификации связи в связанном объекте.

С точки зрения структуры данных **связь ассоциации** размещается в переменной экземпляра. В случае множественности «один» переменная экземпляра имеет тип-класс. Значением переменной является ссылка на объект этого класса. В случае множественности «многие» переменная экземпляра имеет тип интерфейса `Collection` (коллекция) из библиотеки Java `java.util.Collection`. Значением переменной является ссылка на некоторый объект, который реализует интерфейс `Collection`. Рис. 12.3 показывает интерфейс `Collection`.

Интерфейс `Collection` представляет **контейнер** для объектов (называемых *элементами*). Его методы позволяют добавлять новые элементы, удалять существующие элементы, находить элемент, если он находится там, проверять, является ли коллекция пустой, проверять его размер (число элементов), размещать элементы в массив и просматривать его элементы, используя объект `Iterator` (итератор).

Библиотека `java.util.Collection` предоставляет три категории контейнеров: `List` (список), `Set` (множество) и `Map` (карта). Эти категории сами представляют интерфейсы, каждый лишь с двумя или тремя реализациями [25].

Конкретными классами, которые реализуют функциональные возможности интерфейса `List`, являются `ArrayList` (список-массив), `LinkedList` (связанный список) и `Vector` (вектор). `ArrayList` — упорядоченный список элементов, реализованных в виде массива. Он обеспечивает быстрый прямой доступ к элементам, но может быть медленным при добавлении и удалении элементов. `LinkedList` лучше для последовательного перемещения по элементам, но медленнее при случайном доступе к конкретным элементам. `Vector` рассматривается как класс, доставшийся по наследству, но остается популярным вариантом для хранения связей ассоциации [56].

Имеются два конкретных класса, реализующих функциональные возможности интерфейса `Set`: `HashSet` (хэш-множество) и `TreeSet` (дерево-множество). Оба гарантируют, что элементы уникальны (поскольку множество не разрешает наличие двойных элементов). `HashSet` использует функцию `hashing` (хэширование), чтобы обеспечить быстрый доступ к конкретным элементам. `TreeSet` реализует структуру дерева, которая содержит элементы в отсортированном порядке. Это облегчает отслеживание, основанное на порядке сортировки, формирование отсортированного подмножества или нахождение первого либо последнего объекта.

Интерфейс `Map` поддерживает пары с ключевым параметром, которые позволяют находить значение, используя ключ. Индексы реляционных БД ис-

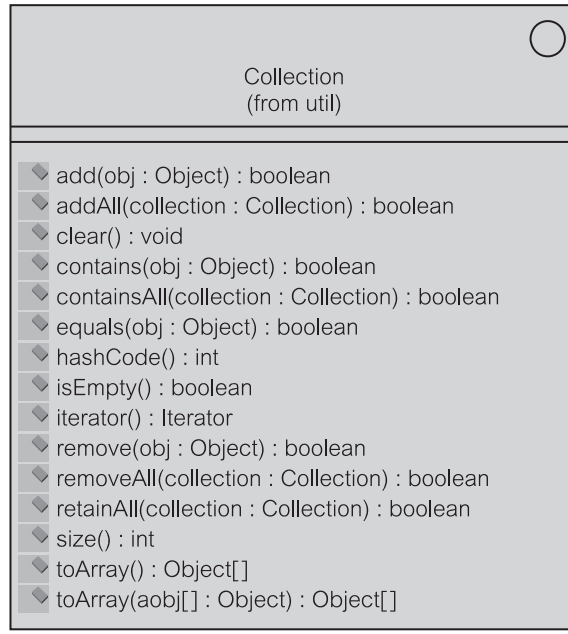


Рис. 12.3. Интерфейс Collection

пользуют тот же принцип (раздел 10.1.6). Имеются две реализации интерфейса `Map`: `HashMap` (хэш-карта) и `TreeMap` (дерево-карта). `HashMap` использует специальную величину, называемую хэш-кодом, который позволяет обеспечить быстрый доступ к ключу объекта и поэтому быстро возвратить значение объекта. `TreeMap` позволяет просматривать значения объектов в порядке отсортированных ключей объектов.

Возвращаясь к рис. 12.2, можно видеть, что отношения агрегирования от `Actor` и `Movie` к интерфейсу `Collection` имеют множественности `0..1`. Необязательное участие (то есть множественность, равная нулю) означает, что объект `Actor` или `Movie` может быть вовсе не перечислен в объекте `ListedAs` (перечисление). Множественность «один» означает, что объект `Actor` или `Movie` может быть связан с объектом, который содержит коллекцию ссылок на «много» объектов `ListedAs`. Поэтому коллекция представляет множественность «многие» в концептуальной модели.

Коллекции Java не знают классов объектов, которые они содержат. В действительности же они содержат ссылки на объекты класса `Object` (объект), который является корнем всех классов Java. Однако фактически коллекция Java позволяет использовать ссылки на размещенные объекты любого класса. Чтобы иметь необходимую степень безопасности, Java обеспечивает приведение объектов к правильному типу, когда они извлечены из коллекции. С точки зрения безопасности типов решение языка C++ использовать параметризованные типы предпочтительней для реализации связей ассоциации.

Ассоциации на объектах-сущностях

Листинг 12.1 представляет класс `Movie`. Листинг 12.2 представляет фрагмент класса `Actor`, который показывает его элементы данных. Переменная `listedAs` определяет коллекцию ссылок на объекты `ListedAs`. Коллекция — это `ArrayList`, созданная внутри конструктора `Actor()`. Методы `addListedAs()` (добавить в список) и `removeListedAs()` (удалить из списка) используются, чтобы добавлять или удалять элементы `ArrayList`. Это заменяет (косвенно, через `ListedAs`) связи ассоциации между актерами и фильмами.

Листинг 12.2. Actor.java

Actor.java

```
7:   public class Actor {
8:      private double actorCode;
9:
10:     private String actorName;
11:
12:     private Collection listedAs;
13:
14:     public Actor(double code, String name){
15:        this.actorCode = code;
16:        this.actorName = name;
17:        listedAs = new ArrayList();
18:     }
19:
20:     public void addListedAs(ListedAs l) {
21:        listedAs.add(l);
22:     }
23:
24:     public void removeListedAs(ListedAs l) {
25:       listedAs.remove(l);
26:     }
66:  }
```

Листинг 12.3 представляет фрагмент класса `ListedAs`. Он показывает, как каждый объект `ListedAs` поддерживает связи со своими объектами `Movie` и `Actor`. Ссылки на объекты `Movie` и `Actor` передаются в аргументах вызова конструктора `ListedAs()`. Как только объект `ListedAs` будет создан, он передает себя в качестве аргумента в вызовах методов `addListedAs` объектов `Movie` и `Actor`. Эти вызовы обеспечивают реализацию «ссылочной целостности» между `ListedAs` и `Movie` и между `ListedAs` и `Actor`.

Листинг 12.3. ListedAs.java

ListedAs.java

```
5:   public class ListedAs {
6:      private double position;
7:
8:      private Movie movie;
9:
10:     private Actor actor;
11:
12:     public ListedAs(Movie m, Actor a, double position){
13:        this.movie = m;
14:        this.actor = a;
15:        this.position = position;
16:
17:        //регистрация этого ListedAs у Movie и Actor
18:        movie.addListedAs(this);
19:        actor.addListedAs(this);
20:     }
69:
70:  }
```

Параметризованные типы C++

В идеале переменная экземпляра, которая хранит коллекцию ссылок на другие объекты, должна иметь тип класса коллекции, который определяет все в этой коллекции в терминах параметра типа. Такой класс называется по-разному — **параметризованным типом**, **шаблоном класса** или **базовым классом**. Прежде чем параметризованный тип может использоваться, параметр типа должен быть заменен существующим классом.

Параметризованные типы не поддерживаются в Java, хотя экспериментальный компилятор Java с такой поддержкой имеется в фирме Sun. Такие типы существуют в языке C++ [27]. Рис. 12.4 показывает, как параметризованные типы могут использоваться в приложении C++ для `MovieActor`.

Рис. 12.4 изображает два параметризованных типа: `Set` и `List`. Класс `Set` определен как шаблон класса на основе класса-параметра `M`, а класс `List` — на основе класса-параметра `A`. Все элементы-переменные и элементы-функции классов `Set` и `List` должны быть определены в терминах параметров `M` и `A` соответственно.

Прежде чем такие переменные или функции могут быть использованы, параметризованные типы должны быть заменены конкретными классами. Эти классы обозначены на рис. 12.4 как «`anonymous_type`» (анонимный тип). **Классы «`anonymous_type`»** заменяют параметр типа именем класса. Функциональные возможности переменных и функций, таким образом, зависят от конкретной параметризации типа.

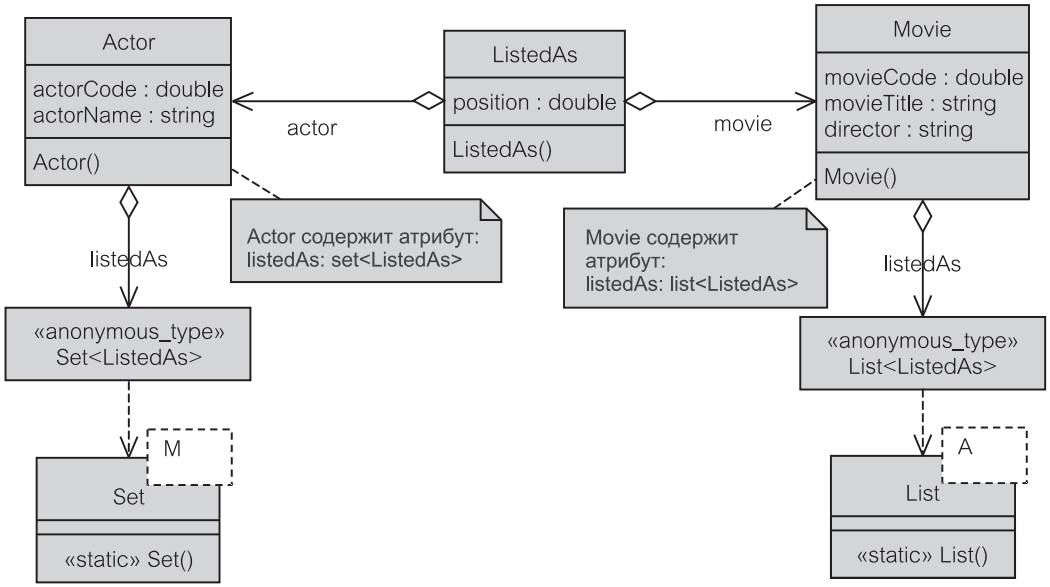


Рис. 12.4. Параметризованные типы C++

Например, `Actor` содержит переменную `listedAs`. Тип этой переменной задан как `Set<ListedAs>`. В действительности это устанавливает связь агрегирования/ассоциации от `Actor` к коллекции ссылок (`Set`), указывающих на объекты класса `ListedAs`.

Листинг 12.4 показывает синтаксис C++, соответствующий модели на рис. 12.5. Определения классов для параметризованных типов задают параметры классов, на которых они определены. Точно так же реализации всех методов, типа конструкторов `Set()` и `List()`, указывают на параметр класса. В этом случае параметризованный тип используется, чтобы инициализировать объекты коллекции, типа `Set<ListedAs> listedAs`.

Сравнение рисунков 12.2, 12.4 и связанных с ними листингов кода показывает, что использование параметризованных типов C++ лучше использования Java-интерфейса `Collection` для реализации ассоциаций с множественностью «многие». Создание новых типов, основанных на других типах, делает возможным определить переменные, имеющие тип классов коллекций, которые могут содержать только объекты определенного класса.

Листинг 12.4. Программирование параметризированных типов на языке C++

Программирование параметризированных типов на языке C++

```
template <class M> class Set
{
   public:
      static Set();
      ...
};
```

```
template <class M> Set::Set() {...}
```

```
#include "Set.h"
class Actor
{
   public:
      Actor();
   private:
      double actorCode;
      string actorName;
      Set<ListedAs> listedAs;
};
Actor::Actor() { }
```

```
template <class A> class List
{
   public:
      static List();
      ...
};
template <class A> List::List() {...}
```

```
#include "List.h"
class Movie
{
   public:
      Movie();
   private:
      double movieCode;
      string movieTitle;
      string director;
      List<ListedAs> listedAs;
};
Movie::Movie(){ }
```

```
#include "Actor.h"
#include "Movie.h"
class ListedAs
{
   public:
      ListedAs();
   private:
      double position;
      Movie movie;
      Actor actor;
};
ListedAs::ListedAs(){
   position = -1;
}
```

Рис. 12.5. Интерфейс Connection

12.1.3. Доступ к БД в Java

Как было рассмотрено в главе 10, единственным способом доступа к данным в реляционной БД является использование некоторого SQL. Приложения Java и апплеты могут осуществить доступ к БД, используя **Java Database Connectivity** (JDBC) — связь в Java с БД — или **SQL for Java** (SQLJ) — SQL для Java. JDBC и SQLJ позволяют Java вызывать встроенный SQL и вызывать хранимые процедуры (раздел 10.1.5). Стандарт JDBC является составной частью **Java Development Kit** (JDK) — комплекта разработки Java, являющегося библиотекой классов (пакет `java.sql`).

СУБД может применять технологию использования Java для кодирования хранимых процедур БД (то есть, метод Java может быть размещен в БД и запускаться оттуда). Oracle поддерживает такую технологию под именем **Java Stored Procedures** (хранимые процедуры Java) [13, 54]. Это пример среды двусторонней интеграции, где Java может вызывать хранимую процедуру и наоборот. Технология имеет ограничения и не обсуждается далее в этой книге.

Доступ к БД в Java может также быть получен из компонентов Java многократного использования типа **JavaBeans**, **Enterprise JavaBeans** (EJB) или

Business Components for Java (BC4J) [13, 54]. Последнее является технологией Oracle. JavaBeans позволяет использовать вызовы JDBC в методах Bean-компонентов. EJB позволяет создавать компоненты, которые работают в среде на стороне сервера. EJB представляет средний уровень трехуровневой или многоуровневой структуры (раздел 9.2.2). EJB-компонент может содержать JDBC- и SQLJ-операторы. BC4J является составной частью интегрированной среды программирования Oracle JDeveloper. В то время как EJB дает сервисы, характерные для БД, чтобы обеспечить постоянство, транзакции и безопасность, BC4J концентрируется на инкапсуляции бизнес-логики в отдельный уровень. В Oracle BC4J-компонент может быть развернут как EJB-объект сервера.

Java был разработан для сред Web-технологии. Используемые технологии включают Java-апплеты, Java-сервлеты и Java Server Pages (JSP) — серверные страницы Java[2]. Как и любая Java-технология, технологии для Web-сред имеют возможность работы с БД через средства, которые рассматриваются в этом разделе.

Сравнение JDBC и SQLJ

JDBC и SQLJ — два стандартных способа включить SQL-операторы в код Java, чтобы связаться с БД. SQLJ запускается поверх JDBC и требует транслятора, который заменяет вложенные SQL-операторы на запросы к SQLJ-среде (библиотеке) времени выполнения. Эта библиотека времени выполнения использует JDBC.

JDBC формирует API-функции, чтобы связаться с БД, и использует JDBC-драйвер, чтобы соединиться с БД. Имеются четыре типа JDBC-драйверов [111]:

1. JDBC/ODBC-мост плюс ODBC-драйвер — это требует Open Database Connectivity (ODBC) — открытое соединение с БД — драйвер, который должен быть установлен на компьютере клиента перед тем, как будет использоваться JDBC для доступа к БД.
2. Native API (собственная API-библиотека) — определенный продавцом драйвер, который конвертирует запросы JDBC в обращения к API-библиотеке, специфичной для СУБД, типа Oracle Call Interface (OCI) — интерфейс вызовов в Oracle; это требует, чтобы на машине клиента был загружен некоторый двоичный код СУБД.
3. Open Protocol Net (открытая сеть протоколов) — данный протокол переводит запросы JDBC в независимый от БД сетевой протокол, обеспечивающий доступ ко многим различным БД с того же самого клиента Java; трансляция к протоколу СУБД выполняется на сервере, обеспечивая прозрачную для клиента связь.
4. Native Protocol Net (собственная сеть протоколов) — данный протокол подобен типу 3 драйвера, но использует свой собственный сетевой протокол для доступа к специфичной БД, определенной продавцом.

[2] Апплет — программа, обычно выполняемая на локальном компьютере (клиенте), сервлет — программа, выполняемая на сервере, а ее результаты передаются клиенту. (*Прим. перев.*)

Как только Java-программа загрузит драйвер БД, будет установлена связь с БД (объект `Connection` — связь). Затем могут быть созданы операторы и запросы JDBC или SQLJ к хранимым процедурам (объекты `Statement` — оператор), которые можно выполнить с помощью установленной связи. Набор результатов, возвращаемый после выполнения запроса и хранимых процедур, требует использования объекта `ResultSet` (набор результатов). Когда обработка будет закончена, программа закрывает объекты `Statement`, `ResultSet` и `Connection`.

Коды JDBC и SQLJ могут быть смешаны в программе, используя одну и ту же связь и наборы результатов. SQLJ состоит из SQLJ-транслятора и SQLJ-среды времени выполнения. Транслятор является препроцессором, который конвертирует исходную программу SQLJ (с расширением `.sqlj`) в источник Java (с расширением `.java`) перед компиляцией источника для создания файла класса (с расширением `.class`). SQLJ-среда времени выполнения состоит из SQLJ-библиотеки, которая выполняет на чистом языке Java SQLJ-операторы, входящие в программу. Как правило, SQLJ-среда времени выполнения использует JDBC-драйвер для доступа к БД.

SQLJ — более высокий уровень среды программирования, нежели JDBC. Обычно здесь можно программировать те же самые функциональные возможности с меньшим количеством кода. Синтаксис и семантика SQL-операторов, используемых в программе, проверяются и оптимизируются во время компиляции. Окончательный код более точен и может обеспечивать лучшее выполнение функций. Однако JDBC имеет преимущество в непосредственной поддержке конструирования и выполнения динамических SQL-запросов. В динамическом SQL-запросе детали объектов БД, типа имен таблиц и столбцов, могут быть неизвестны во время компиляции, и их следует задать программе во время выполнения.

Установление связи с БД

Установление связи с БД включает использование JDBC-класса `DriverManager` (управление драйвером) и JDBC-интерфейса `Connection` (из библиотеки `java.sql`). Имеется несколько способов загрузки драйвера с использованием JDBC/ODBC-моста. Проще всего использовать метод `Class.forName` (строка 16 в листинге 12.5). Метод берется из пакета `java.lang.Class`. Он неявно создает экземпляр драйвера и регистрирует его с объектом `DriverManager`.

Листинг 12.5. Организация связи с БД

Организация связи с БД

```
11:  public class Connection {
12:     private java.sql.Connection conn;
13:
14:  public Connection() throws Exception {
15:     //установление связи
16:     Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
17:      conn =
18:         DriverManager.getConnection(
19:            "jdbc:oracle:thin:@localhost:1521:oracle8i",
20:            "pse1",
21:            "pse1");
22:      }
136: }
```

Объект связи (conn — строка 17) создается вызовом метода getConnection() (получить соединение) объекта DriverManager (строка 18). Метод getConnection() требует три параметра: строку связи URL (Uniform Resource Locator — унифицированный указатель информационного ресурса в Интернете), указывающую на экземпляр БД (строка 19), пользовательское имя БД (строка 20) и пароль БД (строка 21).

Рис. 12.5 показывает методы интерфейса Connection. Интерфейс включает методы, возвращающие три вида интерфейсов, которые могут использоваться для выполнения SQL-операторов. Эти интерфейсы — Statement (оператор), PreparedStatement (подготовленный оператор), и CallableStatement (вызываемый оператор).

Выполнение SQL-операторов

Чтобы выполнить SQL-запрос к БД, необходимо создать объект оператора. Объект оператора должен быть подключен к одному из трех JDBC-интерфейсов: Statement, PreparedStatement или CallableStatement.

Объект Statement используется, чтобы выполнить простые SQL-операторы, которые не имеют параметров. Объект PreparedStatement может использоваться, чтобы неоднократно выполнить тот же самый SQL-оператор, но каждый раз с различными условиями поиска или значениями. Объект CallableStatement используется, чтобы вызвать хранимые процедуры БД.

Чтобы загрузить объекты-сущности в память, приложению MovieActor («Фильм-Актер») требуется запрос к БД и создание объектов-сущностей по набору результатов, возвращенных запросами. Для простоты приложение MovieActor состоит только из пяти классов: трех классов сущностей, класса MovieSearcher (поиск фильма) и класса Connection. Класс MovieSearcher начинает обработку и отображает результаты доступа к БД на экране. Класс Connection взаимодействует с БД.

Рис. 12.6 (так же как и рис. 2.13) представляет диаграмму последовательности действий для процесса заполнения кэша сущностей после извлечения данных из БД. Объект searcher:MovieSearcher инициализирует объект conn:Connection и затем посылает ему сообщение readAll() (читать все). Метод readAll() конструирует строку SQL-запроса и передает его методу query() (запрос).

Метод query() посылает сообщение createStatement() (создать оператор) объекту conn. Это показано на рис. 12.6 как сообщение к JDBC-интерфейсу Connection. Сообщение createStatement() возвращает объект Statement. Сообщение executeQuery() (выполнить за-

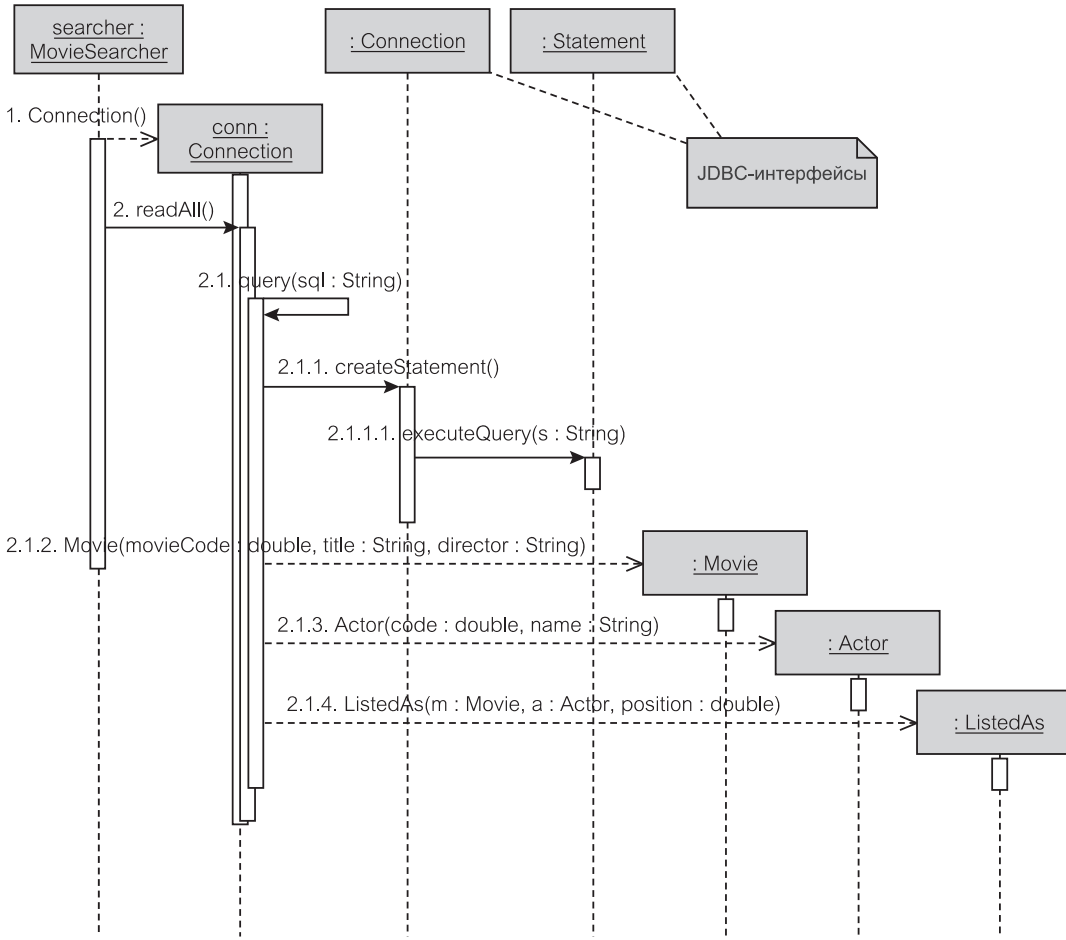


Рис. 12.6. Диаграмма последовательности действий для доступа к БД и создания типов сущностей

прос) объекта `Statement` формирует объект `ResultSet`, который возвращается методу `readAll()`.

Объект `ResultSet` (реализующий JDBC-интерфейс `ResultSet`) не показан на рис. 12.6. Он представляет набор записей, полученных в результате выполнения запроса. Объект позволяет проходить по записям и обращаться к значениям данных конкретных элементов (столбцов) в записи. Поскольку SQL-запрос является запросом объединения, который восстанавливает воедино все записи из трех таблиц в БД `MovieActor`, просмотр `ResultSet` может привести к созданию объектов `Movie` (фильм), `Actor` (актер) и `ListedAs` (перечисление) в кэше памяти приложения.

Листинги 12.6 и 12.7 представляют детали кода, которые диаграмма последовательности действий на рис. 12.6 не могла охватить. Код для класса `MovieSearcher` в листинге 12.6 не требует дальнейшего объяснения. Наиболее интересный метод — `readAll()` в классе `Connection` листинга 12.7.

Листинг 12.6. MovieSearcher.java

MovieSearcher.java

```
8:   public class MovieSearcher {
13:     private Connection conn;
14:     private Collection listedAs;
15:
16:     public MovieSearcher() throws Exception {
17:        conn = new Connection();
18:        retrieveAll();
19:     }
204:
205:    public static void main(String args[]) {
207:          MovieSearcher searcher = new MovieSearcher();
249:    }
250: }
```

Метод `readAll()` представлен строками 23–42 в листинге 12.7. Он сначала создает коллекцию `ArrayList` (список-массив) в строке 24. SQL-оператор помещается в строку (строка 26) и затем передается в качестве аргумента методу `query`. Метод `query` показан в строках 45–48. Он создает объект, соответствующий интерфейсу `Statement`, который может использоваться для запуска SQL-запросов в связанной с ним БД. Объект `st` оператора создается, выполняя метод `createStatement()` объекта `conn` (строка 46). На строке 47 SQL-запрос выполняется в объекте `st` и полученные из БД записи размещаются в экземпляре объекта `ResultSet`, называющемся `rs` (строка 25).

Листинг 12.7. Connection.java

Connection.java

```
11:  public class Connection {
12:    private Java.sql.Connection conn;
13:
14:    public Connection() throws Exception {
15:       //установление связи
16:       Class.forName("oracle.jdbc.driver.OracleDriver");
17:       conn =
18:          DriverManager.getConnection(
19:             "jdbc:oracle:thin:@localhost:1521:oracle8i",
20:             "pse1",
21:             "pse1");
22:    }
23:    public Collection readAll() throws Exception{
24:       Collection c = new ArrayList();
```

```
25:       ResultSet rs =
26:         query("SELECT * FROM movie m, actor a, listed_as l
                      WHERE m.movie_code = l.movie_code
                         AND l.actor_code = a. actor_code");
27:       while (rs.next()) {
28:         double movieCode = rs.getDouble("movie_code");
29:         String movieTitle = rs.getString("movie_title");
30:         String director = rs.getString("director");
31:
32:         double actorCode = rs.getDouble("actor_code");
33:         String actorName = rs.getString("actor_name");
34:
35:         double position = rs.getDouble("position");
36:
37:         Movie m = new Movie(movieCode, movieTitle, director);
38:         Actor a = new Actor(actorCode, actorName);
39:         ListedAs la = new ListedAs(m, a, position);
40:         c.add(la);
41:        }
42:        return c;
43:     }
44:
45:     private ResultSet query(String sql) throws Exception {
46:        Statement st = conn.createStatement();
47:        return st.executeQuery(sql);
48:     }
136: }
```

В строке 27 метод `next()` (следующий) объекта `ResultSet` организует циклический просмотр записей, по одной записи одновременно. Данные выбираются в операторах присваивания (строки 28–35). Из данных каждой приводимой записи в кэше памяти инициализируются три объекта-сущности: `Movie`, `Actor` и `ListedAs` (строки 37–39).

Каждый объект `ListedAs` содержит три элемента данных. Два из них — объекты `Movie` и `Actor`. Фактически коллекция объектов `ListedAs` содержит все записи БД `MovieActor`. Объекты помещены в коллекцию с помощью метода `add()` (добавить) в строке 40.

Вызов хранимых процедур и функций

Метод `preparedCall()` (подготовить вызов), определенный в интерфейсе `Connection` (рис. 12.5), возвращает объект `CallableStatement`, который может быть использован, чтобы вызвать хранимую процедуру или функцию. Листинги 12.8 и 12.9 иллюстрируют код, необходимый, чтобы вызвать, выполнить и отобразить результаты, возвращенные хранимой функцией `string_search` (поиск строки), описанной в главе 10 в листинге 10.8.

Метод `displaySearchMoviesByStoredFunction()` (отобразить поиск фильмов хранимой функцией) класса `MovieSearcher` в листинге 12.8 просит,

чтобы объект conn выполнил метод searchMoviesByStoredFunction() — поиск фильмов хранимой функцией (строка 62). Строка поиска («vamp» — вамп) жестко закодирована в вызове из главного метода displaySearchMoviesByStoredFunction() (строка 244).

Листинг 12.8. displaySearchMoviesByStoredFunction() — метод в MovieSeacher

displaySearchMoviesByStoredFunction() — метод в MovieSeacher

```
8:   public class MovieSearcher {
59:
60:     public void displaySearchMoviesByStoredFunction(String
        str)
              throws Exception{
61:        System.out.println("Searching movies on specified
                                                 string:" +str);
62:        Collection c =
                  conn.searchMoviesByStoredFunction(str);
63:        Iterator it = c.iterator();
64:        while(it.hasNext()){
65:           displayMovie((Movie) it.next());
66:           System.out.println();
67:        }
68:     }
194:    private void displayMovie(Movie movie) {
195:      System.out.println("Movie Code: " +
                  movie.getMovieCode());
196:      System.out.println("Movie Title: " +
                  movie.getMovieTitle());
197:      System.out.println("Movie Director: " +
                  movie.getDirector());
198:    }
205:    public static void main(String args[]) {
244:      searcher.displaySearchMoviesByStoredFunction("vamp");
249:    }
250: }
```

Листинг 12.9. searchMoviesByStoredFunction() — метод в Connection

searchMoviesByStoredFunction() — метод в Connection

```
11:   public class Connection {
115:
116:     public Collection searchMoviesByStoredFunction(
                        String searchStr) throws Exception{
117:        Collection c = new ArrayList();
118:
```

```
119:        CallableStatement st = conn.prepareCall(
                "{ ? = call MovieSearch.string_search (?)}");
120:        st.registerOutParameter(1, OracleTypes.CURSOR);
                                       //мы ожидаем курсор
121:        st.setString(2,searchStr);
122:        st.execute();
123:        ResultSet rs = (ResultSet) st.getObject(1);
124:        while (rs.next()) {
125:           double movieCode = rs.getDouble("movie_code");
126:           String movieTitle = rs.getString("movie_title");
127:           String director = rs.getString("director");
128:           Movie m = new Movie(movieCode, movieTitle,
                                                  director);
130:           c.add(m);
131:        }
132:        closeResult(rs);
133:
134:        return c;
135:     }
136: }
```

Объект `MovieSearcher` ожидает `Collection` (коллекция) объектов `Movie`, которая возвращается как результат обращения в строке 62. После этого используется объект `Iterator` — итератор (рис. 12.3), чтобы просмотреть объекты `Movie` (строки 63–67) и запросить метод `displayMovie()` — отобразить фильм (строки 194–198), для выполнения отображения результатов на экране. Рис. 12.7 — образ экрана, который показывает результат поиска фильмов, содержащих строку «`vamp`» в названии фильма (сравните выполнение в листинге 10.7).

Чтобы выполнить сервис обнаружения строки «`vamp`» в названиях фильмов (листинг 12.9), объект `Connection` должен активизировать хранимую функцию `string_search` (листинг 10.8). Сначала инициализируется `ArrayList`, чтобы зафиксировать `Collection`, полученную в результате поиска (строка 117). Метод `prepareCall()` (подготовить вызов) вызывает хранимую функцию и квалифицирует запрос именем пакета, содержащего БД (строка 119). Знак вопроса отмечает необходимость задания парамет-

Рис. 12.7. Отображение результатов поиска строки с использованием хранимой функции

ров — они известны как *присваиваемые параметры* или *«заполнители»*. Первый знак вопроса перед знаком равенства представляет возвращаемую функцией величину. Знак вопроса в конце вызова означает входной параметр `searchStr` (поиск строки).

Вызов хранимой функции создает объект класса `CallableStatement`. Возвращаемая хранимой функцией величина регистрируется в этом объекте как выходной параметр, используя метод `registerOutParameter()` (регистрация выходного параметра). Поскольку возвращаемая величина (обозначенная цифрой 1), как ожидается, будет коллекцией объектов, используется механизм курсора БД (строка 120). Входной параметр (обозначенный цифрой 2) связан с объектом `Connection` посредством метода `setString()` — задать строку (строка 121). Это завершает конструирование объекта `CallableStatement`, после чего вызывается метод `execute()`, чтобы активизировать хранимую функцию `string_search` (строка 122).

В строке 123 объект `CallableStatement` присваивается объекту `ResultSet`. Это присваивание позволяет использовать метод `getObject()` (получить объект) вместо метода `getCursor()` (получить курсор). Программа может теперь просматривать объекты `ResultSet` и добавлять их в `Collection` (строка 130), которая возвращается к `MovieSearcher` (строка 62 в листинге 12.8).

12.2. Управляемая тестированием разработка

Управляемая тестированием разработка — практика, которую популяризирует **быстрая разработка программного обеспечения** [66], главным представителем которой является **eXtreme Programming** (XP) [71]. Однако управляемая тестированием разработка может быть и должна быть охвачена другими процессами разработки, включая Unified Process (UP) [53].

Управляемая тестированием разработка означает и управляемое тестированием программирование и управляемое тестированием проектирование. Управляемая тестированием разработка побуждает разработчиков писать испытательные спецификации и программы, прежде чем завершить проект и перед началом «деления кода приложения». Код приложения пишется как ответ на испытательный код, но не наоборот.

Если испытательный код написан до того, как будет написан код приложения, то ясно, что одна тестовая программа будет при запуске выдавать ошибку. Это основной момент. Испытательный код пишется, чтобы искать недостатки в коде приложения. Код приложения должен быть реализован так, чтобы можно было при его последующем запуске использовать тестирование. Это подтвердит обеспечение требуемых функциональных возможностей. Суть управляемой тестированием разработки заключается в том, чтобы обеспечить именно разработку ПО, но не его проверку.

Управляемая тестированием разработка — итеративный и пошаговый процесс, сочетаемый с написанием кода приложения. Идея состоит в том, чтобы сначала написать испытательный код для небольшой задачи, для которой пишется код приложения, достаточный для запуска тестирования. Затем

испытательный код расширяется, после чего пишется дополнительный код приложения, и т. д.

Управляемая тестированием разработка требует, чтобы набор испытательных единиц был запрограммирован так, чтобы можно было использовать принцип **«белого ящика» для тестирования** конкретных классов и основного взаимодействия между классами. Обычное понимание принципа «белого ящика» для тестирования заключается в том, что это «тестирование на основе кода» [61]. Идея состоит в том, чтобы изучить код, понять его и проверять (реализовывать) возможные пути выполнения. Однако в управляемой тестированием разработке реальный код пока не существует. Код можно только предполагать на основе существующего проекта, и пути могут быть рассмотрены «всухую» (то есть управлять кодом мысленно, без использования компьютера).

Управляемая тестированием разработка начинается как тестирование, основанное на отдельных **испытательных (программных) компонентах**. Понятие испытательного компонента имеет две интерпретации: испытательный компонент можно рассматривать или как цель испытания, или как ресурс испытания. Целью испытательного компонента обычно является отдельный класс в коде приложения, но это могут также быть и лишь отдельные методы класса, и несколько взаимодействующих классов. Как ресурс, испытательный компонент представляет часть кода — класс или несколько классов, которые выполняют тестирование. Методы в пределах класса испытательного компонента называются **контрольными примерами**.

Контрольные примеры могут быть объединены в **тестовые наборы**, чтобы запускать коллекцию контрольных примеров для тестирования нескольких классов или даже всей системы. Интересный вопрос: где поместить испытательный код по отношению к коду приложения, который будет проверяться? Имеется множество возможностей.

Как рассмотрено Эккелем [25], контрольный пример может быть написан в пределах метода `main()` каждого класса, который будет тестироваться (то есть в пределах каждого испытательного компонента). Как сказано в разделе 11.2.3, один из методов `main()` инициализирует первый объект в приложении и начинает работу программы. Это метод `main()` класса, помещаемого в командной строке. Методы `main()` в других классах могут успешно использоваться для тестирующих компонентов.

Альтернативой к размещению контрольного примера в пределах метода `main()` может быть размещение их в статическом внутреннем классе в пределах класса, который будет тестироваться [25]. **Внутренний класс** в Java — это определение класса, размещенное внутри определения другого класса (**внешнего класса**). Так как внутренний объект может иметь свободный доступ ко всем элементам включающего его внешнего объекта, он может успешно использоваться для хранения контрольного примера. Во время компиляции внутренний класс формируется как отдельный класс. Если внешний класс имеет имя `X`, а внутренний класс — имя `Y`, то сформированный класс будет называться `X$Y`. Класс `X$Y` может использоваться для тестирования, но он может быть исключен из любого кода, передаваемого пользователям.

Использование метода `main()` или внутреннего класса — жизнеспособные подходы к тестированию отдельных компонентов, но они не обеспечивают шаблона для полного тестирования. Шаблон целиком лежит на обязанности разработчиков и программистов. Всесторонний и систематический подход к управляемой тестированием разработке лучше обеспечивает базовые шаблоны тестирования, которые являются библиотеками классов и интерфейсов, предназначенных для облегчения реализации тестирования. JUnit [49] — шаблон Java, рекомендованный для быстрой разработки ПО и для XP, в частности.

12.2.1. Шаблон JUnit

JUnit определяет шаблон для создания контрольных примеров, тестовых наборов и инструментальных средств при управлении ими. Основные классы шаблона JUnit — `TestCase` (контрольный пример) и `TestSuite` (тестовый набор). Оба реализуют интерфейс, называемый `Test` (тестирование). `Test` определяет методы запуска теста.

`TestSuite` — сложный класс, который может содержать (концептуально) более одного `Test`. Поскольку `Test` — только интерфейс, реализованный или как `TestCase`, или как `TestSuite`, объект `TestSuite` может содержать несколько объектов `TestCase` и/или другой объект `TestSuite`. Таким образом, объект `TestSuite` может быть включен в более сложные объекты `TestSuite`, которые в свою очередь могут быть сами включены и так далее рекурсивно.

Рис. 12.8 изображает общую структуру JUnit-шаблона. Модель использует один из наиболее фундаментальных и наиболее полезных паттернов в объек-

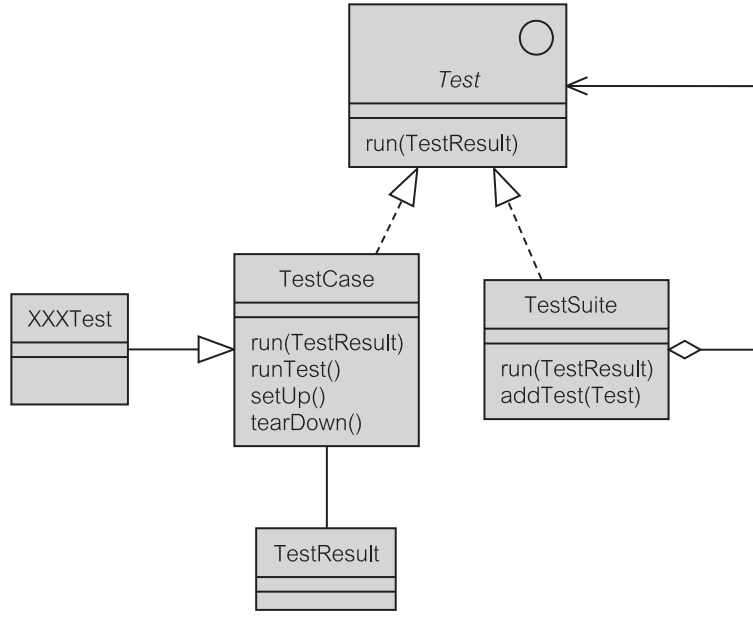


Рис. 12.8. JUnit в терминах паттерна Компоновщик

тно-ориентированном проектировании и программировании — **паттерн Компоновщик** (Composite) [32]. Паттерн позволяет рекурсивно составлять объекты. Отдельные объекты `TestCase` и объединения (агрегаты) их (объекты `TestSuite`) обращаются друг к другу одинаковым образом. `Test` является интерфейсом, который представляет и индивидуальные объекты, и объединения их. Объекты-клиенты, типа `XXXTest`, управляют конкретными объектами в объединении через интерфейс `Test`.

Испытательный компонент может быть реализован как подкласс, который расширяет `TestCase`. Если границы испытательного компонента — класс `XXX`, то испытательный класс может быть назван `XXXTest`. `XXXTest` определяется как подкласс `TestCase`, который добавляет различные испытательные методы к методам, унаследованным от `TestCase`. `XXXTest` обычно помещается в тот же самый пакет, что и класс `XXX`.

Тестирование класса `XXX` предполагает, что объекты этого класса созданы во время выполнения теста и взаимодействуют с процессом исполнения теста. Эти объекты — известные как **закрепление** испытательного компонента — устанавливают контекст для класса `XXXTest`. Объекты закрепления обеспечивают цель для тестируемого кода. Тестируемый код «использует» объекты закрепления и формирует результаты тестирования.

Метод `setUp()` (установить) используется для инициализации объектов закрепления, а метод `tearDown()` (демонтировать) используется, чтобы удалить закрепление в конце каждого контрольного примера. Если испытательный компонент запускает несколько контрольных примеров, то методы `setUp()` и `tearDown()` вызываются несколько раз, чтобы каждый контрольный пример начинался с «чистого листа».

Метод `runTest()` (запустить тест) используется, чтобы управлять испытательным компонентом, вызывая желательные контрольные примеры (например, в пределах класса `XXXTest`). JUnit поддерживает статические и динамические способы запуска контрольных примеров. Статический способ имеет два варианта. Первый вариант статического способа показан в листинге 12.10. Используемый одним из нескольких классов `TestRunner` (запуск теста) JUnit-шаблона, метод `runTest()` вызывает контрольные примеры, перечисленные в нем.

Листинг 12.10. Метод runTest() в классе TestCase

```
runTest() — метод в XXXTest

public class XXXTest extends TestCase {
   public XXXTest(String testName) {
      super(testName);
   }
   public void testSomething() {
   ...
   }
   public void testSomethingElse() {
   ...
```

```
    }
    public void runTest() {
       testSomething();
       testSomethingElse();
    }
}
```

Листинг 12.11. Метод runTest() в анонимном внутреннем классе

```
runTest() в анонимном внутреннем классе класса YYYTestUnit

public class YYYTestUnit extends TestCase {
   public static Test suite () {
      TestSuite suite = new TestSuite();
      TestCase test = new XXXTest("my test") {
         public void runTest() {
            testSomething();
            testSomethingElse();
            testSomethingDifferent();
         }
         public void testSomethingDifferent(){
            ...
         }
      };
      suite.addTest(test);
      suite.addTest(new YYYTestUnit("another test"));
      return suite;
   }
   public static void main(String args[]){
      junit.textui.TestRunner.run(suite());
      //junit.swingui.TestRunner.run(YYYTestUnit.class);
   }
}
```

Второй вариант статического способа использует преимущество анонимного внутреннего класса. **Анонимный внутренний класс** — внутренний класс без имени, который представлен возвращаемым значением, полученным от создания нового объекта некоторого класса. Считая, что метод `runTest()` не включен в класс `XXXTest`, как в листинге 12.10, метод `runTest()` может быть определен в анонимном внутреннем классе, что показано полужирным шрифтом в листинге 12.11. Строка типа «my test» (мой тест) требуется, чтобы идентифицировать тестирование, если оно выявит ошибку.

Обратите внимание, что анонимный внутренний класс вызывает контрольные примеры явно. Он не должен вызывать все контрольные примеры, определенные в классе `XXXTest`, и может вводить и реализовывать новые кон-

трольные примеры типа `testSomethingDifferent()` (тестирование чего-то другого) в листинге 12.11. Обратите также внимание, что испытательный компонент может быть запущен в тексте пользовательского интерфейса (в консольном окне) или в GUI-окне (используя библиотеку визуальных компонентов swing).

Динамический способ подготовки испытательного компонента и запуск его использует отражение Java, чтобы реализовать метод `runTest()` во время выполнения. **Отражение** — механизм, позволяющий определить информацию о классе и методе во время выполнения и обеспечивающий возможность создавать объекты (и выполнять их методы) классов, которые неизвестны во время компиляции.

Листинг 12.12 показывает, как находится и используется динамически испытательный метод. Динамический способ предполагает, что строка аргументов, передаваемая конструктору объекта `XXXTest`, является именем контрольного примера, который нужно вызвать с методом `runTest()`.

Листинг 12.12. Метод runTest(), реализованный динамически

runTest(), реализованный динамически

```
public static Test suite() {
   TestSuite suite = new TestSuite();
   TestCase test1 = new XXXTest("testSomething");
   TestCase test2 = new XXXTest("testSomethingElse");
   suite.addTest(test1);
   suite.addTest(test2);
   return suite;
}
public static void main(String args[]){
   junit.textui.TestRunner.run(suite());
}
```

JUnit обеспечивает визуальный интерфейс, чтобы запускать контрольные примеры и тестовые наборы. Интерфейс упрощает использование тестов, он показывает ход тестирования и указывает успешные испытания зеленой полосой индикатора выполнения. Если тестирование выявит ошибку, будет показана красная полоса индикатора выполнения, а имена неудавшихся испытаний будут внесены в список.

12.2.2. Управляемая тестированием разработка в управлении электронной почтой

Как указано Мартином [66], управляемая тестированием разработка создает проект, приводящий к ПО, которое легко вызывать и тестировать. Эта цель достигается путем минимизации соединения между классами. В управляемой тестированием разработке основной механизм разъединения классов — это окружение их интерфейсами (раздел 9.1.7).

В итерации 1 учебного примера ЕМ разъединение классов является главной мотивацией структурного шаблона PCMEF+ (раздел 9.2.2). Сам шаблон до использования управляемой тестированием разработки делает проект легко вызываемым и тестируемым. Для этого интерфейсам дают надлежащее содержание. Большинство интерфейсов сгруппировано в пакете `acquaintance` (знакомство), который используется, чтобы нарушить циклические зависимости и гарантировать соблюдение принципа нисходящей зависимости (раздел 9.2.2).

Управляемая тестированием разработка в итерации 1 предполагает существование *первоначального структурного проекта*. Она не является заменой отсутствующей структуры. Кроме того, в учебных целях итерация 1 минимизирует число классов в PCMEF-пакетах. Цели управляемой тестированием разработки в итерации 1 — внутреннее структурирование классов, разработка методов и задание передачи сообщений для обеспечения взаимодействий. Эта разработка выполняется одновременно с проектированием классов и взаимодействий (глава 11), и она управляет программированием.

Взаимодействие «Регистрационное имя» — одно из девяти взаимодействий, определенных для итерации 1 (раздел 11.4). Будучи относительно простым взаимодействием, оно может использоваться, чтобы объяснить процесс, философию и результаты управляемой тестированием разработки.

PCMEF-структура требует, чтобы передача сообщения между уровнями была *нисходящей* от более высоко расположенного пакета к более низко расположенному пакету. Любая *восходящая* связь должна пройти через интерфейс или передать событие. В сценарии «Регистрационное имя» пользователя просят задать регистрационное имя. Это легко выполняется следующим кодом в классе `PMain` (класс «основной» пакета `presentation`):

```
PConsole console;
console.displayLogin();
```

Управляемая тестированием разработка, подобно другим видам тестирования, не предусматривает тестирование всего и все время. Нет смысла писать контрольные примеры, которые бесполезны. Не так уж много видов тестирования может быть выполнено в классе `PConsole` (класс «консоль» пакета `presentation`) при реализации регистрационного имени. `PConsole` просто делегирует сервис, относящийся к регистрационному имени, более низкому уровню — классу `CActioner` (класс «исполнитель» пакета `control`).

Учебный пример ЕМ является СУБД-приложением. Проверка регистрационного имени требует организации связи с БД, а также проверки, что пользователь является служащим в таблице `Employee` (служащий). Класс `PConsole` получает пользовательское имя и пароль и передает эту информацию классу `CActioner` следующим образом:

```
class PConsole{
  CActioner actioner;
  void displayLogin(){
    ...
    actioner.login(username, password);
  }
}
```

Вышеупомянутый фрагмент кода ясно показывает, что имеется потребность протестировать реализацию регистрационного имени класса CActioner. CActioner — конкретный класс, который представляет уровень управления (согласно паттерну Фасад). Контрольный пример для этого представлен в листинге 12.13.

Листинг 12.13. Метод testLogin() в CActionerTest

Метод testLogin() в CActionerTest

```
void testLogin(){
   CActioner actioner = new CActioner();
   try{
      IAEmployee emp = actioner.login("user","passwd");
      assertEquals(emp.getLoginName(), "user");
   }catch(Exception exc){
      fail("Exception occurs during login");
   }
}
```

JUnit предоставляет различные методы assert (контроль), чтобы получить результаты тестирования, типа assertTrue() (контроль истинности), assertSame() (контроль одного и того же значения), assertNull() (контроль пустого значения), assertEquals() (контроль равенства). Метод assertEquals() наиболее полезен. Он используется в листинге 12.13 для контроля того, что пользовательское имя, переданное методу login() (регистрационное имя), совпадает с регистрационным именем того пользователя, который находится в таблице Employee (служащий) и получен вызовом метода getLoginName() (получить регистрационное имя) объекта EEmployee (класс «служащий» пакета entity).

Метод testLogin() (контроль регистрационного имени) имеет некоторое намерение по отношению к CActioner. CActioner должен реализовать это намерение. Такая операция иногда называется **программированием намерений** [66]. Ясно, что метод класса CActioner, который анализирует регистрационное имя, должен возвратить объект EEmployee (интерфейс IAEmployee — интерфейс «служащий» пакета acquaintance). Это означает, что регистрационное имя служащего совпадает с введенным пользовательским именем. Возвращаясь назад, можно сказать, что программирование намерений также показывает, как должны быть построены методы клиента (PConsole). Фрагмент окончательного кода в CActioner показан в листинге 12.14.

Листинг 12.14. Метод login() в CActioner

Метод login() в CActioner

```
private IAEmployee emp;
...
public Object login( String username, String passwd ) {
   emp = broker.login( username, passwd );
}
```

CActioner теперь может возвратить объект EEmployee, но выполнение сервиса login() делегируется далее объекту MBroker (класс «посредник» пакета mediator). Объект emp (служащий) — представление конкретного объекта класса EEmployee, но на этой стадии нет никакой необходимости или причины заняться конкретной реализацией объекта класса EEmployee. Поэтому метод login() использует вместо этого интерфейс IAEmployee. Использование интерфейса достаточно для вызова assertEquals() (контроль совпадения) в листинге 12.13.

Метод testLogin() (контроль регистрационного имени) в листинге 12.13 помогает в реализации CActioner, но это все еще не столь полезно для проведения регулярных испытаний объекта класса CActioner. CActioner делегирует сервис login() классу MBroker, и именно MBroker возвращает объект emp классу CActioner. В большинстве случаев это более практично для тестирования его источника.

Листинг 12.15 представляет метод testLogin(), реализованный в классе MBrokerTest (тестирование посредника). Код позволяет проверять успех и неудачу операции с помощью операторов блокировки/разблокировки в строках 39 и 43. Предполагая, что пользовательское имя и пароль являются «pse1» и «pse1», разблокировка в строке 39 и одновременная блокировка в строке 43 приведет к успешному выполнению тестирования.

Листинг 12.15. Метод testLogin() в MBrokerTest

Метод testLogin() в MBrokerTest

```
35:     public void testLogin(){
36:        System.out.println(«Mediator: Login Testing»);
37:        try{
38:           MBroker broker = new MBroker();
39:           //IAEmployee emp = broker.login("pse1","pse1");
43:           IAEmployee emp = broker.login("wrong","wrong");
44:           assertNotNull(emp);
45:           assertEquals(emp.getLoginName(),"pse1");
46:        }catch(Exception exc){
47:           exc.printStackTrace();
48:           fail("Fail to login:"+exc.getMessage());
49:        }
50:     }
```

Программирование намерений, основанное на контрольном примере, в листинге 12.15 может дать метод login(), показанный в листинге 12.16. Код четко показывает необходимость соединения с БД, чтобы получить информацию о служащем. FConnection (класс «соединение» пакета foundation) управляет связью, а FReader (класс «чтение» пакета foundation) осуществляет поиск. Необходимы отдельные контрольные примеры, чтобы проверить эти два класса. Поиск объекта emp выполняется с помощью сообщения, содержащего запрос, который показан в строках 83–85.

Найденный набор результатов размещается в объекте `emp` (строка 86). Метод `mapEmployee()` (поместить информацию о служащем) — единственный метод, который знает, как создать объект `EEmployee`.

Листинг 12.16 не показывает, как инициализируются объекты `FConnection` и `FReader`. Формирование экземпляра может быть сделано в пределах специального контрольного примера для `MBroker`, наподобие `MBrokerTest`. Однако формирование экземпляра может также быть сделано как часть реализации тестового набора, в котором `MBrokerTest` является всего лишь одним из многих контрольных примеров. Рис. 12.9 показывает окно `TestRunner` (запуск теста), выполняющее `EMSSuite` (отображение набора). Образ экрана был взят, когда `TestRunner` сообщил об отказе метода `testLogin()`, показанного в листинге 12.15.

Выполнение `EMSSuite` на рис. 12.9 должно использовать дополнительные контрольные примеры, которые будут включать объекты `FConnection` и `FReader`. Конструирование таких контрольных примеров показывает тесную связь этих двух классов. Экземпляр `FReader` может быть получен только через экземпляр `FConnection`. Это приемлемо, по крайней мере, в настоящее время, потому что данные объекты принадлежат к одному и тому же пакету (`foundation` — основание).

Листинг 12.16. Метод login() в MBroker

Метод login() в MBroker

```
76:     public IAEmployee login(String username, String passwd)
        {
77:        try {
78:           if (!connection.isConnected()) {
79:              //ошибка соединения
80:              if (!connection.connect(username, passwd))
81:                 return null;
82:           }
83:           java.sql.ResultSet rs =
84:              reader.query(
85:             "select * from employee where login_name = '" +
                                              username + "'");
86:           IAEmployee emp = mapEmployee(rs);
87:           reader.closeResult(rs);
88:           return emp;
89:        } catch (Exception exc) {
90:        }
91:        return null;
92:     }
```

Листинги 12.17 и 12.18 показывают контрольные примеры в `FConnectionTest` (класс «тестирование соединения» пакета `foundation`) и `FReaderTest` (класс «тестирование чтения» пакета `foundation`), которые, когда они объединены с методом `testLogin()` в

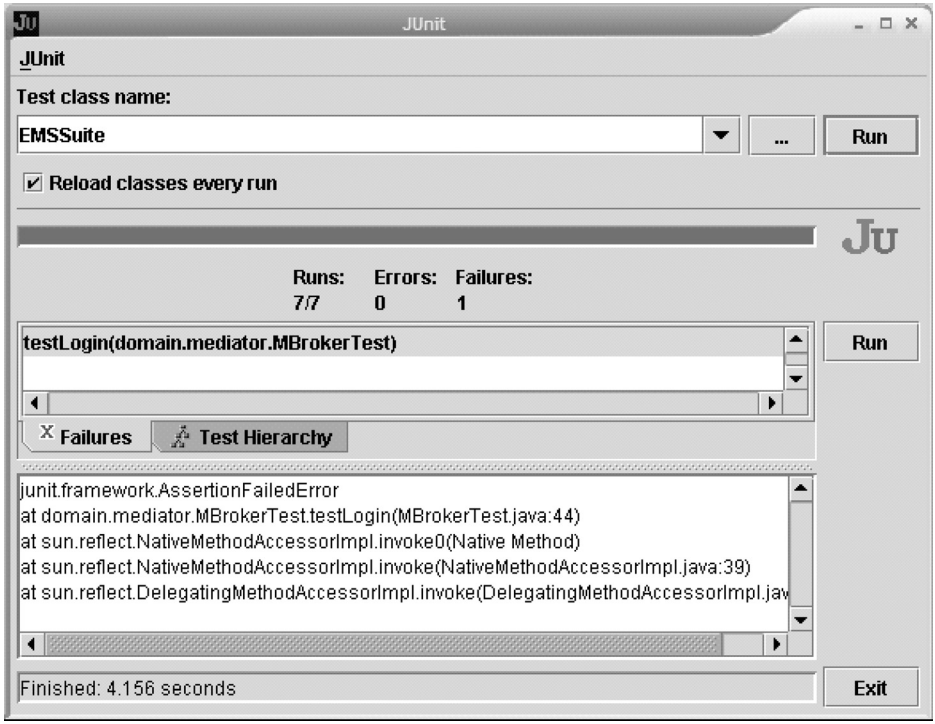


Рис. 12.9. Ошибка теста MBrokerTest()

`MBrokerTest`, позволяют выполнить достаточно полное тестирование взаимодействия «Регистрационное имя».

Листинг 12.17. Метод testConnecting() в FConnectionTest

Метод testConnecting() в FConnectionTest

```
43:     public void testConnecting(){
44:       System.out.println("Foundation: Connection Testing");
46:        try{
48:           FConnection conn = new
       FConnection(IAConstants.DB_DRIVER,IAConstants.DB_URL);
49:           conn.connect("pse1","pse1");
51:           assertTrue(conn.isConnected());
52:        }catch(Exception exc){
53:           exc.printStackTrace();
54:           fail("Fail to connect: "+exc.getMessage());
55:        }
56:     }
```

Листинг 12.18. Метод testRetrieveEmployee() в FReaderTest

```
Метод testRetrieveEmployee() в FReaderTest

56:     public void testRetrieveEmployee(){
57:        System.out.println("Foundation: Retrieval Employee
                                                    Testing");
58:        try{
59:           FConnection conn =
                 new FConnection(IAConstants.DB_DRIVER,
                                          IAConstants.DB_URL);
60:           FReader reader = conn.getReader();
61:           conn.connect("pse1","pse1");   //связь с БД
62:           java.sql.ResultSet rs = reader.query
                 ("Select * from employee where login_name
                                                  = 'pse1'");
63:           IAEmployee emp = mapEmployee(rs);
64:           assertEquals(emp.getLoginName(), "pse1");
65:        }catch(Exception exc){
66:           fail("Exception in requesting emp data:
                                           "+exc.getMessage());
67:        }
68:     }
```

12.3. Приемочные испытания и регрессионное тестирование

Управляемая тестированием разработка обеспечивает проектные решения, создает документацию ПО и облегчает развитие ПО. Цели управляемой тестированием разработки меньше, чем у отдельных единиц разработки. Испытательными компонентами обычно являются отдельные классы. Даже если испытательным компонентом является совокупность классов, характер его использования является испытанием «белого ящика». Управляемая тестированием разработка не гарантирует, что функциональные требования системы реализованы. Тестирование «белого ящика» не может, например, обнаружить «пропущенные функциональные возможности» — часть функциональных возможностей, которые не были реализованы по какой-либо причине.

Приемочные испытания заполняют этот пробел [66]. Заинтересованные в проекте стороны, клиенты и пользователи, проводят приемочные испытания. Это — **испытания «черного ящика»**, которые проверяют, выполнены ли требования сценария использования, не рассматривая внутреннюю работу ПО. Управляемая тестированием разработка определяет реализацию. Приемочные испытания проверяют и одобряют реализацию.

Рис. 12.10 обрисовывает место и роль тестирования в жизненном цикле разработки. Отправная точка к управляемой тестированием разработке — множество документов: документ сценария использования, концептуальная

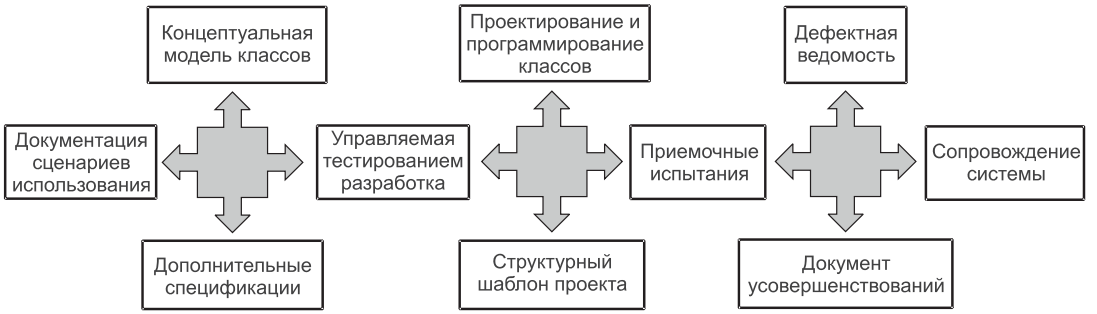


Рис. 12.10. Тестирование в жизненном цикле разработки

модель класса и дополнительные спецификации (глава 8). Место и роль структурного шаблона проекта (глава 9) в отношении управляемой тестированием разработки — несколько спорная проблема. Быстрая разработка не предполагает, что реализация структурного шаблона возложена на программистов. Однако в крупномасштабных разработках новые проекты являются расширениями предшествующих решений, и управление такими разработками требует, чтобы существовал ясный структурный проект и он был расширяемым. В этой книге структурное проектирование (PCMEF-шаблон) рассмотрено ранее управляемой тестированием разработки.

Достаточно интересен факт, что приемочные испытания являются тестированием «черного ящика». Это делает возможным писать содержание приемочных испытаний совершенно независимо от способа проектирования системы и приемов программирования. Приемочные испытания пишутся как **сценарии тестирования**. Ожидается, что большинство этих сценариев тестирования можно будет автоматизировать, используя *инструментальные средства захвата/воспроизведения*, программируя на *языке использования сценариев* (который поставляется с инструментальным средством захвата/воспроизведения), или даже программируя на языке, используемом для прикладной разработки (в некоторых случаях испытательный код — составная часть прикладного кода).

Сценарии тестирования, которые не могут быть автоматизированы, будут использоваться для ручного тестирования человеком-испытателем. Любые ошибки системы, обнаруженные приемочными испытаниями, классифицируются как **дефекты** (чтобы закрепить в текущей итерации) или **усовершенствования** (которые откладываются в сторону для рассмотрения в будущих итерациях).

Подобно контрольным примерам и тестовым наборам управляемой тестированием разработки, большинство приемочных испытаний должно быть в форме программ, которые выполняют код реализации и либо пропускают реализованную программу, либо бракуют ее. Фактически сформированные приемочные испытания часто реализуются в **контрольных примерах** и **тестовых наборах** способом, подобным показанному на рис. 12.8. Однако есть два главных различия.

Во-первых, программный компонент приемочных испытаний — часть функциональных возможностей, которые обычно охватывают многие классы. Во-вторых, приемочные испытания содержат **точки проверки**, где выясняется, выполнены ли ожидаемые функциональные возможности, которые должна обеспечить реализация. Точки проверки определяют ожидания заинтересованных сторон в отношении поведения системы при конкретных входных условиях, бизнес-правилах, ожидаемых результатах бизнес-операций, ожидаемых изменениях в БД, желаемых путях представления вычислений и обработки результатов и т. д.

Регрессионное тестирование — повторное выполнение соответствующих приемочных испытаний на последовательных итерациях кода. Цель регрессионного тестирования состоит в том, чтобы гарантировать, что итеративные расширения (шаги) кода не привели к непреднамеренным побочным эффектам и ошибкам в старых частях кода, которые, как предполагалось, не менялись в течение итерации.

Так как регрессионные тесты должны быть заново выполнены, по крайней мере, до формирования изделия в конце каждой итерации, они должны быть автоматизированы в максимально возможной степени. Ручное регрессионное тестирование людьми-испытателями для больших систем непрактично.

Элемент приемочных испытаний/регрессионного тестирования состоит из:

- сценария тестирования;
- реализации сценария тестирования (если возможно автоматическое выполнение сценария);
- спецификации входных данных, используемых для тестирования;
- документа результатов испытаний.

12.3.1. Сценарии тестирования в управлении электронной почтой

Сценарий тестирования может быть составлен в виде таблицы (таблица 12.1), показывающей **шаги тестирования** и *точки проверки*. Шаги и точки проверки последовательно пронумерованы. Буква `S` или `V` после номера указывает на шаг (step) или точку проверки (verification point). Кроме того, строки с точками проверки изображены курсивом.

Каждая **точка проверки** формулируется как вопрос. В зависимости от того, как вопрос сформулирован, или положительный (*`true`* — истина), или отрицательный (*`false`* — ложь) ответ на вопрос может означать успешную проверку. Столбец `Проверка` определен в предположении **автоматической проверки**. Возможные варианты точек проверки могут быть: *`false`*, *`true`* или *`unknown`* (неизвестно). Если автоматизированная проверка не может сама определить результат анализа, точка проверки даст результат *`unknown`*. Чтобы завершить проверку `unknown` и сделать вывод, прошло ли испытание этой точки проверки успешно или нет, может быть использована **ручная проверка** тестировщиком.

Таблица 12.1. Документ сценария тестирования

Строка	Проверка	Описание
1S		Вход в директиву «регистрационное имя» — неправильное пользовательское имя и пароль
2V	*false*	*Правильное регистрационное имя?*
3S		Вход в директиву «регистрационное имя» — правильное пользовательское имя и пароль
4V	*true*	*Правильное регистрационное имя?*
5S		Вход в директиву «получить служащего» — корректировка пользовательского имени
6V	*true*	*Существует служащий с регистрационным именем?*
7S		Вход в директиву «число исходящих сообщений»
8V	*true*	*Это число сообщений можно отобразить?*
9S		Вход в директиву «список исходящих сообщений». Необязательно формирование максимального числа отображаемых сообщений. Отображаются только идентификаторы и темы исходящих сообщений
10V	*true*	*Отображается информация о сообщениях?*
11S		Вход в директиву «получение исходящего сообщения». Получение идентификатора (id) исходящего сообщения. Отображение идентификатора, темы и текста исходящего сообщения согласно полученному идентификатору
12V	*true*	*Отображается информация о сообщении?*
13S		Вход в директиву «передача сообщения по электронной почте». Сообщение по электронной почте к самому себе отображается в шаге 12
14V	*unknown*	*Получено электронное сообщение по вашему адресу?*
15S		Вход в директиву «получение исходящего сообщения» для идентификатора (id), только что посланного по электронной почте. Отображаются идентификатор, тема и текст сообщения
16V	*false*	*Отображается информация о сообщении?*
17S		Вход в директиву «переустановить систему тестирования». Переустановка БД в состояние, которое было до выполнения сценария тестирования
18S		Вход в директиву «получение исходящего сообщения» для идентификатора (id), только что посланного по электронной почте. Отображаются идентификатор, тема и текст сообщения
19V	*true*	*Отображается информация о сообщении?*
20S		Вход в директиву «закрыть EM-приложение»
21V	*true*	*Завершена работа с регистрационным именем?*
22S		Завершение или новый запуск этого сценария тестирования

Сценарии тестирования нацелены на тестирование отобранных функциональных возможностей, определенных заинтересованными сторонами проекта — пользователями, менеджерами, спонсорами, — но также и разработчиками. Сценарий тестирования в таблице 12.1

- тестирует неправильно введенное (`2V`) и правильно введенное (`4V`, `6V`) регистрационное имя;
- получает и отображает исходящие сообщения для подключенного служащего (`8V`, `10V`, `12V`);

- передает по электронной почте указанное исходящее сообщение (`14V`);
- пытается получить и отобразить только что переданное по электронной почте исходящее сообщение (`16V`);
- заново устанавливает систему отладки, то есть повторно устанавливает БД в первоначальное состояние и отображает ранее переданное по электронной почте исходящее сообщение (`19V`);
- тестирует успешное завершение работы с регистрационным именем (`21V`) и заканчивает тестирование.

12.3.2. Испытательные входные и выходные данные и регрессионное тестирование в управлении электронной почтой

Итерация 1 подсистемы EM является консольным приложением. По этой причине нельзя воспользоваться преимуществами средств захвата/воспроизведения, потому что они требуют GUI-приложения. Приемочные испытания для итерации 1 реализованы на Java. Один такой испытательный класс называется `CEMSAcceptanceTest` (класс «приемочные испытания для подсистемы EM» пакета `control`) и помещен в EM-пакет `control` (управление) итерации 1. Класс реализует сценарий тестирования, приведенный в таблице 12.1.

Полный код для `CEMSAcceptanceTest` можно взять с Web-сайта данной книги. Для упрощения ссылок номера строк у фрагментов кода, рассматриваемых в книге, оставлены теми же, что и в полном коде на Web-сайте.

Испытательные входные данные для `CEMSAcceptanceTest` можно задать или в интерактивном режиме, или с помощью отдельного файла. Интерактивный способ допускает выполнение шагов тестирования в любой последовательности, но для частого тестирования и для регрессионного тестирования, в частности, следует использовать входной файл.

Листинг 12.19 является примером входного файла для тестирования. Входной файл содержит директивы для всех шагов и точек проверки сценария тестирования, приведенного в таблице 12.1. Директивы, которые относятся к `CEMSAcceptanceTest`, реализованы в виде постоянных полей, как показано в листинге 12.20. Значения директив составляют инвариантную часть системы отладки. *Система отладки* определяет инвариантный набор входных данных (любые входные файлы и ожидаемое содержание БД), который не должен изменяться от испытания к испытанию.

Например, сообщение о равенстве идентификатора (id) величине `14` для директивы `11s` (листинг 12.19) подразумевает, что исходящее сообщение в БД с этим id должно оставаться неизменным при различных выполнениях тестовой программы. На практике эта конкретная запись должна быть помечена в БД как *испытательные данные*, которые никогда не следует изменять в процессе нормального (производственного) использования БД.

Листинг 12.19. Входной файл для тестирования

Входной файл для тестирования

```
1s:  login
        pse1
        pse10
2v:  false
3s:  login
        pse1
        pse1
4v:  true
5s:  getemployee
6v:  true
7s:  countmessages
8v:  true
9s:  listmessages
        20
10v: true
11s: getmessage
        14
12v: true
13s: sendmessage
        14
        bliong@ics.mq.edu.au
        leszek@ics.mq.edu.au
        Product missing
        The Product name for your ad CI223375XY is missing.
        Can you please supply it?
14v: unknown
15s: getmessage
        14
16v: false
17s: resetTestbed
18s: getmessage
        14
19v: true
20s: closeEM
21v: true
22s: quit
```

Когда исходящее сообщение `14` будет извлечено из БД, его содержание отображается и проверяется (`12v`). Это содержание затем передается шагу `13s`, который полагает, что сообщение id, как и инструкции электронной почты «from» (от кого), «to» (кому), «subject» (тема) и «body» (содержимое), передаются в списке аргументов директивы `SendMessage` (послать сообщение). Ожидаемая проверка `SendMessage` (`14v`) не дает определенного результата (unknown), потому что у тестовой программы или EM-программы

нет никакой возможности проверить, что исходящее сообщение действительно достигло получателя. Это должно быть отдельно проверено получателем.

Листинг 12.20. Директивы для тестирования

Директивы для тестирования

```
46:     public static final String DIRECTIVES[] =
47:        new String[] {
48:           "Login",
49:           "GetEmployee",
50:           "GetMessage",
51:           "ListMessages",
52:           "SendMessage",
53:           "CountMessages",
54:           "ResetTestbed",
55:           "CloseEM",
56:           "Quit" };
```

Листинг 12.21 представляет собой выходной файл при «успешном» тестировании, то есть выполнение теста, где все точки проверки во входном файле были успешно просмотрены. **Регрессионное тестирование** может быть выполнено, используя входной файл, которому соответствует выходной файл, сформированный ранее тестовой программой. Если программа обнаружит, что такая пара вход/выход уже существует, она выполняет регрессионное тестирование с использованием входного файла и проверяет, являются ли результаты испытаний в предыдущем и текущем выходных файлах одними и теми же. Проверка выполняется просто обнаружением различия между первым и вторым выходными файлами. Если второй выходной файл уже существует при запуске программы, она заменит его содержание новым содержанием.

Листинг 12.21. Выходной файл при успешном тестировании

Выходной файл при успешном тестировании

```
2v:  OK
4v:  OK
6v:  OK
8v:  OK
10v: OK
12v: OK
14v: UNKNOWN
16v: OK
19v: OK
21v: OK
```

Листинг 12.22. Метод main() в CEMSAcceptanceTest

Метод main() в CEMSAcceptanceTest

```
293:    public static void main(String args[]) throws Exception
        {
294:       String inputFilename = null;
296:       CEMSAcceptanceTest at = new CEMSAcceptanceTest();
297:
298:       if (args.length == 0) {
299:          BufferedReader in =
300:        new BufferedReader(new InputStreamReader(System.in));
301:          String s =
302:             "Do you want to go interactive (I) or
                 provide input filename (F):";
303:          System.out.print(s);
304:          s = in.readLine().trim().toLowerCase();
305:          if (s.equals("i"))
306:            at.performTesting(goInteractiveMode(),
                                       inputFilename);
307:          else {
308:             System.out.print("Filename: ");
309:             inputFilename = in.readLine().trim();
310:             at.performTesting(
311:                goFileMode(new File(inputFilename)),
312:                inputFilename);
313:          }
314:       } else {
315:          inputFilename = args[0].trim();
316:          at.performTesting(goFileMode(new
                        File(inputFilename)), inputFilename);
317:       }
319:    }
```

12.3.3. Реализация сценария тестирования в управлении электронной почтой

Метод `main()` (основной) класса `CEMSAcceptanceTest` (класс «приемочные испытания для подсистемы EM» пакета `control`) инициализирует объект класса `CEMSAcceptanceTest` (листинг 12.22, строка 296), и он затем ожидает или интерактивного входа от пользователя, или испытательного входного файла (строка 302). Если выходной файл от предыдущего испытательного запуска существует, он заменяется новым (строка 316).

Единственное, что конструктор `CEMSAcceptanceTest()` делает (листинг 12.23), это создание нового объекта класса `CActioner` — класс «исполнитель» пакета `control` (строка 61). Пакет `control` (управление) итерации 1 — единственный уровень, с которым `CEMSAcceptanceTest` связывается при организации тестирования.

Получив директиву и список аргументов, метод `test()` (листинг 12.22) выполняет тестирование в соответствии с директивой. Каждая директива знает, каковы ее аргументы. Не соответствующие аргументы вызывают исключение `IllegalArgumentException` (исключение, связанное с неправильными аргументами), которое приводит к ошибке в тестировании.

Листинг 12.23. Конструктор CEMSAcceptanceTest()

Конструктор CEMSAcceptanceTest()

```
60:     public CEMSAcceptanceTest() {
61:        actioner = new CActioner();
62:        }
```

Листинг 12.24. Метод test() в CEMSAcceptanceTest

Метод test() в CEMSAcceptanceTest

```
253:   public void test(String directive, List args)
                                        throws Exception {
254:      if ("Login".equals(directive))
255:         doLogin(args);
256:      else if ("Getemployee".equals(directive))
257:         doGetEmployee();
258:      else if ("Listmessages".equals(directive))
259:         doListMessages(args);
260:      else if ("Getmessage".equals(directive))
261:         doGetMessage(args);
262:      else if ("Sendmessage".equals(directive))
263:         doSendMessage(args);
264:      else if ("Countmessages".equals(directive))
265:         doCountMessages();
266:      else if ("Resettestbed".equals(directive))
267:         doResetTestbed();
268:      else if ("Closeem".equals(directive))
269:         doCloseEM();
270:      else
271:         throw new IllegalArgumentException
                                ("Wrong directive is given");
272:   }
```

Как пример, директива `GetMessage` — получить сообщение (`11s`) обращается к методу `test()`, чтобы вызвать метод `doGetMessage()` — выполнить получение сообщения (строка 261 в листинге 12.24). Метод `doGetMessage()` предполагает, что сообщение id передается в списке аргументов (листинг 12.25). Затем он запрашивает `CActioner`, чтобы тот извлек исходящее сообщение (строка 152), и конструирует содержание исходящего сообщения, подготавливая это сообщение для передачи по электронной почте с помощью директивы `13s`.

Листинг 12.25. Метод doGetMessage() в CEMSAcceptanceTest

Метод doGetMessage() в CEMSAcceptanceTest

```
149:    private void doGetMessage(List args) {
150:       try {
151:          lastAction =
152:             actioner.retrieveMessage(Integer.parseInt
                                  (args.get(0).toString()));
153:          if (lastAction == null)
154:             throw new IllegalArgumentException
                                 ("No such message retrieved");
156:          IAOutMessage msg = (IAOutMessage) lastAction;
157:          StringBuffer buf = new StringBuffer();
158:          buf.append("Message id:").
                  append(msg.getMessageID()).append("\n");
159:          buf.append("Message subject: ").
                  append(msg.getSubject());
160:          buf.append("Message text:").
                  append(msg.getMessageText());
161:          buf.append("\n\n");
163:          recordData(buf.toString());
164:       } catch (Exception exc) {
165:          throw new IllegalArgumentException
                              ("Wrong messageid is given.");
166:       }
167:    }
```

Код выполнения теста показан в листинге 12.26. Он запускает тест — выход теста соответствует ожидаемому выходу, представленному в листинге 12.21. Результат точки проверки `14v` должен все еще анализироваться вручную.

Листинг 12.26. Листинг выполнения

Листинг выполнения

```
1s: login
pse1
pse10
2v: false
--> 2v: OK
3s: login
pse1
pse1
4v: true
--> 4v: OK
```

```
5s: getemployee
6v: true
--> 6v: OK
7s: countmessages
8v: true
=====================================
Total messages: 2
=====================================
--> 8v: OK
9s: listmessages
20
10v: true
=====================================
List of messages
Message id: 14
Message subject: Product missing

Message id: 45
Message subject: Are you satisfied?
=====================================
--> 10v: OK
11s: getmessage
14
12v: true
=====================================
Message id: 14
Message subject: Product missing
Message text:The Product name for your ad CI223375XY is
missing. Can you please supply it?
=====================================
--> 12v: OK
13s: sendmessage
14
bliong@ics.mq.edu.au
leszek@ics.mq.edu.au
Product missing
The Product name for your ad CI223375XY is missing. Can you
please supply it?
14v: unknown
--> 14v: UNKNOWN
15s: getmessage
14
16v: false
--> 16v: OK
17s: resetTestbed
18s: getmessage
14
19v: true
```

```
====================================
Message id: 14
Message subject: Product missing
Message text:The Product name for your ad CI223375XY is
missing. Can you please supply it?
====================================
--> 19v: OK
20s: closeEM
21v: true
--> 21v: OK
22s: quit
```

12.4. Итерация 1. Образы экрана времени выполнения

Итерация 1 является неграфическим консольным приложением. Программа может быть запущена в «окне DOS», как показано на рис. 12.11. Выполнение в DOS визуально не отделяет выход программы от входа пользователя. Выполнение приложения в интегрированной среде разработки (IDE) может обеспечить такое разделение. Рис. 12.12 показывает выполнение в IDE-окне.

Листинг 12.27 представляет «распечатку экрана» выполнения итерации 1. Распечатка была получена копированием/размещением одного сеанса запуска на экране DOS. Сеанс демонстрирует «удачный путь» из документа сценария использования, то есть, основного потока и подпотоков (разделы 8.2.2 и 8.2.3).

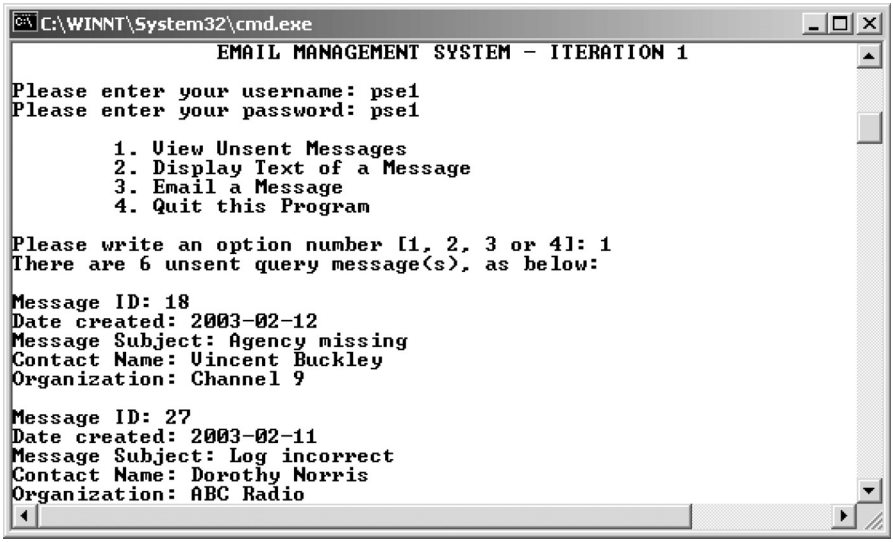


Рис. 12.11. Выполнение итерации 1 в экране DOS
Источник: образ экрана Microsoft® DOS, перепечатано с разрешения Microsoft Corporation, Copyright © 1990–2003 Microsoft Corporation

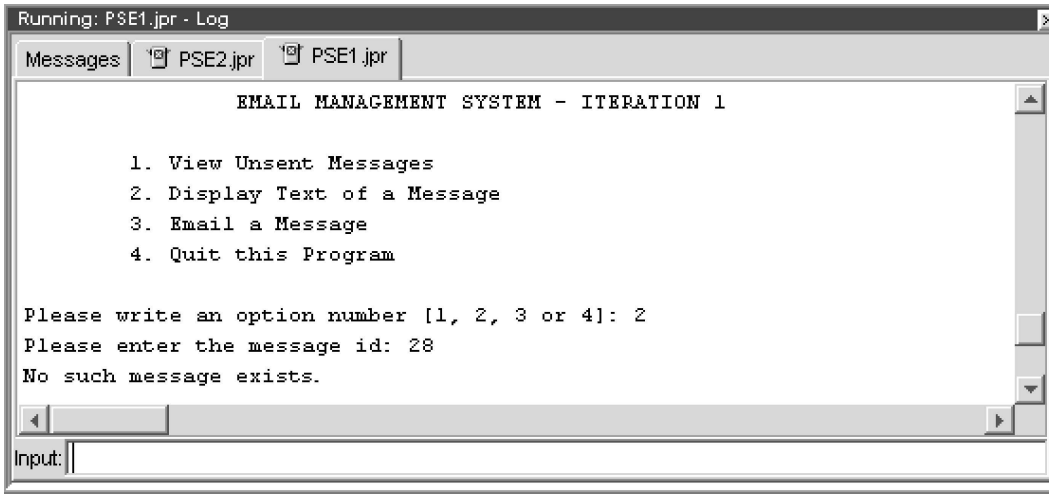


Рис. 12.12. Выполнение итерации 1 в IDE-экране

Листинг 12.27. Распечатка экрана выполнения итерации 1 — «удачный путь»

Распечатка экрана выполнения итерации 1 — «удачный путь»

```
                EMAIL MANAGEMENT SYSTEM - ITERATION 1

Please enter your username: pse1
Please enter your password: pse1

        1. View Unsent Messages
        2. Display Text of a Message
        3. Email a Message
        4. Quit this Program

Please write an option number [1, 2, 3 or 4]: 1
There are 2 unsent query message(s), as below:

Message ID: 14
Date created: 2003-02-12
Message Subject: Product missing
Contact Name: Pablo Romero
Organization: SBS Sydney

Message ID: 45
Date created: 2003-02-11
Message Subject: Are you satisfied?
Contact Name: Agatha Sommer
Organization: Ford Australia
```

```
        1. View Unsent Messages
        2. Display Text of a Message
        3. Email a Message
        4. Quit this Program

Please write an option number [1, 2, 3 or 4]: 2
Please enter the message id: 14
Message ID: 14
Date created: 2003-02-12
Message Subject: Product missing
Contact Name: Pablo Romero
Organization: SBS Sydney
Message Text:
The Product name for your ad CI223375XY is missing. Can you
please supply it?

        1. View Unsent Messages
        2. Display Text of a Message
        3. Email a Message
        4. Quit this Program

Please write an option number [1, 2, 3 or 4]: 3
Please enter the message id you want to email: 14
The message is successfully sent and updated.

        1. View Unsent Messages
        2. Display Text of a Message
        3. Email a Message
        4. Quit this Program

Please write an option number [1, 2, 3 or 4]: 1
There are 1 unsent query message(s), as below:

Message ID: 45
Date created: 2003-02-11
Message Subject: Are you satisfied?
Contact Name: Agatha Sommer
Organization: Ford Australia

        1. View Unsent Messages
        2. Display Text of a Message
        3. Email a Message
        4. Quit this Program

Please write an  option number [1, 2, 3 or 4]:
```

Листинг 12.28 представляет «распечатку экрана» выполнения итерации 1 для «неудачного пути» из документа сценария использования, то есть, потока исключения (раздел 8.2.4).

Листинг 12.28. Распечатка экрана выполнения итерации 1 — «неудачный путь»

Распечатка экрана выполнения итерации 1 — «неудачный путь»

```
                EMAIL MANAGEMENT SYSTEM - ITERATION 1

Please enter your username: pse1
Please enter your password: wrong_password
Invalid login. Please try again. This will be your second try.
Please enter your username: pse1
Please enter your password: pse1

        1. View Unsent Messages
        2. Display Text of a Message
        3. Email a Message
        4. Quit this Program

Please write an option number [1, 2, 3 or 4]: 5
Invalid option. Please choose 1,2,3, or 4.

        1. View Unsent Messages
        2. Display Text of a Message
        3. Email a Message
        4. Quit this Program

Please write an option number [1, 2, 3 or 4]:
Please write an option number [1, 2, 3 or 4]: 1
There are 20 unsent query message(s), as below:

Message ID: 50
Date created: 2003-02-12
Message Subject: Your competitors
Contact Name: Pablo Romero
Organization: SBS Sydney

Message ID: 27
...

There are 22 unsent query messages left unretrieved in
database.

        1. View Unsent Messages
```

```
        2. Display Text of a Message
        3. Email a Message
        4. Quit this Program

Please write an option number [1, 2, 3 or 4]: 3
Please enter the message id you want to email: 50
Failed in sending message. Verify the mail properties or
message's addresses.
Failure info: no network connection or mail address invalid.

                EMAIL MANAGEMENT SYSTEM - ITERATION 1

        1. View Unsent Messages
        2. Display Text of a Message
        3. Email a Message
        4. Quit this Program

Please write an option number [1, 2, 3 or 4]:
```

Итерация 1 выполняет некоторое основное форматирование сообщения, передаваемого по электронной почте, перед тем, как оно будет послано по своему назначению. Рис. 12.13 показывает пример одного такого сообщения. Перед посылкой исходящему сообщению добавляют заголовок с «приветствием», строку с благодарностями и имя отправителя.

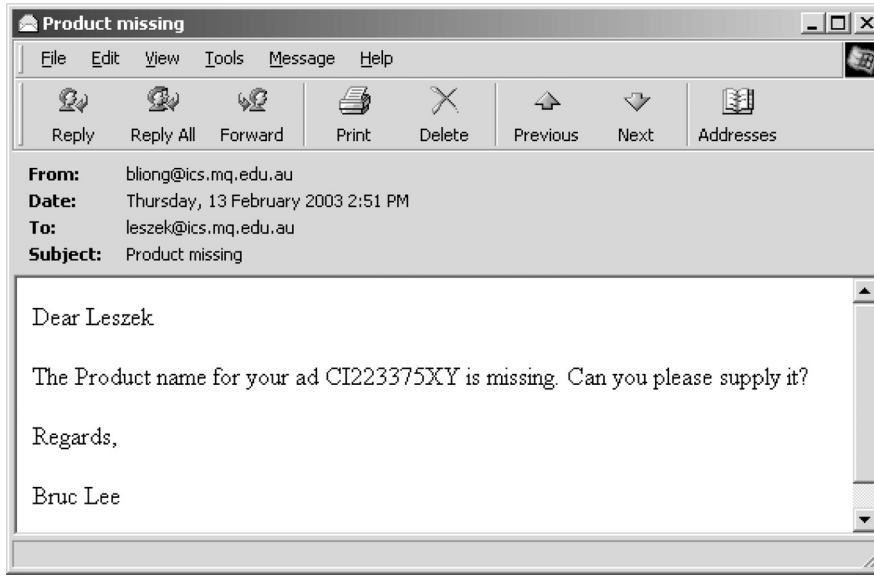


Рис. 12.13. Итерация 1 сообщения по электронной почте после дополнения его адресата

Резюме

1. Определение *класса Java* состоит из имени класса и его видимости, элементов данных и методов, конструкторов, видимости элементов и различных суперклассов и интерфейсов класса.
2. *Интерфейс класса* (его протокол) — это не то же самое, что и интерфейс Java или UML-интерфейс.
3. Имеются две категории *элементов данных*: переменные экземпляра и переменные класса. Имеются также две категории *методов*: методы экземпляра и методы класса.
4. *Перегружаемые методы* имеют одно и то же имя, но различную сигнатуру.
5. *Коллекция Java* представляет контейнер для объектов. Она может хранить многие ссылки на объекты и, следовательно, использоваться, чтобы хранить связи ассоциации с множественностью «многие».
6. В C++ используется *параметризованный тип*, чтобы реализовать связи ассоциации с множественностью «многие».
7. JDBC и SQLJ — два стандартных способа включить SQL-операторы в код Java, чтобы связаться с БД.
8. Установление связи с БД включает использование JDBC-класса `DriverManager` (управление драйвером) и JDBC-интерфейса `Connection` (связь) из библиотеки `java.sql`.
9. Должен быть создан объект оператора, чтобы выполнить SQL-запрос в БД. Объект оператора должен соответствовать одному из трех JDBC-интерфейсов: `Statement` (оператор), `PreparedStatement` (подготовленный оператор) или `CallableStatement` (вызываемый оператор).
10. Метод `preparedCall()` (подготовленный вызов) возвращает объект `CallableStatement` (вызываемый оператор), который может использоваться, чтобы вызвать хранимую процедуру или функцию.
11. *Управляемая тестированием разработка* требует написания тестовых программ до написания кода приложения, чтобы проверять его с помощью этих программ. Такая разработка является примером *тестирования «белого ящика»*. Управляемая тестированием разработка поддерживается JUnit-шаблоном.
12. *Приемочные испытания* используются, чтобы проверить код приложения после того, как он будет написан. Они являются примером *тестирования «черного ящика»*. Приемочные испытания пишутся как *сценарии тестирования* с *точками проверки*. Недостатки, обнаруженные приемочными испытаниями, подразделяются на *дефекты* и *усовершенствования*.
13. *Регрессионное тестирование* — выполнение заново приемочных испытаний при последовательных итерациях создания кода. Регрессионное тестирование требует стабильных систем отладки. *Система отладки* определяет инвариантное множество входных данных (любые входные файлы и ожидаемое содержание БД), которое не должно изменяться от испытания к испытанию.

14. Управляемая тестированием разработка, последовательное программирование приложения и приемочные испытания завершаются созданием исполняемой программы. Итерация 1 программы является *консольным приложением*, которое может быть запущено в окне DOS или в IDE-окне.

Ключевые термины

BC4J . . . *См.* Business Components for Java
Business Components for Java 459
EJB . . . . . *См.* Enterprise JavaBeans
Enterprise JavaBeans 458
eXtreme Programming 467
JavaBeans 458
Java Database Connectivity. 458
Java Development Kit 458
Java Stored Procedures 458
JDBC . . . . *См.* Java Database Connectivity
JDK. . . . . См. Java Development Kit
JUnit 469
SQL for Java 458
SQLJ . . . . . *См.* SQL for Java
XP . . . . . *См.* eXtreme Programming
автоматическая проверка 480
анонимный внутренний класс 471
базовый класс *См.* параметризованный тип
быстрая разработка ПО 467
видимость 446
внешний класс 468
внутренний класс 468
возвращаемый тип. 447
дефект 479
закрепление 470
интерфейс класса 447
испытательный компонент 468
класс «`anonymous_type`». 455
коллекция 452
конструктор 448
контейнер *См.* коллекция
контрольный пример 468, 479
локальная переменная 447
метод. 447
метод класса 447
метод экземпляра 447
отражение 472
параметризованный тип 455
паттерн Компоновщик. 470
перегрузка . . . . . *См.* перегрузка метода
перегрузка метода 447
переменная экземпляра 447
поле . . . . . *См.* элемент данных
приемочные испытания. 445, 478
программирование намерений. 474
простой тип данных 447
протокол класса . . *См.* интерфейс класса
прототип метода 447
регрессионное тестирование 480, 484
ручная проверка 480
связь ассоциации 452
сигнатура. . . . . *См.* сигнатура метода
сигнатура метода 447
ссылочный тип данных. . . *См.* тип-класс
сценарий тестирования 479
тестирование «белого ящика» 468
тестирование «черного ящика» 478
тестовый набор 468, 479
точка проверки 480
тип аргумента 447
тип-класс. 447
управляемая тестированием разработка. . .
. . . . . 445, 467
управляемое тестированием программирование 445
усовершенствование. 479
шаблон класса *См.* параметризованный тип
шаг тестирования 480
элемент данных 447

Обзорные вопросы

1. Объясните различие между интерфейсом класса (протоколом класса), интерфейсом Java и UML-интерфейсом.
2. Приватные элементы данных (атрибуты) доступны только экземплярам класса, то есть любому объекту класса, а не только объекту, который со-

держит эти элементы данных. Приведите пример в Java с двумя объектами одного класса, такой, когда один из них имеет прямой доступ к значению атрибута другого объекта. Можно ли сделать какие-либо интересные выводы из этого вопроса?

3. Приведите Java-пример, иллюстрирующий, что локальная переменная внутри метода класса не является элементом данных.
4. Приведите Java-пример, иллюстрирующий, что к методу класса можно обращаться или через объект класса, или без объекта, ссылаясь на имя класса.
5. Приведите Java-пример, иллюстрирующий перегрузку метода.
6. В Java ассоциации с множественностью «многие» реализуются с помощью Java-коллекции. Имеются три главных вида коллекций: `List` (список), `Set` (множество) и `Map` (карта). Обсудите все «за» и «против» этих коллекций для реализации ассоциаций.
7. Сравните коллекции и параметризованные типы в качестве двух технологий для реализации ассоциаций.
8. Сравните JDBC и SQLJ для доступа к БД из прикладной программы.
9. Перечислите классы/интерфейсы библиотеки `java.sql`, которые являются наиболее полезными для программирования доступа в Java к БД.
10. Управляемая тестированием разработка завершается двумя категориями кода — испытательный код и прикладной код. Книга объясняет три подхода того, где испытательный код может быть размещен по отношению к прикладному коду. Каковы эти подходы?
11. JUnit поддерживает статические и динамические способы запуска контрольных примеров. Какие механизмы языка Java используются для реализации каждого способа?
12. Проанализируйте код в листинге 12.29, приведенном ниже. Каков выход программы? Просмотрите ее и объясните, что происходит.
13. Обратимся к разделу 12.2.2 и листингу 12.18. Как «получение информации о служащем» проверяется в листинге? Кажется, что нет никакой явной части листинга, которая проверяет, является ли поиск успешным. Объясните, как это тестирование выполнено.
14. Обратимся к разделу 12.3.1 и листингу 12.19. Сделайте комментарии относительно предположений, принятых для директивы `sendmessage` — послать сообщение (строка `13s`).

Примеры задач

Обучение и упражнения учебного примера

1. Обратимся к разделу 12.1.3 и листингу 12.5. Предположим, что вы разработали такой класс `Connection` (связь) и хотите написать `TestCase` (контрольный пример) как модуль для тестирования этого класса. Как бы вы могли протестировать этот класс?

2. Обратимся снова к разделу 12.1.3 и листингу 12.5. Продолжая предыдущий вопрос, предположим, что класс `Connection` не объявляет каких-либо формируемых исключений. Можно ли разработать `TestCase` для проверки его конструктора? Если можно, покажите, как.
3. Обратимся к разделу 12.2.2 и листингу 12.13. Как бы вы разработали заглушку для листинга? То есть, предположим, что вы не разработали метод `login()` (регистрационное имя) класса `CActioner` (класс «исполнитель» пакета `control`), но вы хотели бы иметь `TestCase`, соответствующий тому, что показано в листинге 12.13. Как бы вы сделали это?

Листинг 12.29. Программа-пример

Программа-пример

```
7:   public class TestMe{
8:      private String field1;
9:
10:     private TestMe anotherMe;
11:
12:     public static void main(String args[]){
13:        TestMe me = new TestMe();
14:        me.field1 = "Hello1";
15:        me.anotherMe = me;
16:        me.anotherMe.field1 = "Hello2";
17:        System.out.println(me.field1);
18:        Systern.out.println(me.anotherMe.field1);
19:        me.convertMe(me);
20:        System.out.println(me.field1);
21:        System.out.println(me.anotherMe.field1);
22:     }
23:     private void convertMe(TestMe anotherMe){
24:        anotherMe.field1 = "Hello3";
25:        anotherMe.anotherMe = new TestMe();
26:        anotherMe.anotherMe.field1 = "Hello4";
27:        anotherMe = anotherMe;
28:        anotherMe = new TestMe();
29:        anotherMe.field1 = "Hello5";
30:        anotherMe.anotherMe = new TestMe();
31:        anotherMe.anotherMe.field1 = "Hello6";
32:     }
33:  }
```

4. Обратимся к разделу 12.1.3 и листингу 12.8. Этот листинг изменен, как показано в листинге 12.30 ниже. Модификации показаны подчеркиванием текста. Создайте `TestCase` для приложения `MovieActor`, чтобы проверить код клиента в листинге 12.8. Что следует сделать, чтобы гарантировать успешное выполнение метода `testDisplaySearchMoviesByStoredFunction()`

(проверка отображения найденных фильмов с помощью хранимой функции)? Покажите соответствующий код. (Намек: создайте заглушку.)

Небольшой проект — система использования временного протокола

Этот небольшой проект предполагает, что вы имеете решения предыдущих вопросов проекта TLS (Time Logging System — система использования временного протокола), в особенности тех, которые находятся в главе 11. Эти решения включены в руководство для преподавателя, которое доступно преподавателям на Web-сайте книги.

1. Напишите класс испытательной единицы для поддержки управляемой тестированием разработки взаимодействия «Удаление временной записи».
2. Разработайте приемочный контрольный пример для взаимодействия «Удаление временной записи».

Листинг 12.30. Пример программы тестирования

```
testSearchMoviesByStoredFunction()

8:   public class MovieSearcherTest {
59:
60:     public void testDisplaySearchMoviesByStoredFunction(
               String str)
               throws Exception{
61:        System.out.println(
              "Searching movies on specified string:" +str);
62:        Collection c =
                        conn.searchMoviesByStoredFunction(str);
63:        Iterator it = c.iterator();
64:        while(it.hasNext()){
65:           displayMovie((Movie) it.next());
66:           System.out.println();
67:        }
68:     }
194:    private void displayMovie(Movie movie) {
195:       System.out.println("Movie Code: " +
                              movie.getMovieCode());
196:       System.out.println("Movie Title: " +
                              movie.getMovieTitle());
197:       System.out.println("Movie Director: " +
                              movie.getDirector ());
198:    }
205:    public static void main(String args[]) {
244:      searcher.displaySearchMoviesByStoredFunction("vamp");
249:    }
        ...
250: }
```

Небольшой проект — управление информацией о деловых партнерах

Этот небольшой проект предполагает, что вы имеете решения предыдущих вопросов проекта CIM, в особенности тех, которые находятся в главе 11. Эти решения включены в руководство для преподавателя, которое доступно преподавателям на Web-сайте книги.

1. Напишите приемочный контрольный пример на Java для взаимодействия «`Read OrganizationContact`» (чтение информации о партнере-организации).

Глава

13 Итерация 1. Аннотированный код

Учебный пример EM, рассматриваемый в этой книге, достаточно мал, что позволяет представить большую часть окончательного кода. Предыдущие главы использовали код EM, чтобы иллюстрировать обсуждаемые темы. Хотя была предпринята специальная попытка не представлять код вне его полного контекста, единственный способ полностью представить этот контекст — рассмотреть код целиком.

Соответственно, последние главы частей 2, 3 и 4 этой книги демонстрируют большую часть кода, полученного в последовательных итерациях учебного примера EM. Любой код, не представленный в этих главах по причине недостатка места, может быть взят с Web-сайта книги.

Эта глава содержит Java-код для итерации 1. Код БД сюда не включен, потому что он представлен в главе 10. Код представлен согласно типовой практике оформления Java-документации. Обзор кода объясняет иерархию пакетов, интерфейсов и классов. Диаграмма классов дает визуальный обзор кода. Большинство полей и методов перечислено и аннотируется. Все существенные методы обсуждены более подробно. Некоторые вспомогательные моменты, менее важные или тривиальные методы опущены в обсуждении этой главы.

13.1. Обзор кода

Java-код для итерации 1 строго соответствует структурному шаблону PCMEF, представленному в разделе 9.2.2. Более точно, он соответствует шаблону PCMEF+, который добавляет пакет `acquaintance` (знакомство) к другим пакетам PCMEF. Поэтому **иерархия пакетов** следующая:

- `acquaintance` (знакомство);
- `presentation` (представление);
- `control` (управление);
- `domain.entity` (пакет «сущность» уровня «предметная область»);
- `domain.mediator` (пакет «посредник» уровня «предметная область»);
- `foundation` (основание).

Пакет `acquaintance` содержит все интерфейсы, необходимые для связи между объектами в несоседних пакетах. Для простоты итерация 1 сокращает до минимума использование других видов интерфейса. Имеется только один

интерфейс вне пакета acquaintance. В **иерархии интерфейсов** имеются пять интерфейсов. Согласно PCMEF-соглашению имена интерфейсов начинаются с заглавной буквы «I».

- interface acquaintance.IAConstants (интерфейс «константы» пакета знакомств);
- interface acquaintance.IAContact (интерфейс «деловой партнер» пакета знакомств);
- interface acquaintance.IAEmployee (интерфейс «служащий» пакета знакомств);
- interface acquaintance.IAOutMessage (интерфейс «исходящее сообщение» пакета знакомств);
- interface domain.entity.IEDataSupplier (интерфейс «источник данных» пакета «сущность» уровня «предметная область»).

Чтобы подчеркнуть ясность и выдвинуть на первый план простоту PCMEF-шаблона, итерация 1 ограничивает число классов в каждом пакете. Это приводит к тому, что некоторые классы оказываются слишком большими и делающими слишком много. Такой подход нарушает единство классов в обмен на лучшее (меньшее) их соединение. В последующих итерациях будет уделено внимание однородности классов, и неоднородные классы будут разбиты на меньшие части. **Иерархия классов** в итерации 1 следующая:

- presentation.PMain (класс «основной» в пакете «представление»);
- presentation.PConsole (класс «консоль» в пакете «представление»);
- control.CActioner (класс «исполнитель» в пакете «управление»);
- domain.entity.EContact (класс «партнер» пакета «сущность» уровня «предметная область»);
  - implements acquaintance.IAContact (реализует интерфейс «контакт» пакета «знакомство»);
- domain.entity.EEmployee (класс «служащий» пакета «сущность» уровня «предметная область»);
  - implements acquaintance.IAEmployee (реализует интерфейс «служащий» пакета «знакомство»);
- domain.entity.EOutMessage (класс «исходящее сообщение» пакета «сущность» уровня «предметная область»);
  - implements acquaintance.IAOutMessage (реализует интерфейс «исходящее сообщение» пакета «знакомство»);
- domain.mediator.MBroker (класс «посредник» пакета «посредник» уровня «предметная область»);
  - implements acquaintance.IAConstants (реализует интерфейс «константы» пакета «знакомство»), domain.entity.IEDataSupplier (интерфейс «источник данных» пакета «сущность» уровня «предметная область»);
- foundation.FConnection (класс «соединение» пакета «основание»);
- foundation.FReader (класс «чтение» пакета «основание»);
- foundation.FWriter (класс «запись» пакета «основание»).

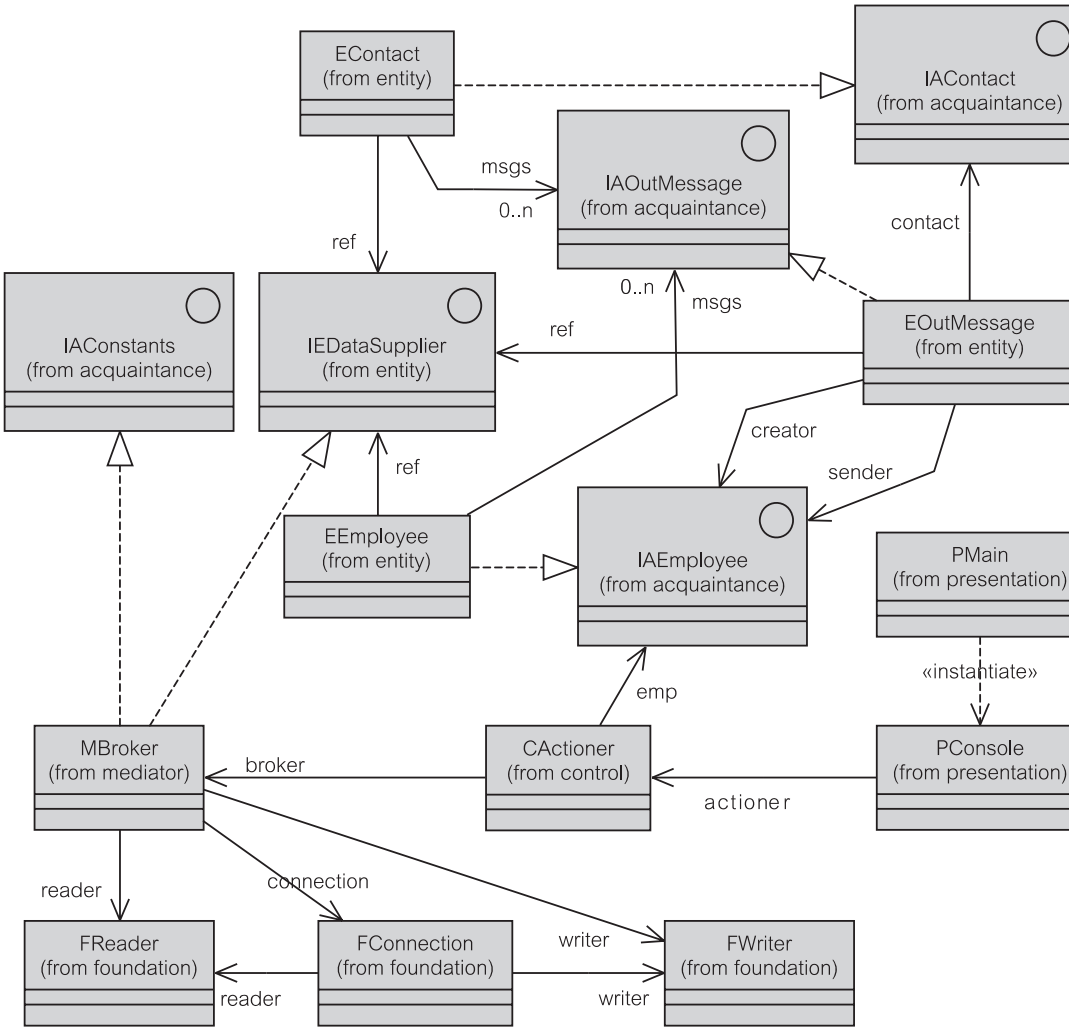


Рис. 13.1. Диаграмма классов для итерации 1 EM

Рис. 13.1 показывает окончательную модель классов для итерации 1. Модель слегка расширяет диаграмму классов, представленную на рис. 11.2. Интерфейс `IEDataSupplier` добавлен в окончательном решении. На рис. 13.1 также сделано явным существование класса `PMain`.

13.2. Пакет Acquaintance

Пакет `acquaintance` (знакомство) состоит из четырех интерфейсов:

- `IAConstants` (интерфейс «константы» пакета `acquaintance`);
- `IAContact` (интерфейс «деловой партнер» пакета `acquaintance`);

- `IAEmployee` (интерфейс «служащий» пакета `acquaintance`);
- `IAOutMessage` (интерфейс «исходящее сообщение» пакета `acquaintance`).

13.2.1. Интерфейс IAConstants

Как объяснено в разделе 11.1.2, интерфейс может использоваться, чтобы хранить константы системы. Группировка констант в единственном интерфейсе позволяет настраивать (параметризировать) поведение программы для внедрения и для различных вариантов использования. Альтернативой к использованию интерфейса могло бы быть создание файла ресурсов, чтобы хранить всю информацию инициализации. Интерфейс, который размещает константы системы, делает задачу инициализации более прозрачной. Пользователь должен только изменить значения констант и перекомпоновать систему до внедрения.

Листинг 13.1 объясняет значение постоянных полей в интерфейсе `IAConstants` (интерфейс «константы» пакета `acquaintance`). Листинг 13.2 показывает код.

Листинг 13.1. Сводка полей интерфейса IAConstants

```
static String  DB_DRIVER
Драйвер в формате, указанном поставщиком БД. Мы используем
Oracle.
static String  DB_URL
JDBC URL для БД.
static int     MAX_MESSAGE
Предельное число сообщений, которые могут быть отображены.
Обратите внимание, что это не максимальное число извлекаемых
сообщений.
static String  SMTP_ADDRESS
SMTP-адрес для передачи по электронной почте[1].
```

Листинг 13.2. Интерфейс IAConstants.java

```
IAConstants.java

11:  public interface IAConstants {
16:     String DB_DRIVER = "oracle.jdbc.driver.OracleDriver";
19:     String DB_URL = "jdbc:oracle:thin:
        @localhost:1521:oracle8i";
23:     String SMTP_ADDRESS = "mail.ics.mq.edu.au";
26:     int MAX_MESSAGE = 20;
27:  }
```

[1] SMTP — Simple Mail Transfer Protocol — основной протокол электронной почты в Интернете. (*Прим. перев.*)

Листинг 13.3. Сводка методов интерфейса IAEmployee

`String`	**`getEmail`**`()` Получение электронного адреса служащего.
`String`	**`getEmployeeID`**`()` Получение идентификатора служащего.
`String`	**`getFamilyName`**`()` Получение фамилии служащего.
`String`	**`getFirstName`**`()` Получение имени служащего.
`String`	**`getLoginName`**`()` Получение регистрационного имени служащего.
`Collection`	**`getUnsentOutMessages`**`(int numMsgRetrieve)` Получение всех сообщений (вплоть до numMsgRetrieve — число извлеченных сообщений), предназначенных этому служащему.
`void`	**`removeSentOutMessage`**`()` Удаление сообщения из списка этого служащего, вероятно, потому, что оно уже послано.
`void`	**`setEmail`**`(java.lang.String email)` Задание электронного адреса служащего.
`void`	**`setEmployeeID`**`(java.lang.String id)` Задание идентификатора служащего.
`void`	**`setFamilyName`**`(java.lang.String familyName)` Задание фамилии данного служащего.
`void`	**`setFirstName`**`(java.lang.String fname)` Задание имени данного служащего.
`void`	**`setLoginName`**`(java.lang.String login)` Задание регистрационного имени служащего.

Листинг 13.4. Интерфейс IAEmployee.java

IAEmployee.java

```
9:   public interface IAEmployee {
11:     public void setEmployeeID( String id );
14:     public String getEmployeeID();
17:     public void setFirstName( String fname );
20:     public String getFirstName();
23:     public void setFamilyName( String familyName );
26:     public String getFamilyName();
29:     public void setEmail( String email );
32:     public String getEmail();
35:     public void setLoginName( String login );
38:     public String getLoginName();
41:     public Java.util.Collection getUnsentOutMessages(int
                      numMsgRetrieve);
44:     public void removeSentOutMessage(IAOutMessage msg);
46:  }
```

13.2.2. Интерфейс IAEmployee

Листинги 13.3 и 13.4 определяют интерфейс `IAEmployee` (интерфейс «служащий» пакета `acquaintance`). Класс `EEmployee` (класс «служащий» пакета `entity`) реализует этот интерфейс. Объект `EEmployee` представляет текущего пользователя EM-приложения.

13.2.3. Интерфейс IAContact

Листинги 13.5 и 13.6 определяют интерфейс `IAContact` (интерфейс «деловой партнер» пакета `acquaintance`). Класс `EContact` (класс «деловой партнер» пакета `entity`) реализует этот интерфейс. Объект `EContact` представляет бизнес-сущность, которой может быть послано исходящее сообщение.

Листинг 13.5. Сводка методов интерфейса IAContact

`String`	**`getContactID`**`()` Получение идентификатора делового партнера.
`String`	**`getEmail`**`()` Получение электронного адреса делового партнера.
`String`	**`getFamilyName`**`()` Получение фамилии делового партнера.
`String`	**`getFirstName`**`()` Получение имени делового партнера.
`String`	**`getOrganization`**`()` Получение организации, которой принадлежит данный деловой партнер.
`Collection`	**`getUnsentOutMessages`**`(int numMsgToBeRetrieved)` Получение всех исходящих сообщений, предназначенных данному деловому партнеру.
`void`	**`removeSentOutMessage`**`(IAOutMessage msg)` Удаление исходящего сообщения, предназначенного данному деловому партнеру, вероятно, потому, что оно уже послано.
`void`	**`setContactID`**`(java.lang.String contactID)` Задание идентификатора делового партнера.
`void`	**`setEmail`**`(java.lang.String email)` Задание электронного адреса делового партнера.
`void`	**`setFamilyName`**`(java.lang.String familyName)` Задание фамилии делового партнера.
`void`	**`setFirstName`**`(java.lang.String fname)` Задание имени делового партнера.
`void`	**`setOrganization`** `(java.lang.String org)` Задание организации, которой принадлежит деловой партнер.

Листинг 13.6. Интерфейс IAContact.java

IAContact.java

```
8:   public interface IAContact {
10:     String getContactID();
13:     void setContactID( String contactID );
16:     String getFirstName();
19:     void setFirstName( String fname );
22:     String getFamilyName();
25:     void setFamilyName( String familyName );
28:     String getEmail();
31:     void setEmail( String email );
34:     String getOrganization();
37:     void setOrganization( String org );
40:     java.util.Collection getUnsentOutMessages(int
         numMsgToBeRetrieved);
43:     void removeSentOutMessage(IAOutMessage msg);
44:  }
```

13.2.4. Интерфейс IAOutMessage

Листинги 13.7 и 13.8 определяют интерфейс `IAOutMessage`. Этот интерфейс представляет исходящие сообщения от компании к различным деловым партнерам.

Листинг 13.7. Сводка методов IAOutMessage

`IAContact`	**`getContact`**`()` Получение делового партнера для этого исходящего сообщения.
`java.sql.Date`	**`getCreationDate`**`()` Получение даты создания исходящего сообщения.
`IAEmployee`	**`getCreatorEmployee`**`()` Извлечение служащего, который первоначально создал это исходящее сообщение.
`int`	**`getMessageID`**`()` Получение идентификатора исходящего сообщения.
`String`	**`getMessageText`**`()` Получение содержимого исходящего сообщения.
`IAEmployee`	**`getSenderEmployee`**`()` Извлечение служащего, который отвечает за передачу этого исходящего сообщения.
`java.sql.Date`	**`getSentDate`**`()` Получение даты передачи этого исходящего сообщения.

java.lang.String	**getSubject**()
	Получение темы исходящего сообщения.
void	**setContact**(IAContact contact)
	Задание делового партнера для данного исходящего сообщения.
void	**setCreationDate**(java.sql.Date date)
	Задание даты создания этого исходящего сообщения.
void	**setCreatorEmployee**(IAEmployee emp)
	Задание служащего, который создает это сообщение.
void	**setMessageID**(int id)
	Задание идентификатора исходящего сообщения.
void	**setMessageText**(java.lang.String text)
	Задание текста исходящего сообщения.
void	**setSenderEmployee**(IAEmployee emp)
	Задание служащего, который отвечает за передачу этого исходящего сообщения.
void	**setSentDate**(java.sql.Date date)
	Задание даты передачи этого сообщения.
void	**setSubject**(java.lang.String subject)
	Задание темы исходящего сообщения.

Листинг 13.8. IAOutMessage.java

IAOutMessage.java

```
11:  public interface IAOutMessage {
14:     int getMessageID();
17:     void setMessageID( int id );
20:     IAEmployee getSenderEmployee();
23:     void setSenderEmployee(IAEmployee emp);
26:     IAEmployee getCreatorEmployee();
29:     void setCreatorEmployee(IAEmployee emp);
32:     IAContact getContact();
35:     void setContact(IAContact contact);
37:     String getSubject();
40:     void setSubject( String subject );
43:     String getMessageText();
46:     void setMessageText( String text );
49:     java.sql.Date getCreationDate();
52:     void setCreationDate( java.sql.Date date )
55:     java.sql.Date getSentDate();
58:     void setSentDate( java.sql.Date date );
60:  }
```

13.3. Пакет Presentation

Пакет `presentation` (представление) содержит два класса:

- `PMain` (класс «основной» пакета `presentation`);
- `PConsole` (класс «консоль» пакета `presentation`).

13.3.1. Класс PMain

Функция метода `main()` (основная) тривиальна (и это справедливо), как демонстрируют листинги 13.9 и 13.10. Метод `main()` содержит только одну функцию — `new PMain()` (создание нового объекта PMain) в строке 23. Для этого используется конструктор `PMain()`, который затем вызывает конструктор класса `PConsole` (класс «консоль» пакета `presentation`). Ссылка на объект `PConsole` размещается в переменной `console` (консоль). Переменная используется, чтобы послать сообщение `displayLogin()` (отобразить регистрационное имя) объекту `PConsole`, а в действительности, передать управление объекту `PConsole`, таким образом, полностью обеспечивая работу приложения.

Листинг 13.9. Сводка методов класса PMain()

```
PMain()
        Конструирует объект PMain в начале работы приложения.
static void   main(java.lang.String[] args)
        Основной метод — начало работы приложения.
```

Листинг 13.10. PMain.java

PMain.java

```
9:   public class PMain {
11:     public PMain() {
12:        try {
13:           PConsole console = new PConsole();
14:           console.displayLogin();
15:        }
16:        catch ( Exception exc ) {
17:           exc.printStackTrace();
18:        }
19:     }
22:     public static void main( String[] args ) {
23:        new PMain();
24:     }
25:  }
```

13.3.2. Класс PConsole

Класс `PConsole` (класс «консоль» пакета `presentation`) представляет консольное окно, которое итерация 1 предоставляет пользователю, чтобы взаимодействовать с приложением. Листинг 13.11 содержит резюме методов в `PConsole`.

Листинг 13.11. Сводка методов класса PConsole

PConsole()	
	Создание нового консольного объекта.
PConsole(CActioner actioner)	
	Создание нового консольного объекта для заданного исполнителя.
private void	**display**(java.lang.String s)
	Отображение строки на экране.
private void	**displayConfirmation**()
	Сообщение по электронной почте успешно передано, информирование об этом пользователя.
private void	**displayEmailFailure**(java.lang.String msg)
	Неудачная попытка передачи сообщения по электронной почте.
void	**displayLogin**()
	Отображение экрана для ввода регистрационного имени и организация цикла, пока пользователь правильно введет это имя или пока не будет превышено допустимое число попыток.
private boolean	**displayLoginError**(java.lang.Object status)
	Информирование пользователя о неправильном вводе регистрационного имени.
void	**displayMenu**()
	Отображение главного меню приложения.
private void	**displayMessages**(java.util.Collection msgs, boolean detail)
	Отображение сведений об исходящем сообщении на экран и, если переменная `detail` (детали) установлена, выводится также и тело исходящего сообщения.
private void	**displayMessageText**(java.lang.String input)
	Отображение содержимого исходящего сообщения.
private	**getUserInput**(java.lang.String[] prompts)
java.lang.Object	Получение входной информации от пользователя в соответствии с текущим состоянием программы.
private boolean	**prepareMessage**(java.lang.String msgID)
	Подготовка исходящего сообщения, заданного своим MsgID (идентификатор сообщения), для отправки.
private	**readInput**()
java.lang.String	Чтение входной строки.
private void	**tooManyMessages**(int num)
	Сообщение, что имеется слишком много исходящих сообщений, чтобы их можно было вывести на экран. Не все они извлечены из БД.
private void	**viewMessages**()
	Просмотр списка непосланных сообщений.

Элементы данных, объявленные в PConsole, показаны в листинге 13.12. Обратите внимание на ассоциацию PConsole с CActioner (класс «исполнитель» пакета control).

Листинг 13.12. Поля в PConsole.java

Поля в PConsole.java

```
17:  public class PConsole {
18:     private BufferedReader in;
19:     private PrintStream out;
20:     private static final String VIEW_MESSAGE = "1";
21:     private static final String VIEW_TEXT_MESSAGE = "2";
22:     private static final String SEND_MESSAGE = "3";
23:     private static final String QUIT = "4";
24:
25:     /** Запрос информации о регистрационном имени */
26:     private static final int LOGIN_STATE = 1;
27:
28:     /** Отображение меню и запрос выбора пользователя */
29:     private static final int NORMAL_STATE = 2;
30:
31:     /** Состояние, в котором мы находимся */
32:     private int state;
33:
34:     /** Ассоциация с уровнем control (управление) */
35:     private CActioner actioner;
36:
```

Конструирование объекта PConsole

Итерация 1 содержит два конструктора для создания объекта PConsole (листинг 13.13). Объект PMain — основной (листинг 13.10, строка 13) запускает конструктор по умолчанию (строки 38–40). Конструктор, который создает PConsole, определяя объект CActioner (строки 43–47), в итерации 1 не используется — он упомянут в одном из вопросов учебного примера в конце главы.

Листинг 13.13. Конструкторы PConsole()

Конструкторы PConsole()

```
38:     public PConsole() {
39:        this(new CActioner());
40:     }
41:
43:     public PConsole(CActioner actioner) {
```

```
44:         in = new BufferedReader(new InputStreamReader
            (System.in));
45:         out = System.out;
46:         this.actioner = actioner;
47:     }
```

Листинг 13.14. Метод displayLogin() в PConsole

Метод displayLogin() в PConsole

```
58:     public void displayLogin() {
59:       display("\t\tEMAIL MANAGEMENT SYSTEM -
            ITERATION 1\n\n");
60:        state = LOGIN_STATE;
61:
62:       // цикл, пока пользователь не введет правильную
             информацию[2]
63:        while (true){
64:           Object status = getUserInput(new String[] {
65:           "Please enter your username: ",
66:           "Please enter your password: " });
67:
68:           if (displayLoginError(status)) break;
               // Вероятно, регистрационное имя правильное
69:        }
70:
71:        display("\n");
72:        state = NORMAL_STATE;
73:
74:        // продолжение нормального выполнения программы,
75:        // если регистрационное имя правильное
76:        displayMenu();
77:     }
```

Листинг 13.15. Метод displayLoginError() в PConsole

Метод displayLoginError() в PConsole

```
295:    private boolean displayLoginError(Object status) {
296:       if(status == CActioner.LOGIN_OK) return true;
297:       else if(status == CActioner.LOGIN_FAIL_1){
```

[2] Следует иметь в виду, что переносить комментарий на следующую строку таким образом, как показано в операторе 62, нельзя. Подобный комментарий может занимать только одну строку. По-видимому, это произошло из-за недостатка размера строки в книге. На Web-сайте книги, естественно, все изображено правильно. Аналогичные ситуации имеются и в других листингах. (*Прим. перев.*)

```
298:          display("Invalid login. Please try again.
                       This will be your second try.\n");
299:          return false;
300:       }else if(status == CActioner.LOGIN_FAIL_2){
301:          display("Invalid login. Please try again.
                       This will be your last try.\n");
302:          return false;
303:       }else if(status == CActioner.LOGIN_FAIL_3){
304:         display("You have failed 3x to login. Program will
               quit.\n");
305:          actioner.exit();
306:          return false;
307:       }
308:          return false;
309:    }
```

Отображение регистрационного имени и меню

После запуска приложения пользователя запрашивают об его регистрационном имени. Это выполняет метод `displayLogin()` — отобразить регистрационное имя (листинг 13.14). Неправильные регистрационные имена обрабатываются методом `displayLoginError()` — отобразить сообщение об ошибке ввода регистрационного имени (листинг 13.15), вызываемым в строке 68. Правильное регистрационное имя завершается сообщением к методу `displayMenu()` (строка 76). Метод `displayMenu()` показан в листинге 13.16.

Листинг 13.16. Метод displayMenu() в PConsole

Метод displayMenu() в PConsole

```
160:    public void displayMenu() {
161:       state = NORMAL_STATE;
162:       StringBuffer sb =
163:         new StringBuffer
             ("\t\tEMAIL MANAGEMENT SYSTEM - ITERATION 1\n\n");
164:       sb.append("\t1. View Unsent Messages\n");
165:       sb.append("\t2. Display Text of a Message\n");
166:       sb.append("\t3. Email a Message\n");
167:       sb.append("\t4. Quit this Program\n\n");
168:       sb.append("Please write an option number
                     [1, 2, 3 or 4]: ");
169:
170:       while (true)
171:       getUserInput(new String[] { sb.toString()});
173:    }
```

Просмотр исходящих сообщений

Листинги от 13.17 до 13.20 объясняют, как исходящие сообщения представляются в консольном окне. Метод `viewMessages()` (просмотр сообщений) в листинге 13.17 запрашивает `CActioner`, чтобы извлечь сообщения из БД (строка 179). Чтобы действительно показать извлеченные исходящие сообщения на экране, запрашивается метод `displayMessages()` — отобразить сообщения (строка 193). Метод `displayMessages()` показан в листинге 13.18. Вызов метода `tooManyMessages()` (слишком много сообщений) в строке 196 предназначен, чтобы сообщить пользователю, представляют ли отображенные исходящие сообщения все сообщения, находящиеся в БД. Метод `tooManyMessages()` представлен в листинге 13.20.

Метод `displayMessages()` (листинг 13.18) пробегает коллекцию объектов `EOutMessage` (класс «исходящее сообщение» пакета `entity`) и отображает на экран одновременно одно исходящее сообщение. Если аргумент по имени `detail` (детали) установлен, то текст исходящего сообщения также отображается (строки 224–231). Это происходит, когда пользователь выбирает вариант 2 в списке меню (листинг 13.16, строка 165). Такие опции приводят к запуску метода `displayMessageText()` — отобразить текст сообщения (листинг 13.19).

Листинг 13.17. Метод viewMessages() в PConsole

Метод viewMessages() в PConsole

```
176:    private void viewMessages() {
177:       Collection msgs = new ArrayList();
178:       int remainder =
179:          actioner.retrieveMessages(msgs,
                 IAConstants.MAX_MESSAGE);
180:       Iterator it = msgs.iterator();
181:       IAOutMessage msg;
182:       StringBuffer buf = new StringBuffer();
183:       if (msgs.size() == 0)
184:          buf.append("No unsent query messages.");
185:       else {
186:       buf.append("There are ").append(msgs.size()).append(
187:             " unsent query message(s), as below:\n\n");
188:       }
190:       display(buf.toString());
191:       buf = null;
192:
193:       displayMessages(msgs, false);
194:
195:       if (remainder > 0)
196:          tooManyMessages(remainder);
198:    }
```

Листинг 13.18. Метод displayMessages() в PConsole

Метод displayMessages() в PConsole

```
206:    private void displayMessages(java.util.Collection msgs,
                                     boolean detail) {
207:       for (Iterator it = msgs.iterator(); it.hasNext();) {
208:          IAOutMessage msg = (IAOutMessage) it.next();
209:             IAContact contact = msg.getContact();
210:          StringBuffer buf = new StringBuffer();
211:          buf.append
                 ("Message ID:").append(msg.getMessageID())
                                                 .append("\n");
212:          buf.append
                  ("Date created: ").append(msg.getCreationDate())
213:                                               .append("\n");
214:          buf.append
                 ("Message Subject: ").append(msg.getSubject())
                                                 .append("\n");
215:          buf.append("Contact Name: ")
216:             .append(contact.getFirstName())
217:             .append(" ")
218:             .append(contact.getFamilyName())
219:             .append("\n");
220:          buf.append("Organization: ").append(
221:             contact.getOrganization()).append(
222:             "\n");
223:
224:          if (detail)
225:             buf.append
                    ("Message Text: \n").append(msg.
226:                 getMessageText()).append("\n");
228:             buf.append("\n");
                   //дополнительная пустая строка,
                    чтобы отделить каждое сообщение от других
229:             display(buf.toString());
230:             buf = null;
231:             }
232:    }
```

Метод `displayMessageText()` (листинг 13.19) показывает полное содержание исходящего сообщения. Поскольку состояние БД могло измениться между временем, когда исходящее сообщение было внесено в список отображаемых на экран, и временем, когда пользователь потребовал его показ, метод просит, чтобы `CActioner` выполнил новый поиск исходящего сообщения в БД (строка 146). После успешного поиска вызывается метод `displayMessages()` с аргументом `detail`, имеющим значение `true` (истина).

Листинг 13.19. Метод displayMessageText() в PConsole

Метод displayMessageText() в PConsole

```
143:    private void displayMessageText(String input) {
144:       try {
145:          int i = Integer.parseInt(input);
146:          IAOutMessage msg = actioner.retrieveMessage(i);
147:          if (msg == null) {
148:             display("No such message exist.\n\n");
149:             return;
150:          }
151:          displayMessages(java.util.Collections.
                              singleton(msg),
              true);
152:       } catch (NumberFormatException exc) {
153:          display("Invalid Message ID.\n\n");
154:       } catch (Exception exc2) {
155:          display("Could not retrieve the message\n\n");
156:       }
157:    }
```

Листинг 13.20. Метод tooManyMessages() в PConsole

Метод tooManyMessages() в PConsole

```
280:    private void tooManyMessages(int num) {
281:       if (num == 0)
282:          return;
283:       if (num == 1)
284:          display("There is "
285:             + num
286:             + " unsent outmessage left unretrieved in
                    database.\n\n");
287:       else
288:          display(
289:             "There are "
290:             + num
291:             + " unsent outmessages left unretrieved in
                    database.\n\n");
292:    }
```

Требование к передаче по электронной почте исходящего сообщения

Пользователь запрашивает передачу по электронной почте исходящего сообщения, выбирая опцию 3 в списке меню (листинг 13.16, строка 166). Это активизирует метод `prepareMessage()` — подготовить сообщение (листинг 13.21). Метод заново извлекает данные из БД, находящейся в текущем состо-

янии (строки 242–243), затем формирует содержание исходящего сообщения и требует от CActioner послать его по электронной почте (строка 257). В зависимости от результата метод просит сотрудничающие методы показать пользователю подтверждение передачи (строка 260) или ошибку (строка 262).

Листинг 13.21. Метод prepareMessage() в PConsole

Метод prepareMessage() в PConsole

```
239:    private boolean prepareMessage(String msgID) {
240:       try {
241:          int id = Integer.parseInt(msgID);
242:          IAOutMessage msg = actioner.retrieveMessage(id);
243:          IAEmployee emp = actioner.getEmployee();
244:          IAContact contact = msg.getContact();
245:
246:          String subject = msg.getSubject();
247:          String from = emp.getEmail();
248:          String to = contact.getEmail();
249:
250:          StringBuffer body = new StringBuffer();
251:          body.append
                 ("Dear ").append(contact.getFirstName()).
                                                 append("\n\n"),
252:          body.append(msg.getMessageText());
253:          body.append("\n\nRegards,\n\n").append(

254:             emp.getFirstName()).append("\n");
255:
256:          boolean b =
257:             actioner.sendMessage(msg, from, to, subject,
                                              body.toString());
258:
259:          if (b)
260:             displayConfirmation();
261:          else
262:             displayEmailFailure
                    ("no network connection or mail address
                    invalid.\n");
263:
264:          return b;
265:       } catch (Exception exc) {
266:          displayEmailFailure(exc.getMessage());
267:       }
268:        return false;
269:    }
```

13.4. Пакет Control

Пакет `control` (управление) содержит только один класс — `CActioner` (класс «исполнитель» пакета `control`). Этот класс инкапсулирует все операции, известные в итерации 1 из учебного примера ЕМ.

13.4.1. Класс CActioner

Листинг 13.22 представляет сводку методов класса `CActioner` (класс «исполнитель» пакета `control`). Поля класса `CActioner` перечислены в листинге 13.23.

Элементы данных, объявленные в `PConsole` (класс «консоль» пакета `presentation`), показаны в листинге 13.12. Обратите внимание на ассоциацию `PConsole` и `CActioner`.

Листинг 13.22. Сводка методов класса CActioner

`CActioner()`	
	Создание нового объекта класса `CActioner`.
`CActioner``(MBroker broker)`	
	Создание нового объекта класса CActioner с заданием объекта класса MBroker.
`void`	**`exit`**`()`
	Завершение приложения.
`IAEmployee`	**`getEmployee`**`()`
	Извлечение текущего зарегистрированного служащего.
`private void`	**`handleEmailException`**`(java.lang.Exception exc)`
`java.lang.Object`	**`login`**`(java.lang.String username, java.lang.String passwd)`
	Регистрация пользователя с заданными пользовательским именем и паролем.
`void`	**`logout`**`()`
	Удаление пользователя без завершения приложения.
`IAOutMessage`	**`retrieveMessage`**`(int id)`
	Извлечение исходящего сообщения по его идентификатору.
`int`	**`retrieveMessages`**`(java.util.Collection msgs, int numMsgs)`
	Извлечение `numMsgs` непосланных исходящих сообщений в `msgs` (список сообщений).
`private boolean`	**`send`**`(java.lang.String from, java.lang.String to, java.lang.String subject, java.lang.String body)`
	Послать сообщение.
`boolean`	**`sendMessage`**`(IAOutMessage msg)`
	Послать сообщение и скорректировать БД в случае успешной передачи.
`boolean`	**`sendMessage`**`(IAOutMessage msg, java.lang.String from, java.lang.String to, java.lang.String subject, java.lang.String body)`
	Послать сообщение и скорректировать его в случае успешной передачи.

Листинг 13.23. Поля в CActioner.java

Поля в CActioner.java

```
15:  public class CActioner {
16:     /** Регистрационное имя правильное */
17:     public static final Object LOGIN_OK = "LOGIN OK";
18:
19:/** Первая попытка при неправильном регистрационном имени */
20:   public static final Object LOGIN_FAIL_1 = "LOGIN FAIL 1";
21:
22:/** Вторая попытка при неправильном регистрационном имени */
23:   public static final Object LOGIN_FAIL_2 = "LOGIN FAIL 2";
24:
25:   /** Заключительная попытка при неправильном
          регистрационном имени */
26:   public static final Object L0GIN_FAIL_3 = "LOGIN FAIL 3";
27:
28:     /**
29:      * Разрешено максимальное число попыток неправильного
           ввода
30:      *  регистрационного имени перед завершением программы
31:      */
32:     private static final int MAX_LOGIN_TRY = 3;
33:
34:     /** Ассоциация с посредником */
35:     private MBroker broker;
36:
37:     /** Единственный объект EEmployee, который мы имеем */
38:     private IAEmployee emp;
39:
40:     /** Пока выполняются попытки ввести регистрационное имя
         */
41:     private volatile int loginTry = 0;
42:     // Временный, потому что мы не держим его дольше, чем
            необходимо
```

Конструирование объекта CActioner

Итерация 1 содержит два конструктора для создания объекта `CActioner` (листинг 13.24). Объект `PConsole` (листинг 13.13, строка 39) вызывает конструктор, используемый по умолчанию (строки 45–52). Конструктор, который создает `CActioner`, определяя объект `MBroker` — класс «посредник» пакета `mediator` (строки 55–57), используется в решении задач.

Листинг 13.24. Конструкторы класса CActioner

Конструкторы класса CActioner

```
45:     public CActioner() {
46:        try {
47:           broker = new MBroker();
48:        }catch ( Exception exc ) {
49:           exc.printStackTrace();
50:           exit(); //завершить приложение, если это
                      происходит
51:        }
52:     }
55:     public CActioner( MBroker broker ) {
56:        this.broker = broker;
57:     }
```

Инициализация регистрационного имени

`PConsole` делает ответственным класс `CActioner` за проверку регистрационного имени пользователя в БД и, следовательно, за разрешение продолжения работы пользователя с приложением. Метод `login()` — регистрационное имя (листинг 13.25) получает объекты `String username` (строка с пользовательским именем) и `String passwd` (строка с паролем) — строка 65 — и передает их в вызове объекту `MBroker` (строка 67). Статус регистрационного имени, возвращаемого объекту класса `PConsole` — постоянная величина.

Листинг 13.25. Метод login() в CActioner

Метод login() в CActioner

```
60:        * Регистрационное имя пользователя, в котором даны
           * пользовательское имя и пароль
61:        * @param — пользовательское имя (имя пользователя
           * при подключении)
62:        * @param — пароль (пароль для БД)
63:        * @return — статус регистрационного имени:
           * LOGIN_OK, LOGIN_FAIL_1 и т. д.
64:        */
65:     public Object login( String username, String passwd ) {
66:        loginTry++;
67:        emp = broker.login( username, passwd );
68:
```

```
69:       if(emp != null) {
70:          LoginTry=0; // переустановка
71:          return LOGIN_OK;
72:       }else{
73:          switch(loginTry){
74:             case 1: return LOGIN_FAIL_1;
75:             case 2: return LOGIN_FAIL_2;
76:             default: return LOGIN_FAIL_3;
77:          }
78:       }
79:    }
```

Листинг 13.26. Метод retrieveMessages() в CActioner

Метод retrieveMessages() в CActioner

```
160:    public int retrieveMessages( Collection msgs,
                                     int numMsgs ) {
161:       return broker.retrieveUnsentMessages( emp, msgs,
                                          numMsgs );
165:    }
```

Поиск исходящих сообщений

Требование извлечь исходящие сообщения из БД поступает к `CActioner` от `PConsole` (листинг 13.17, строка 179). `CActioner` делегирует этот сервис объекту класса `MBroker`. Метод `retrieveMessages()` — извлечь сообщения (листинг 13.26) требует, чтобы `MBroker` извлек до `numMsgs` (число сообщений) исходящих сообщений и разместил их в коллекцию `msgs` — сообщения (строка 161). Должны быть извлечены только исходящие сообщения, созданные для обработки только тому служащему, который передается в аргументе `emp` (служащий). `CActioner` поддерживает ассоциацию с `emp` (листинг 13.23, строка 38). Метод `retrieveMessages()` (извлечь сообщение) возвращает число `int` (целое) исходящих сообщений, оставшихся неизвлеченными из БД (строка 160).

Листинг 13.27. Метод retrieveMessage() в CActioner

Метод retrieveMessage() в CActioner

```
172:    public IAOutMessage retrieveMessage( int id ) {
173:       return broker.retrieveUnsentMessage( emp, id )
174:    }
```

Листинг 13.28. Перегружаемые методы sendMessage() в CActioner

Перегружаемые методы sendMessage() в CActioner

```
97:     public boolean sendMessage(IAOutMessage msg){
98:       return sendMessage(msg,
                             msg.getSenderEmployee().getEmail(),
99:                          msg.getContact().getEmail(),
                             msg.getSubject(),
                             msg.getMessageText());
100:    }
103:    public boolean sendMessage(IAOutMessage msg,
                                   String from,
104:    String to, String subject, String body ) {
106:       try{
107:          if ( send( from, to, subject, body ) ) {
108:         return broker.updateMessage( msg.getMessageID() );
109:          }
110:       }catch(Exception exc){
111:          handleEmailException(exc);
112:       }
113:       return false;
114:    }
```

Метод `displayMessageText()` (отобразить текст сообщения) в `PConsole` запрашивает извлечение всех деталей единственного исходящего сообщения (листинг 13.19, строка 146). Сервис для этого запроса обеспечивается методом `retrieveMessage()` в `CActioner` (листинг 13.27). `CActioner` делегирует запрос дальше к `MBroker` (строка 173). Метод `retrieveMessage()` возвращает объект `EOutMessage` — класс «исходящее сообщение» пакета `entity` (строка 172). Это должно быть объектом класса `EOutMessage` (или пустым указателем, если ничто не было извлечено), потому что `EOutMessage` — единственный класс, который реализует интерфейс `IAOutMessage` (интерфейс «исходящее сообщение» пакета `entity`).

Передача по электронной почте исходящего сообщения

`CActioner` отвечает за передачу по электронной почте исходящего сообщения, когда это необходимо пользователю. `CActioner` имеет два перегружаемых метода, называемые `sendMessage()` (послать сообщение), чтобы выполнить эту услугу (листинг 13.28). В итерации 1 используется только один из этих методов. `PConsole` запрашивает услуги второго метода (строки 103–114). Этот метод вызывается из `PConsole` (листинг 13.21, строка 257). Если исходящее сообщение успешно послано, `MBroker` корректирует запись исходящего сообщения в БД (строка 108). Фактическая операция передачи по электронной почте выполняется методом `send()` — послать (строка 107),

который использует библиотеку JavaMail™ API, чтобы выполнить эту работу (см. следующий раздел).

Использование JavaMail™ API

Библиотека **JavaMail™** API — обычно известная как **MAPI** (Mail API — интерфейс API электронных сообщений) — является библиотекой для чтения, создания и передачи сообщения по электронной почте. JavaMail™ API поддерживает различные протоколы передачи по электронной почте, включая SMTP (Simple Mail Transfer Protocol — простой протокол электронной почты), POP (Post Office Protocol — почтовый протокол) и IMAP (Interactive Mail Access Protocol — протокол интерактивного доступа к электронной почте). Итерация 1 использует JavaMail™ API, чтобы послать сообщение по электронной почте через Simple Mail Transfer Protocol (SMTP). Сервер SMTP отправителя сообщения по электронной почте транспортирует сообщение серверу SMTP получателя сообщения. Получателю может быть передано сообщение через протоколы POP или IMAP.

JavaMail™ API зависит от JavaBeans Activation Framework (JAF — шаблон активизации компонентов JavaBeans). JAF использует формат Multipurpose Internet Mail Extension (MIME — многоцелевые расширения электронной почты в Интернете). Класс `MimeMessage` (сообщение MIME) используется, чтобы создать объекты, которые понимают типы и заголовки MIME.

В итерации 1 передача по электронной почте выполняется методом `send()` (листинг 13.29), вызываемым методом `sendMessage()` (листинг 13.28, строка 107). При вызове метод `send()` получает электронные адреса `from` (отправитель) и `to` (получатель), а также тему сообщения и его содержание. Строки `from` и `to` передаются конструктору класса `InternetAddress` (адрес Интернета), чтобы инициализировать объекты адреса для полей `from` и `to` сообщений (строки 123–125).

Объект `Properties` — свойства (строки 128–131) используется, чтобы поместить информацию типа адреса сервера электронной почты, пользовательского имени и пароля. Этот объект передается конструктору объекта `Session` — сеанс (строка 134). В итерации 1 достаточен единственный используемый по умолчанию сеанс электронной почты. Объект `Session` затем передается конструктору `MimeMessage`, чтобы создать объект `Message` — сообщение (строка 137). Строки 139–146 устанавливают компоненты объекта `Message`. Наконец, объект `Message` передается методу `send()` объекта `Transport` (передача), который выполняет передачу сообщения по электронной почте.

13.5. Пакет Entity

Пакет `entity` (сущность) состоит из одного интерфейса и трех классов:

- `IEDataSupplier` (интерфейс «источник данных» пакета `entity`);
- `EContact` (класс «деловой партнер» пакета `entity`);
- `EEmployee` (класс «служащий» пакета `entity`);
- `EOutMessage` (класс «исходящее сообщение» пакета `entity`).

13.5.1. Интерфейс IEDataSupplier

Интерфейс `IEDataSupplier` (интерфейс «источник данных» пакета `entity`) отвечает за предоставление данных объектам-сущностям и за обеспечение для них возможности поддерживать связи ассоциации друг с другом. В результате этого объекты-сущности, находясь в кэше памяти, представляют точный снимок подмножества состояния БД.

Листинг 13.29. Приватный метод send() в CActioner (JavaMail™ API)

Приватный метод send() в CActioner (JavaMail™ API)

```
121:    private boolean send( String from, String to,
                  String subject, String body )
122:    throws Exception{
123:     InternetAddress fromAddr = new InternetAddress(from );
124:     InternetAddress[] toAddr = new InternetAddress[ 1 ];
125:     toAddr[ 0 ] = new InternetAddress ( to );
126:
127:     // конструирование классов MAPI
128:     java.util.Properties props =
                         new java.util.Properties();
129:
130:     // мы предполагаем, что здесь используем SMTP
131:     props.put{ "mail.smtp.host", constants.SMTP_ADDRESS );
132:
133:       // задание новой сессии работы с электронной почтой
134:       javax.mail.Session session =
          javax.mail.Session.getDefaultInstance( props, null );
135:
136:       // конструирование сообщения электронной почты
137:       javax.mail.Message msg1 =
               new javax.mail.internet.MimeMessage( session );
138:
139:   msg1.setRecipients( javax.mail.Message.RecipientType.TO,
                            toAddr );
140:       msg1.setFrom( fromAddr );
141:
142:       msg1.setSubject( subject );
143:       msg1.setSentDate( new java.util.Date() );
144:
145:       // если желаемый набор символов известен, можно
                  использовать setText(text, charset)
146:       msg1.setText( body );
147:
148:       // наконец, передача сообщения по электронной почте
149:       javax.mail.Transport.send( msg1 );
150:
151:       return true;
152:    }
```

Интерфейс IEDataSupplier обеспечивает также разъединение пакетов entity (сущность) и mediator (посредник), чтобы устранить циклическую зависимость (раздел 9.1.7). Интерфейс реализован классом MBroker (класс «посредник» пакета mediator) из пакета mediator и используется всеми тремя классами пакета entity.

Листинги 13.30 и 13.31 показывают детали интерфейса IEDataSupplier.

Листинг 13.30. Сводка методов класса IEDataSupplier

IAContact	**getContact**(int contactOID) Не может использоваться извне, только для классов пакета entity.
IAEmployee	**getEmployee**(int empOID) Не может использоваться извне, только для классов пакета entity.
int	**retrieveUnsentMessages**(IAContact contact, Java.util.Collection msgs, int numMsgRetrieve) Извлечение числа numMsgRetrieve (число извлеченных сообщений) исходящих сообщений для конкретного contact (деловой партнер) и размещение их в коллекции msgs (сообщения).
int	**retrieveUnsentMessages**(IAEmployee emp, Java.util.Collection msgs, int numMsgRetrieve) Извлечение числа numMsgRetrieve исходящих сообщений для конкретного emp (служащий) и размещение их в коллекции msgs.

Листинг 13.31. Класс IEDataSupplier.java

IEDataSupplier.java

```
12:  public interface IEDataSupplier {
13:     /**
17:     * @param emp — служащий, ответственный за эти сообщения
18:     * @param msgs — коллекция, в которую эти исходящие
                        сообщения будут помещены
19:     * @param numMsgRetrieve — максимальное число
           извлекаемых исходящих сообщений
20:     * @return int — число исходящих сообщений, все еще
                        остающихся неизвлеченными из БД
21:     */
22:     public int retrieveUnsentMessages(IAEmployee emp,
                Collection msgs, int numMsgRetrieve);
23:
24:     /**
28:      * @param contact — деловой партнер, которому
             адресованы исходящие сообщения
32:      */
33:     public int retrieveUnsentMessages(IAContact contact,
                        Collection msgs, int numMsgRetrieve);
34:
```

```
36:     public IAEmployee getEmployee(int empOID)
                    throws Exception;
37:
39:     public IAContact getContact(int contactOID);
40:  }
```

Идентификаторы объектов и паттерн Поле идентификации

Интерфейс `IEDataSupplier` проясняет, что каждому объекту-сущности дается уникальный **идентификатор объекта** (object identifier — OID) [61]. Идентификаторы объектов затем используются для связей ассоциации между объектами, находящимися в кэше памяти. В процессе выполнения ЕМ каждому объекту-сущности дается глобально уникальный идентификатор объекта, когда он впервые создается. Этот идентификатор гарантирует уникальность на протяжении всего выполнения программы.

Идентификаторы объектов, располагаемых в памяти, должны в первую очередь использоваться в объектных СУБД [61]. В ЕМ реализована менее сложная технология. Различные пути реализации идентификаторов объектов являются предметом **паттерна Поле идентификации** [31].

Паттерн Поле идентификации объявляет, что поле идентификатора «сохраняет поле идентификации БД в объекте, чтобы сохранить идентичность между объектом, находящимся в памяти, и строкой в БД» [31]. Другими словами, поле идентификации гарантирует надлежащее соответствие между временными объектами в памяти и сохраняемыми объектами в БД. В самом простом случае паттерн Поле идентификации использует в качестве идентификаторов объектов, располагаемых в памяти, первичный ключ записи БД. Это позволяет предъявить к паттерну меньшее число требований, чем к реализации объектной БД или даже ЕМ-реализации.

Использование первичного ключа в качестве идентификатора объекта проблематично, по крайней мере, по следующим причинам:

- Часто первичные ключи хорошо отражают понятия бизнеса и используются в этом качестве в бизнес-отношениях. Это означает, что они могут через какое-то время измениться. Они также гарантируют, что будут уникальными только в пределах записей одной таблицы, что может создать проблемы, когда будут использоваться для объектов, размещаемых в памяти. Объекты, располагаемые в памяти, используют поля идентификации в качестве средства обработки объектов, независимо от того, знают ли те о классе этих средств обработки объектов.
- Даже если первичные ключи могут обеспечить функции идентификаторов для объектов, созданных в результате чтения записей из БД, приложение, вероятно, создаст новые объекты-сущности в памяти и затем уже будет их размещать на постоянное хранение в БД. Большинство БД имеет собственные механизмы автоматического формировния значения первичного ключа, когда новая запись помещается в таблицу (этот механизм в SQL*SERVER и Sybase даже называется *идентификацией*; в Oracle он называется *последовательностью*). Наличие в памяти первичного ключа, который должен быть заменен формируемым самой БД первичным ключом при записи данных в БД, является, мягко говоря, неуклюжим.

- «Уникальные для таблицы» первичные ключи создают проблемы, когда используются как идентификаторы объектов в классах, которые применяют наследование реализации.
- Реализация идентификаторов объектов в виде полей глобальных уникальных идентификаторов может оказаться очень полезной позже, когда будет принято решение поместить пакет `entity` в информационное хранилище объектов среднего уровня многоуровневой структуры.

Поле идентификации в EM реализуется объединением первых трех букв имени сущности (исключая первую букву, которая определяет пакет — например, `EMP` в случае имени `EEmployee`) со случайно сформированным уникальным целым числом. По этой причине поля идентификации являются идентификаторами объектов, «уникальными для выполнения», то есть, они уникальны во время выполнения программы.

Наконец, обратите внимание, что поля идентификации используются в дополнение к любой ситуации, когда объект также содержит другой объект или содержит ссылку на контейнер других объектов. Как указано Фаулером [31], тот факт, что системы объектов обеспечивают правильное определение «содержания» покрытий (в Java) или указателей на память (в C++), не заменяет необходимости в полях идентификации.

13.5.2. Класс EEmployee

Класс `EEmployee` (класс «служащий» пакета `entity`) реализует интерфейс `IAEmployee` — интерфейс «служащий» пакета `acquaintance` (раздел 13.2.2). Этот класс поддерживает связи ассоциации ко всем исходящим сообщениям (объекты `EOutMessage` — класс «исходящее сообщение» пакета `entity`), назначенным ему и в текущий момент загруженным в память.

Объект-клиент может извлечь исходящие сообщения в соответствии с различными критериями, такими как запрос объекта `EEmployee` относительно его исходящих сообщений, запрос объекта `EContact` (класс «деловой партнер» пакета `entity`) для определения адресованных к нему исходящих сообщений или запрос объекта `MBroker` (класс «посредник» пакета `mediator`) для нахождения требуемых исходящих сообщений. Листинг 13.32 показывает элементы данных `EEmployee`.

Листинг 13.32. Сводка полей класса EEmployee

```
private java.lang.String      email
private java.lang.String      employeeID
private java.lang.String      familyName
private java.lang.String      firstName
private java.lang.String      loginName
private java.util.Collection  msgs
private int                   OID
Private IEDataSupplier        ref
```

Листинг 13.33 объясняет основные методы класса EEmployee. Методы get и set в EEmployee — простых типов данных и здесь исключены.

Листинг 13.33. Сводка методов класса EEmployee

```
EEmployee(IEDataSupplier ref)
                Простой конструктор, используемый по умолчанию.
java.util.Collection  getUnsentOutMessages(int maxMsgsRetrieve)
                Получение всех сообщений (вплоть до numMsgRetrieve — число
                извлекаемых сообщений), предназначенных данному служащему.
void                  removeSentOutMessage(IAOutMessage msg)
                Удаление сообщения из этого списка для служащего, по-видимому,
                из-за того, что оно уже послано.
```

Конструирование объекта EEmployee

Объект EEmployee инициализируется классом MBroker, который реализует интерфейс IEDataSupplier (интерфейс «источник данных» пакета entity). Конструктор для EEmployee показан в листинге 13.34.

Листинг 13.34. Конструктор EEmployee()

Конструктор EEmployee()

```
28:     public EEmployee(IEDataSupplier ref) {
29:        this.ref = ref;
30:     };
```

Получение непосланных сообщений

EEmployee может использовать переменную ref (ссылка), которая указывает на объект MBroker, чтобы запросить извлечение исходящих сообщений, которые еще не были посланы. Число извлеченных исходящих сообщений ограничено значением аргумента maxMsgsRetrieve (максимальное число извлекаемых сообщений). Метод getUnsentMesages() показан в листинге 13.35.

Листинг 13.35. Метод getUnsentMessages() в EEmployee

Метод getUnsentMessages() в EEmployee

```
90:     public Collection getUnsentOutMessages(int
                                   maxMsgsRetrieve) {
91:        if (msgs == null)
92:           ref.retrieveUnsentMessages( this, msgs,
                                          maxMsgsRetrieve );
93:        return msgs;
94:     }
```

В итерации 1 все запросы от CActioner (класс «исполнитель» пакета control) для передачи непосланных сообщений в кэш и удаления посланных сообщений из кэша адресуются к MBroker. Сделано это потому, что MBroker управляет состоянием кэша. Метод GetUnsentMessages() (получить непосланные сообщения) и метод removeSentOutMessages() (удалить посланные исходящие сообщения), представленный в следующем разделе, не используются в итерации 1. Вызываемые классом CActioner, они могли бы использоваться для связи с MBroker. Тот же принцип применяется к подобным методам в EContact, рассмотренным в разделе 13.5.3.

Удаление посланных исходящих сообщений

Объект EEmployee содержит переменную коллекции msgs (сообщения), включающей все объекты OutMessage (исходящее сообщение), предназначенные этому EEmployee и находящиеся в настоящее время в кэше памяти. Точно так же каждый объект OutMessage имеет переменную, связывающую его с EEmployee. Как только исходящее сообщение успешно передано по электронной почте, соответствующий объект OutMessage должен быть удален из коллекции (строка 105 в листинге 13.36), и нужно запросить установить его обратную связь к EEmployee в null — пустой указатель (строка 106).

Листинг 13.36. Метод removeSentOutMessage() в EEmployee

Метод removeSentOutMessage() в EEmployee

```
99:     public void removeSentOutMessage(IAOutMessage msg) {
100:       if (msgs == null) return;
103:           // выделение из списка сообщений
104:           // также выделение этого emp из сообщения
105:       if (msgs.remove(msg))
106:          msg.setSenderEmployee(null);
107:    }
```

13.5.3. Класс EContact

Класс EContact (класс «деловой партнер» пакета entity) реализует интерфейс IAContact — интерфейс «деловой партнер» пакета entity (раздел 13.2.3). Каждый объект EContact поддерживает связи ассоциации ко всем исходящим сообщениям (объекты EOutMessage — класс «исходящее сообщение» пакета entity), адресованным ему и загруженным в настоящее время в память.

Листинг 13.37 показывает элементы данных EContact. Использование полей OID (идентификатор объекта) и ref (ссылка) подобно тому, как было рассмотрено в предыдущем разделе в контексте EEmployee (класс «служащий» пакета entity).

Листинг 13.37. Сводка полей класса EContact

```
private java.lang.String       contactID
private java.lang.String       email
private java.lang.String       familyName
private java.lang.String       firstName
private java.util.Collection   msgs
private int                    OID
private java.lang.String       organization
private IEDataSupplier         ref
```

Листинг 13.38 объясняет основные методы класса `EContact`. Методы `get` (получить) и `set` (задать) в `EContact` — простых типов данных и здесь исключены.

Листинг 13.38. Сводка методов класса EContact

```
EContact(IEDataSupplier ref)
                Простой конструктор, используемый по умолчанию.
java.util.Collection  getUnsentOutMessages(int
                      numMsgToBeRetrieved)
                Получение всех сообщений, предназначенных для данного партнера.
void            removeSentOutMessage(IAOutMessage msg)
                Удаление сообщения, предназначенного для данного партнера,
                по-видимому, из-за того, что оно уже послано.
```

Конструирование объекта EContact

Объект `EContact` инициализируется классом `MBroker` (класс «посредник» пакета `mediator`), который реализует интерфейс `IEDataSupplier` (интерфейс «источник данных» пакета `entity`). Конструктор для `EContact` показан в листинге 13.39.

Листинг 13.39. Конструктор EContact()

Конструктор EContact()

```
24:     public EContact(IEDataSupplier ref) {
25:        this.ref = ref;
26:     }
```

Получение непосланных исходящих сообщений

Метод, используемый для получения непосланных исходящих сообщений, относящихся к `EContact` (листинг 13.40), идентичен представленному в листинге 13.35 для `EEmployee`. `EContact` использует переменную `ref`, которая указывает на объект `MBroker`, чтобы запрашивать извлечение исходящих сообщений, которые еще не были посланы. Число извлеченных исходящих сообщений ограничено значением аргумента `maxMsgsRetrieve` (максимальное число извлекаемых сообщений).

Листинг 13.40. Метод getUnsentMessages() в EContact

Метод getUnsentMessages() в EContact

```
88:     public Collection
                getUnsentOutMessages(int numMsgToBeRetrieved) {
89:        if (msgs == null)
              ref.retrieveUnsentMessages(this,msgs,
                                          numMsgToBeRetrieved);
90:        return msgs;
91:     }
```

Удаление посланных исходящих сообщений

Так же как и у объекта EEmployee, объект EContact имеет переменную коллекции msgs, чтобы разместить все объекты OutMessage (исходящее сообщение), предназначенные для этого EContact и в настоящее время находящиеся в кэше памяти. Подобным же образом объект OutMessage имеет переменную, связывающую его с EContact. Как только исходящее сообщение будет успешно передано по электронной почте, соответствующий объект OutMessage должен быть удален из коллекции (строка 102 в листинге 13.41) и нужно запросить установку его обратной связи к EContact в null — пустой указатель (строка 103).

Листинг 13.41. Метод removeSentOutMessage() в EContact

Метод removeSentOutMessage() в EContact

```
96:     public void removeSentOutMessage(IAOutMessage msg) {
97:        if (msgs == null) return;
100:    // выделение из списка сообщений
101:    // также выделение этого делового партнера из сообщения
102:          if (msgs.remove (msg))
103:             msg.setContact(null);
104:    }
105: }
```

13.5.4. Класс EOutMessage

Класс EOutMessage (класс «исходящее сообщение» пакета entity) реализует интерфейс IAOutMessage — интерфейс «исходящее сообщение» пакета acquaintance (раздел 13.2.4). Каждый объект EOutMessage поддерживает связи ассоциации с EEmployee (класс «служащий» пакета entity) и EContact (класс «деловой партнер» пакета entity), если эти объекты в настоящее время загружены в память.

В итерации 1 служащий, который создает исходящее сообщение (creator_emp_id — идентификатор служащего-создателя в таблице OutMessage — исходящее сообщение), также отвечает и за передачу по электронной почте этого исходящего сообщения. После передачи сообщения

столбцы sender_emp_id (идентификатор служащего-отправителя) и date_emailed (дата отправки сообщения по электронной почте) в таблице OutMessage обновляются (рис. 10.14 в разделе 10.3.1).

Листинг 13.42 показывает элементы данных EOutMessage. Использование полей OID (идентификатор объекта) и ref (ссылка) подобно тому, как было рассмотрено ранее в контексте EEmployee и EContact. Поля contactOID (идентификатор объекта-делового партнера), creatorOID (идентификатор объекта-создателя) и senderOID (идентификатор объекта-отправителя) обеспечивают связи с EContact и EEmployee. В итерации 1 creatorOID и senderOID связываются с одним и тем же объектом EEmployee.

Листинг 13.42. Сводка полей класса EOutMessage

```
private IAContact          contact
private int                contactOID
private java.sql.Date      creationDate
private IAEmployee         creator
private int                creatorOID
private int                messageID
private java.lang.String   messageText
private int                OID
private IEDataSupplier     ref
private IAEmployee         sender
private int                senderOID
private java.sql.Date      sentDate
private java.lang.String   subject
```

Кроме конструктора все методы в EOutMessage являются методами get (получить) и set (задать). Листинг 13.43 показывает те методы get и set, которые обращаются с непростыми типами (то есть классами). Другие методы get и set опущены.

Листинг 13.43. Сводка методов класса EOutMessage

```
EOutMessage  (IEDataSupplier ref)
                Простой конструктор, используемый по умолчанию.
IAContact    getContact()
                Получение делового партнера для этого сообщения.
IAEmployee   getCreatorEmployee()
                Извлечение сотрудника, кто первоначально создал это сообщение.
IAEmployee   getSenderEmployee()
                Извлечение сотрудника, который уполномочен послать сообщение.
void         setContact(IAContact contact)
                Позволяет задать делового партнера.
void         setCreatorEmployee(IAEmployee emp)
                Задание сотрудника, который создает сообщение.
void         setSenderEmployee(IAEmployee emp)
                Задание сотрудника, который уполномочен отправить сообщение.
```

Конструирование объекта EOutMessage

Объект EOutMessage инициализируется классом MBroker (класс «посредник» пакета mediator), который реализует интерфейс IEDataSupplier (интерфейс «источник данных» пакета entity). Простой конструктор, используемый по умолчанию для EOutMessage, показан в листинге 13.44.

Листинг 13.44. Конструктор EOutMessage()

Конструктор EOutMessage()

```
27:     public EOutMessage(IEDataSupplier ref) {
28:        this.ref = ref;
29:     }
```

Получение и задание делового партнера для исходящего сообщения

Как показано в листинге 13.42, EOutMessage имеет поле (contactOID), которое может использоваться, чтобы найти объект EContact для конкретного объекта EOutMessage. Однако contactOID может указывать лишь на EContact, уже находящийся в кэше памяти. Если EContact не в кэше, то он должен быть извлечен из БД и связан с EOutMessage. Связывание завершается заданием полей contact и contactOID. Ясно, что задание объекту EOutMessage объекта EContact и ссылка на тот же самый объект избыточны в итерации 1.

Задача получения и задания EContact для EOutMessage возлагается на методы getContact() (получить делового партнера) и setContact() (задать делового партнера), представленные в листинге 13.45. EOutMessage обращается к методу getContact() объекта MBroker (представленного переменной ref) и передает ему contactOID (строка 123 в листинге 13.45). Когда объект EContact будет возвращен из MBroker, EOutMessage находится в состоянии выполнения setContact() (строки 106–108) и возвращает EContact клиенту в методе getContact() (строки 121–125).

Листинг 13.45. Методы getContact() и setContact() в EOutMessage

Методы getContact() и setContact() в EOutMessage

```
121:    public IAContact getContact() {
122:       if (contact == null)
123:          setContact(ref.getContact(contactOID));
124:       return contact;
125:    }
106:    public void setContact(IAContact contact) {
107:       this.contact = contact;
108:    }
```

Получение и задание служащего-создателя для исходящего сообщения

Задача получения и задания объекта EEmployee как служащего-создателя для объекта EOutMessage аналогична получению и заданию EContact для

EOutMessage, как рассмотрено в предыдущем разделе. Это показано в листинге 13.46.

Листинг 13.46. Методы getCreatorEmployee() и setCreatorEmployee() в EOutMessage

Методы getCreatorEmployee() и setCreatorEmployee() в EOutMessage

```
130:    public IAEmployee getCreatorEmployee() {
131:       try {
132:          if (creator == null)
133:    setCreatorEmployee(ref.getEmployee(creatorOID));
139:       return creator;
140:    }
161:    public void setCreatorEmployee(IAEmployee emp) {
162:       creator = emp;
163:    }
```

Получение и задание служащего-отправителя исходящего сообщения

Задача получения и задания объекта EEmployee как служащего-отправителя для EOutMessage показана в листинге 13.47. Снова обратите внимание, что в итерации 1 создатель и отправитель исходящего сообщения — один и тот же служащий.

Листинг 13.47. Методы getSenderEmployee() и setSenderEmployee() в EOutMessage

Методы getSenderEmployee() и setSenderEmployee() в EOutMessage

```
145:    public IAEmployee getSenderEmployee() {
146:       try {
147:          if (sender == null)
148:             setSenderEmployee(ref.getEmployee(senderOID));
155:       return sender;
156:    }
168:    public void setSenderEmployee(IAEmployee emp) {
169:       sender = emp;
170:    }
```

13.6. Пакет Mediator

Пакет mediator (посредник) состоит из одного класса:

- MBroker (класс «посредник» пакета mediator).

Класс MBroker реализует два интерфейса: IAConstants (интерфейс «константы» в пакете acquaintance — знакомство) и IEDataSupplier (интерфейс «источник данных» в пакете entity — сущность).

Листинг 13.48. Сводка полей класса MBroker

```
private FConnection    connection
                          Связь, используемая читающим/записывающим объектом.
private java.util.Map  contacts
                          Кэш деловых партнеров.
private boolean        dirty
                          Вызывает необходимость повторного извлечения кэша.
private java.util.Map  emps
                          Кэш сотрудников.
private java.util.Map  msgs
                          Кэш сообщений.
private FReader        reader
                          Объект чтения БД.
private int            remainder
                          Количество сообщений, оставшихся в БД.
private FWriter        writer
                          Объект записи в БД.
```

13.6.1. Класс MBroker

Будучи единственным классом в пакете `mediator` (посредник), класс `MBroker` (класс «посредник» пакета `mediator`) выполняет весьма немного работы. Он соединяет и разделяет три PCMEF-пакета (`control` — управление, `entity` — сущность и `foundation` — основание), чтобы ликвидировать неприятные зависимости. `MBroker` осведомлен об объектах пакета `entity`, в настоящее время находящихся в кэше памяти. Он отвечает за извлечение данных из результатов SQL-запросов (уровня `foundation`) и за поддержание основной логики создания компонентов пакета `entity`. `MBroker` управляет передачей данных строк таблиц (результатов запроса) в объекты пакета `entity` и наоборот.

Основные функции `MBroker` косвенно вытекают из элементов данных, которые он содержит, как показано в листинге 13.48.

Листинг 13.49 представляет методы, определенные в `MBroker`.

Листинг 13.49. Сводка методов класса MBroker

```
MBroker()
                          Конструктор посредника.
MBroker(FConnection    connection)
                          Создание нового брокера с конкретной связью.
private IAContact      createContact()
                          Создание нового объекта делового партнера.
private IAEmployee     createEmployee()
                          Создание нового объекта сотрудника.
private IAOutMessage   createOutMessage()
                          Создание нового объекта исходящего сообщения.
private                doRetrieveMessagesAndContacts(IAContact
java.util.Collection   contact, int numMsgs)
                          Выполнение извлечения исходящих сообщений из БД.
```

`private IAOutMessage`	**`doRetrieveUnsentMessageAndContact`**`(IAEmployee em, int msgID)` Выполнение извлечения сообщения с конкретным `msgID` (идентификатор сообщения) непосредственно из БД.
`private java.util.Collection`	**`doRetrieveUnsentMessagesAndContacts`**`(IAEmployee emp, int numMsgs)` Выполнение извлечения непосланных исходящих сообщений из БД.
`private void`	**`flagCache`**`()` Изменение флага, указывающего, является ли кэш пакета `entity` «измененным».
`IAContact`	**`getContact`**`(int contactOID)` Извне не используется, только для классов пакета `entity`.
`IAEmployee`	**`getEmployee`**`(int empOID)` Извне не используется, только для классов пакета `entity`.
`private int`	**`getOID`**`(java.lang.Object o)` Создание идентификатора объекта для заданного объекта.
`private IAOutMessage`	**`getOutMessageFromCache`**`(int msgID)` Удаление исходящего сообщения из кэша, если такое уже там есть.
`private boolean`	**`isDirty`**`()` Находится ли флаг кэша в состоянии «измененный»?
`private boolean`	**`isInCache`**`(java.lang.Integer msgID)` Находится ли исходящее сообщение уже в кэше?
`IAEmployee`	**`login`**`(java.lang.String username, java.lang.String passwd)` Получение регистрационного имени и возвращение деталей служащего.
`void`	**`logout`**`()` Отмена регистрации пользователя, очистка всех связей.
`private IAContact`	**`mapContact`**`(java.sql.ResultSet rs)` Размещение строки данных о деловом партнере в `EContact`.
`private IAEmployee`	**`mapEmployee`**`(java.sql.ResultSet rs)` Размещение строки данных о сотруднике в `EEmployee`.
`private IAOutMessage`	**`mapOutMessage`**`(java.sql.ResultSet rs)` Размещение строки данных об исходящем сообщении в `EOutMessage`.
`private java.util.Collection`	**`mapOutMessages`**`(java.sql.ResultSet rs, int numRetrieve)` Извлечение только `numRetrieve` сообщений из результирующего набора.
`private void`	**`retrieveContacts`**`(java.util.Collection msgs)` Извлечение деловых партнеров для данных сообщений.
`IAOutMessage`	**`retrieveUnsentMessage`**`{IAEmployee emp, int msgID)` Извлечение одного сообщения либо из кэша, либо непосредственно из БД.

int	**retrieveUnsentMessages**(IAEmployee emp, java.util.Collection msgs, int numMsgs) Извлечение непосланных исходящих сообщений из кэша или БД.
boolean	**updateMessage**(int msgID) Индикация, что сообщение было послано.

Конструирование объекта MBroker

Имеются два конструктора, определенных в `MBroker` (листинг 13.50). Конструктор, фактически используемый в итерации 1, показан в строках 48–50. Конструктор, который создает `MBroker` для конкретного объекта `FConnection` — класс «соединение» пакета `foundation` (строки 52–68), упоминается в примерах задач.

Связь для запроса регистрационного имени

После проверки в `CActioner` (класс «исполнитель» пакета `control`) методом `login()` — регистрационное имя (раздел 13.4.1) `MBroker` обращается к `FConnection` для организации соединения пользователя с БД (листинг 13.51, строки 78–81). Проверка регистрационного имени определяет, является ли пользователь служащим в таблице `Employee` (служащий). Это выполняется с помощью SQL-запроса (раздел 12.1.3) в строках 83–85. При успешном завершении проверки регистрационного имени `MBroker` создает новый объект `EEmployee` (класс «служащий» пакета `entity`) в кэше памяти, используя метод `mapEmployee()` — разместить информацию о сотруднике (строка 86), как сказано в следующем разделе.

Листинг 13.50. Конструкторы MBroker()

Конструкторы MBroker()

```
48:     public MBroker() throws ClassNotFoundException {
49:        this(new FConnection(DB_DRIVER, DB_URL));
50:     }
51:
52:     /** Создание нового брокера с конкретным соединением */
53:     public MBroker(FConnection connection) {
54:        try {
55:        dirty = true;
                    // вначале извлечение исходящих сообщений
56:           this.connection = connection;
57:           reader = connection.getReader();
58:           writer = connection.getWriter();
63:        }
64:        msgs = new HashMap();
65:        contacts = new HashMap();
66:        emps = new HashMap();
67:        remainder = 0;
68:     }
```

Листинг 13.51. Метод login() в MBroker

Метод login() в MBroker

```
76:   public IAEmployee login(String username, String passwd) {
77:        try {
78:        if (!connection.isConnected()) {
79:           // соединение не удалось
80:           if (!connection.connect(username, passwd))
81:              return null;
82:        }
83:        java.sql.ResultSet rs =
84:           reader.query(
85:             "select * from employee where login_name = '" +
                                              username + " ' " );
86:           IAEmployee emp = mapEmployee(rs);
87:           reader.closeResult(rs);
88:           return emp;
89:        } catch (Exception exc) {
90:        }
91:        return null;
92:     }
```

Создание кэша сотрудников

Хотя индекс таблицы `Employee` гарантирует, что регистрационные имена служащих уникальны (раздел 10.3.1), SQL-запрос в листинге 13.51 (строки 83–85) допускает возвращение множества записей (`ResultSet` — набор результатов). При вызове метод `mapEmployee()` (листинг 13.52) просматривает множество результатов и создает объект `EEmployee` (строка 413). После создания строка данных помещается в поля объекта `EEmployee` (строки 415–424). Поле `OID` (идентификатор объекта) объекта `EEmployee` задается в строке 426 и используется как индекс коллекции `emps Map` (карта служащих) (строка 427).

Листинг 13.52. Метод mapEmployee() в MBroker

Метод mapEmployee() в MBroker

```
409:    private IAEmployee mapEmployee(java.sql.ResultSet rs) {
410:       EEmployee emp = null;
411:       try {
412:          if (rs.next()) {
413:             emp = (EEmployee) createEmployee();
414:
415:            String empid = rs.getString("employee_id");
416:            String fname = rs.getString("first_name");
417:            String familyname = rs.getString("family_name")
418:            String email = rs.getString{"employee_email");
419:            String loginName = rs.getString("login_name");
```

```
420:             emp.setEmployeeID(empid);
421:             emp.setFirstName(fname);
422:             emp.setFamilyName(familyname);
423:             emp.setEmail(email);
424:             emp.setLoginName(loginName);
425:
426:             emp.setOID(getOID("EMP" + empid));
427:             emps.put(new Integer(emp.getOID()), emp);
428:          }
429:       } catch (Exception exc) {
430:       }
431:       return emp;
432:    }
```

Извлечение непосланных сообщений

Запрашивая `CActioner` (раздел 13.4.1), `MBroker` собирается извлечь исходящие сообщения для данного служащего (пользователя). `MBroker` может извлечь исходящие сообщения из кэша или, если кэш пуст или «изменен», может выполнить извлечение исходящих сообщений из БД. Первая задача обеспечивается методом `retrieveUnsentMessages()` — извлечь непосланные сообщения (листинг 13.53, строки 192–209). Вторая задача решается методом `doRetrieveUnsentMessagesAndContacts()` — извлечь непосланные сообщения и деловых партнеров (строки 211–247).

Листинг 13.53. Методы retrieveUnsentMessages() и doRetrieveUnsentMessagesAndContacts() в MBroker

Методы retrieveUnsentMessages() и doRetrieveUnsentMessagesAndContacts() в MBroker

```
192:    * @param emp — служащий, назначенный для этих сообщений
193:    * @param msgs — коллекция, заполняемая сообщениями
194:    * @param numMsgs — число извлекаемых сообщений
195:    * @return int — число еще не извлеченных сообщений
196:    */
197:    public int retrieveUnsentMessages(
198:       IAEmployee emp,
199:       Collection msgs,
200:       int numMsgs) {
201:       if (msgs.isEmpty()) {
202:          try {
203:           msgs.addAll(doRetrieveUnsentMessagesAndContacts
                                                (emp,numMsgs));
204:          } catch (Exception exc) {
205:            exc.printStackTrace();
206:          }
207:       }
208:       return remainder;
```

```
209:     }
210:
211:     /** Извлечение исходящих сообщений из БД. */
212:     private Collection doRetrieveUnsentMessagesAndContacts(
213:        IAEmployee emp,
214:        int numMsgs)
215:        throws Exception {
222:
223:        java.sql.ResultSet rs =
224:          reader.query(
225:          "Select * from outmessage where sender_emp_id = '"
226:                + emp.getEmployeeID()
227:                + "' and date_emailed is null");
228:
230:        Collection msgs = mapOutMessages(rs, maxRetrieve);
231:        reader.closeResult(rs);
232:        retrieveContacts(msgs);
233:
234:        dirty = false;
235:
236:        // вычисление числа оставшихся сообщений на сервере
237:        rs =
238:           reader.query(
239:              "Select count(*) from outmessage where
                                                sender_emp_id = '"
240:                + emp.getEmployeeID()
241:                + "' and date_emailed is null");
242:        rs.next();
243:        int count = rs.getInt(1);
244:        reader.closeResult(rs);
245:        remainder = count - msgs.size();
246:        return msgs;
247:     }
```

`MBroker` также обеспечивает извлечение единственного исходящего сообщения для данного служащего, используя метод `retrieveUnsentMessage()` — извлечь непосланное сообщение. Этот метод обычно вызывается, чтобы отобразить тело исходящего сообщения непосредственно перед отправкой его по электронной почте. Код этого метода, а также метода, вызываемого им и имеющего имя `doRetrieveUnsentMessageAndContact()`, подобен показанному выше и поэтому здесь не представлен.

Создание кэша исходящих сообщений

Поиск исходящих сообщений в БД завершается размещением объектов `EOutMessage` в кэше. Эта задача выполняется вызовом метода `mapOutMessages()` (разместить исходящие сообщения) в методе `doRetrieveUnsentMessagesAndContacts()` (листинг 13.53, строка 230).

Листинг 13.54. Метод mapOutMessages() в MBroker

Метод mapOutMessages() в MBroker

```
493:    private Collection mapOutMessages(
494:       java.sql.ResultSet rs,
495:       int numRetrieve) {
496:
497:       Collection c = new ArrayList(numRetrieve);
498:       for (int i = 0; i < numRetrieve; i++) {
499:          IAOutMessage msg = mapOutMessage(rs);
500:          if (msg == null)
501:             return c;
502:          c.add(msg);
503:       }
504:       return c;
505:    }
```

Метод `mapOutMessages()` (разместить исходящие сообщения) приведен в листинге 13.54. Метод же `mapOutMessage()` (поместить исходящее сообщение), вызываемый в строке 499, подобен методу `mapEmployee()` в листинге 13.52 и не показан в целях экономии места.

Создание кэша деловых партнеров

Извлечение исходящих сообщений из БД также завершается размещением объектов `EContacts` (класс «деловые партнеры» пакета `entity`) в кэше. Эта задача выполняется вызовом метода `retrieveContacts()` (извлечь информацию о деловых партнерах) из метода `doRetrieveUnsentMessagesAndContacts()` (листинг 13.53, строка 232).

Метод `retrieveContacts()` приведен в листинге 13.55. Метод `mapContact()` (поместить информацию о деловом партнере), вызываемый в строке 268, подобен методу `mapEmployee()` в листинге 13.52 и не показан в целях экономии места.

Листинг 13.55. Метод retrieveContacts() в MBroker

Метод retrieveContacts() в MBroker

```
250:    private void retrieveContacts(Collection msgs) {
251:       Iterator it = msgs.iterator();
252:       EOutMessage msg;
253:       while (it.hasNext()) {
254:          msg = (EOutMessage) it.next();
255:          IAContact contact = null;
256:          Object cntObj = contacts.get(new
                      Integer(msg.getContactOID()));
257:          try {
258:             contact = (IAContact) cntObj;
259:          } catch (ClassCastException exc) {
```

```
260:          }
261:          if (contact == null) {
262:             try {
263:                java.sql.ResultSet rs =
264:                   reader.query(
265:                      "Select * from contact where
                                                  contact_id = '"
266:                         + cntObj.toString()
267:                         + "'");
268:                contact = mapContact(rs);
269:                reader.closeResult(rs);
270:             } catch (Exception exc) {
271:                exc.prihtStackTrace();
272:             }
273:          }
274:       }
275:    }
```

Обновление исходящих сообщений после передачи по электронной почте и восстановление кэша

При успешной передаче по электронной почте исходящего сообщения метод `sendMessage()` (послать сообщение) класса `CActioner` (раздел 13.4.1) обращается к методу `updateMessage()` (скорректировать сообщение) класса `MBroker`. Эта операция имеет двойную цель: скорректировать таблицу `OutMessage` (исходящее сообщение) в БД и восстановить кэш пакета `entity` программы, удаляя переданный по электронной почте объект `EOutMessage` из своих объектов `EEmployee` и `EContact`.

Листинг 13.56 представляет метод `updateMessage()`. В строке 292 метод получает из кэша объект `EOutMessage` (только что переданный по электронной почте). В строке 299 он устанавливает в поле `sentDate` (листинг 13.42) этого объекта текущую дату. Затем он вызывает метод `unmapOutMessage()` — не размещать исходящее сообщение (строка 301), чтобы скорректировать БД (листинг 13.57). Когда это будет сделано, вызов метода `flagCache()` (задание флага кэша) в строке 305 помечает кэш как «измененный». Операторы в строках 308–315 удаляют переданный по электронной почте объект `EOutMessage` из своих объектов `EEmployee` и `EContact` (на самом деле это в итерации 1 не реализовано).

Листинг 13.56. Метод updateMessage() в MBroker

Метод updateMessage() в MBroker

```
287:    public boolean updateMessage(int msgID) {
292:       EOutMessage msg = (EOutMessage)
                  getOutMessageFromCache(msgID)
299:       msg.setSentDate(new
             java.sql.Date(System.currentTimeMillis()));
```

```
300:       try {
301:     java.sql.PreparedStatement st = unmapOutMessage(msg);
302:          if (st.executeUpdate() != 1)
303:             return false;
304:
305:          flagCache();
306:
308:          msgs.remove(new Integer(msg.getOID()));
309:          EContact contact = (EContact) msg.getContact();
310:          contact.removeSentOutMessage(msg);
311:          EEmployee emp = (EEmployee)
                                  msg.getCreatorEmployee();
312:          emp.removeSentOutMessage(msg);
313:          emp = (EEmployee) msg.getSenderEmployee();
314:          emp.removeSentOutMessage(msg);
315:          return true;
319:       }
321:       return false;
322:    }
```

Листинг 13.57. Метод unmapOutMessage() в MBroker

Метод unmapOutMessage()в MBroker

```
508:    private java.sql.PreparedStatement unmapOutMessage
                                            (EOutMessage msg)
509:       throws Exception {
510:       java.sql.PreparedStatement st =
511:          writer.update(
512:             "Update OutMessage set date_emailed = ? where
                                              message_id = ?");
513:       st.setDate(1, msg.getSentDate());
514:       st.setInt(2, msg.getMessageID());
515:       return st;
516:    }
```

13.7. Пакет Foundation

Пакет `foundation` (основание) состоит их трех классов:

- `FConnection` (класс «соединение» пакета `foundation`);
- `FReader` (класс «чтение» пакета `foundation`);
- `FWriter` (класс «запись» пакета `foundation`).

13.7.1. Класс FConnection

Класс `FConnection` (класс «соединение» пакета `foundation`) выполняет типичные задачи по установлению соединения с БД, как было рассмотрено в

разделе 12.1.3. Он использует возможности классов FReader (класс «чтение» пакета foundation) и FWriter (класс «запись» пакета foundation), чтобы соответственно выполнить поиск в БД или модифицировать ее.

Листинг 13.58 показывает элементы данных в FConnection, включая ассоциации с FReader и FWriter.

Листинг 13.58. Сводка полей класса FConnection

```
private java.sql.Connection  conn
private java.lang.String     dbDriver
private java.lang.String     dbUrl
private FReader              reader
private FWriter              writer
```

Листинг 13.59 представляет сводку методов класса FConnection, включая ассоциации с FReader и FWriter.

Листинг 13.59. Сводка методов класса FConnection

```
FConnection (java.lang.String driver, java.lang.String dbUrl)
```
Конструирование нового объекта FConnection.

```
void     close()
```
Закрыть соединение.

```
boolean  connect(java.lang.String username,
                 java.lang.String  passwd)
```
Получить соединение с БД, используя данные username (пользовательское имя) и passwd (пароль). Если связь успешно установлена, но существует старая связь, старая связь будет закрыта и заменена новой связью.

```
FReader  getReader()
```
Установить реализацию чтения по умолчанию, поддерживаемую этим соединением.

```
FWriter  getWriter()
```
Установить реализацию записи по умолчанию, поддерживаемую этим соединением. В идеале этот класс мог быть реализован как AbstractFactory (абстрактная фабрика) или AbstractMethod (абстрактный метод), но мы в данном случае выбираем простой вариант.

```
boolean  isConnected()
```
Установлено ли соединение с БД?

Конструирование объекта FConnection

Конструктор нового объекта FConnection показан в листинге 13.60. Он инициализирует свои элементы данных dbDriver (драйвер БД) и dbUrl (унифицированный указатель БД в Интернете). Класс FConnection совместно использует свой объект java.sql.Connection со своими же объектами FReader и FWriter.

Листинг 13.60. Конструктор FConnection()

```
Конструктор FConnection()

33:     public FConnection( String driver, String dbUrl )
                              throws ClassNotFoundException {
34:        this.dbDriver = driver;
35:        Class.forName( driver );
36:        this.dbUrl = dbUrl;
37:     }
```

Получение соединения с БД

Метод `connect()` (соединение) в листинге 13.61 устанавливает соединение с БД для заданных пользовательского имени и пароля. Если связь успешно установлена, но существует старая связь, эта старая связь будет закрыта и заменена новой связью. Метод возвращает `true` (истина), если установлена только одна связь (или старая, или новая).

Листинг 13.61. Метод connect() в FConnection

```
Метод connect() в FConnection

79:  public boolean connect( String username, String passwd ) {
80:    if (conn != null) {// мы имеем старое соединение,
                                мы можем  закрыть его
81:      if ( ( username != null && username.length() != 0 ) &&
             ( passwd != null && passwd.length() != 0 ) ) {
82:         Connection con = null;
83:         try {
84:           con = DriverManager.getConnection( dbUrl,
                                 username, passwd );
85:         }catch ( Exception exc2 ) {
86:            exc2.printStackTrace(); // отладка
87:         }
88:         if (con != null) {// переключение соединения
89:            try {
90:                conn.close();
91:            }catch ( Exception exc ) {}
92:            conn = con;
93:         }
94:      }
95:    }
96:    else {// соединения не было, создание нового
97:       try {
98:          conn = DriverManager.getConnection( dbUrl,
                                          username, passwd );
99:       }catch ( Exception exc ) {
```

```
100:            exc.printStackTrace(); // отладка
101:         }
102:      }
104:      getReader().setConnection( conn );
105:      getWriter().setConnection( conn );
106:      try {
107:         return conn != null && !conn.isClosed();
108:      }
109:      catch ( Exception exc ) {
110:         return false;
111:      }
112:   }
```

Методы `getReader()` и `getWriter()` в строках 104–105 используются для создания новых объектов `FReader` и `FWriter` и установления соответствующих связей ассоциации от объекта `FConnection`. Сами методы здесь не показаны.

13.7.2. Класс FReader

Класс `FReader` (класс «чтение» пакета `foundation`) отвечает за выполнение строк SQL-запросов, возвращение множества результатов клиенту и зачистку множества результатов после того, как они были обработаны. Листинг 13.62 содержит сводку методов `FReader`. Методы далее не обсуждаются. Принципы выполнения SQL-операторов подробно рассмотрены в разделе 12.1.3.

Листинг 13.62. Сводка методов класса FReader

FReader(java.sql.Connection conn)	Создание объекта `FReader` для представленного соединения.
void	**closeResult**(java.sql.ResultSet rs) `MBroker` (класс «посредник» пакета `mediator`) вызывает этот метод после передачи множества результатов.
java.sql.ResultSet	**query**(java.lang.String sql) Запрос к БД на основе строки SQL.
void	**setConnection**(java.sql.Connection conn) Изменение соединения для этого объекта.

13.7.3. Класс FWriter

Класс `FWriter` (класс «запись» пакета `foundation`) ответствен за выполнение строки корректировки SQL, переданной ему объектом `MBroker` — класс «посредник» пакета `mediator` (листинг 13.57), и зачистку множества результатов после того, как они были обработаны. Листинг 13.63 — сводка методов `FWriter`. Методы далее не обсуждаются. Принципы выполнения SQL-операторов подробно рассмотрены в разделе 12.1.3.

Листинг 13.63. Сводка методов объекта FWriter

```
FWriter(java.sql.Connection conn)
                                Создание объекта FWriter, который записывает
                                все изменения и новые данные с помощью пере-
                                данного соединения.
void                            closeStatement(java.sql.
                                                PreparedStatement st)
void                            setConnection(java.sql.Connection conn)
                                Изменение соединения для этой записи.
java.sql.PreparedStatement      update(java.lang.String sql)
```

Резюме

1. Код Java для итерации 1 строго соответствует структурному шаблону PCMEF.
2. Чтобы подчеркнуть ясность и выдвинуть на первый план простоту PCMEF-шаблона, итерация 1 ограничивает число классов в каждом пакете.
3. Пакет `acquaintance` (знакомство) состоит из четырех интерфейсов: `IAConstants` (интерфейс «константы» пакета `acquaintance`), `IAContact` (интерфейс «деловой партнер» пакета `acquaintance`), `IAEmployee` (интерфейс «служащий» пакета `acquaintance`) и `IAOutMessage` (интерфейс «исходящее сообщение» пакета `acquaintance`).
4. Пакет `presentation` (представление) содержит два класса: `PMain` (класс «основной» пакета `presentation`) и `PConsole` (класс «консоль» пакета `presentation`).
5. Пакет `control` (управление) содержит только один класс: `CActioner` (класс «исполнитель» пакета `control`). Этот класс инкапсулирует все операции, известные в итерации 1 учебного примера EM.
6. JavaMail™ API — обычно известная как MAPI (Mail API) — является библиотекой для чтения, создания и передачи сообщений по электронной почте.
7. Пакет `entity` (сущность) состоит из одного интерфейса и трех классов: `IEDataSupplier` (интерфейс «источник данных» пакета `entity`), `EContact` (класс «деловой партнер» пакета `entity`), `EEmployee` (класс «служащий» пакета `entity`) и `EOutMessage` (класс «исходящее сообщение» пакета `entity`).
8. Пакет `mediator` (посредник) состоит из одного класса: `MBroker` (класс «брокер» пакета `mediator`). `MBroker` связывает и устраняет у пакетов `control`, `entity` и `foundation` неприятные зависимости.
9. Пакет `foundation` (основание) состоит из трех классов: `FConnection` (класс «соединение» пакета `foundation`), `FReader` (класс «чтение» пакета `foundation`) и `FWriter` (класс «запись» пакета `foundation`).

Ключевые термины

Итерация 1. Вопросы и упражнения

Ниже приведен небольшой набор вопросов и упражнений относительно кода итерации 1 подсистемы EM. Большее количество вопросов и упражнений можно получить на Web-сайте книги.

1. Итерация 1 использует библиотеку коллекций Java, чтобы реализовать ассоциации между классами-сущностями. Что является этими ассоциациями? Какой конкретно контейнер Java используется? Какой класс и какой метод инициализирует этот контейнер? Какой класс и какой метод заполняет (загружает) этот контейнер? Как инициализируются связи ассоциации с этим контейнером?
2. Какова цель интерфейса `IAConstants` (интерфейс «константы» пакета `acquaintance`)? Может ли та же цель быть достигнута по-другому? Как?
3. Листинг 13.13 содержит два конструктора для класса `PConsole` (класс «консоль» пакета `presentation`). Второй конструктор `PConsole(CActioner actioner)` не используется в итерации 1. Как бы он мог использоваться? Аналогично листинг 13.50 содержит два конструктора для `MBroker` (класс «брокер» пакета `mediator`), и второй конструктор не используется в итерации 1. Как бы он мог использоваться? Являются ли эти два конструктора в `PConsole` и `MBroker` связанными каким-либо способом?
4. Объясните код после первого метода `SendMessage()` (послать сообщение) в `CActioner` — класс «исполнитель» пакета `control` (листинг 13.28, строки 97–99).
5. Метод `getUnsentMessages()` — получение непосланных сообщений (листинги 13.35 и 13.40) не используется в итерации 1. Каковы последствия этого?
6. Какова причина в итерации 1 заменить старое соединение с БД новым, как это реализовано в методе `connect()` (соединение) класса `FConnection` — класс «соединение» пакета `entity` (листинг 13.61)?
7. Файл ресурса мог бы заменить интерфейс `IAConstants` в итерации 1 (раздел 13.2.1). Покажите содержимое такого файла. Объясните, как приложение будет использовать файл.
8. Рассмотрите метод `displayLoginError()` (отобразить сообщение об ошибке ввода регистрационного имени) в листинге 13.15. Что могло бы быть очевидной альтернативой такой реализации?
9. Объясните, какие следует сделать изменения кода, если было бы принято решение удалить интерфейс `IEDataSupplier` (интерфейс «источник данных» пакета `entity`) в итерации 1. Как был бы изменен PCMEF-шаблон?

Часть 3

Рефакторинг программного обеспечения и разработка пользовательского интерфейса

Итерация 1 учебного примера сформировала программный продукт, который был упрощен по ряду причин. Главная цель состояла в том, чтобы поставить версию проекта, демонстрируя ясное структурное проектирование и возможность его расширения. Вторая цель состояла в том, чтобы использовать при решении только наиболее общую практику разработки ПО и объяснить эту практику таким образом, чтобы упростить образовательную миссию книги.

По-видимому, наиболее очевидными недостатками продукта итерации 1 являются отсутствие *графического пользовательского интерфейса* (Graphical User Interface — GUI) и слабое использование в проекте преимуществ ряда наиболее мощных аспектов объектной технологии. Итерация 2, рассматриваемая в этой части книги, предназначена исправить эти недостатки.

Технологии и методы разработки ПО, раскрытые в итерации 2 учебного примера EM, включают рефакторинг, кэширование в памяти, наследование реализации, компоновку, GUI-проект, вариант GUI-проекта, доступный для Web-структур, видимость, полиморфизм, обработку исключений, параллелизм, потоки и т. д. Структурный шаблон итерации 1 сохранен и усилен, чтобы обеспечить, как и ранее, понятность, удобство сопровождения и масштабируемость продукта. *Быстрая разработка ПО* остается центральным звеном в учебном примере.

Главные учебные цели части 3 и итерации 2 учебного примера заключаются в том, чтобы получить следующие знания:

- рефакторинг кода и то, как качественная исходная структура проекта облегчает это;
- отношение между рефакторингом и паттернами проекта;
- паттерны рефакторинга для среднего уровня (middle tier) системы, которые облегчают будущее внедрение уровня предметной области (domain layer) ПО на сервере приложения;
- принципы проектирования пользовательского интерфейса;
- практика проектирования пользовательского интерфейса, основанная на стандартных компонентах и стандартной модели событий;
- технологии, поддерживающие проектирование пользовательского интерфейса, доступного для Web-структур;
- обеспечение технологий для уровня Web-серверов и управление транзакциями в системах Интернета без сохранения адресов;
- паттерны проектирования для уровня представления (presentation layer) приложения в отношении к пользовательским интерфейсам с использованием рабочего стола и Web-структур.

Глава 14

Требования к итерации 2 и объектная модель

Центром внимания итерации 2 учебного примера управления электронной почтой (Email Management — EM) является создание внешнего GUI-интерфейса к приложению и создание доступных для Web-технологии версий системы. Согласно PCMEF-шаблону большинство изменений в этой итерации находится на уровне представления (presentation), который инкапсулирует взаимодействие с пользователем. Новые функциональные возможности итерации 2 добавляются поверх функциональных возможностей итерации 1.

Типичная практика **быстрой разработки программного обеспечения** рекомендует повторение итераций каждые две недели [66]. В конце каждых двух недель работы работающий программный продукт для текущей итерации демонстрируется клиентам и подвергается приемочным испытаниям. Две недели работы коллектива разработчиков дают результат существенного размера.

Размеры EM-итераций соответствуют принципам быстрой разработки ПО. В то время как невозможно в учебнике исследовать каждую деталь прикладного кода, можно дать ему объяснения, концентрируясь на новых аспектах каждой итерации. Новые аспекты итерации 2 — рефакторинг уровня предметной области приложения и замена уровня представления как для GUI с использованием рабочего стола, так и для доступа к системе на основе Web-технологии.

14.1. Модель сценариев использования

Добавление к приложению внешнего интерфейса GUI, допускающего использование Web-технологии, обращается к свойству применимости в FURPS+, задокументированному в дополнительных спецификациях проекта (раздел 8.4). Итерация 2 добавляет дополнительные особенности функциональных возможностей. Они добавлены в модель сценариев использования и в связанный со сценариями использования документ.

Рис. 14.1 представляет модель сценариев использования для итерации 2. Граница проекта и акторы, которые определяют эту границу, не изменились по сравнению с итерацией 1. Поэтому акторы в модели не показаны. Диаграмма представляет сценарии использования и основные отношения между ними. Новые функциональные возможности итерации 2 охвачены прямоугольником. Некоторые новые сценарии использования являются просто специализациями (сценариями использования подтипов) абстрактного сценария

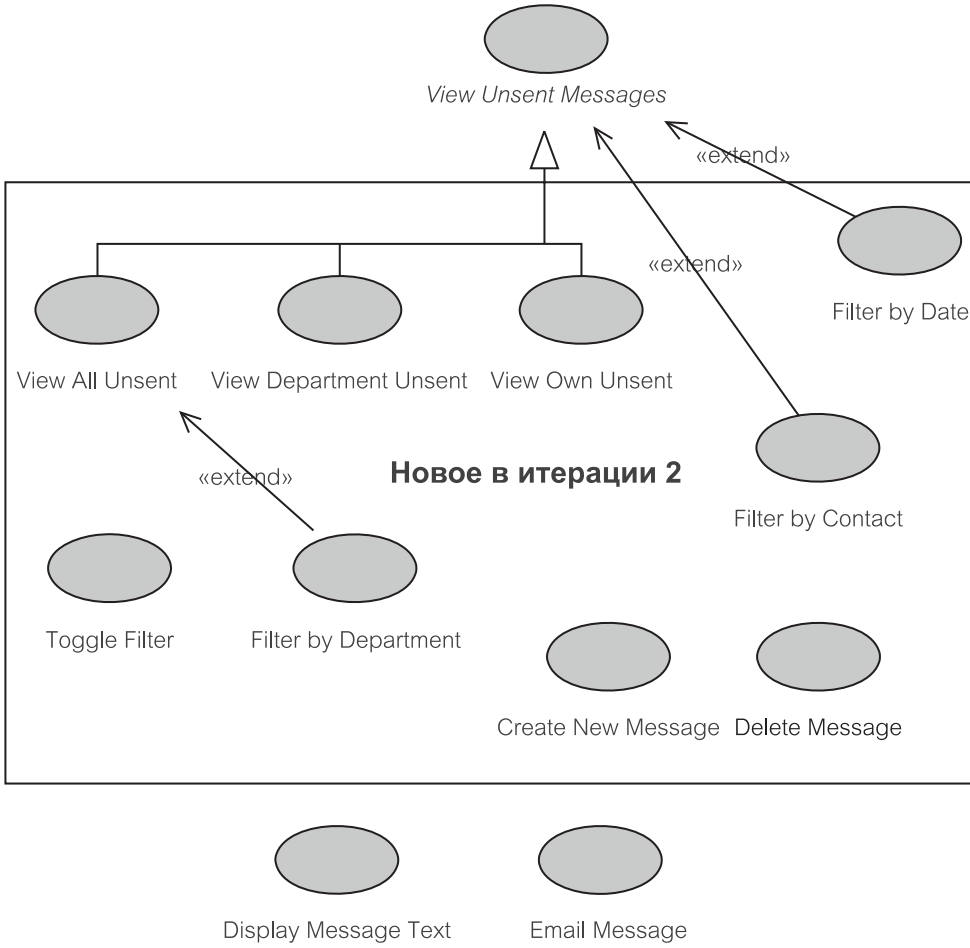


Рис. 14.1. Диаграмма сценариев использования для итерации 2

использования `View Unsent Messages` (просмотр непосланных сообщений). Имеются четыре новых сценария использования, связанных с фильтрацией исходящих сообщений, которые можно в настоящее время отобразить. Наконец, имеется сценарий использования `Create New Message` (создать новое сообщение) и сценарий использования `Delete Message` (удалить сообщение).

В итерации 1 сценарий использования `View Unsent Messages` (просмотр непосланных сообщений) отвечал за отображение всех пока еще не посланных исходящих сообщений и находящихся в БД (то есть со значением параметра `date_emailed` — дата отправки исходного сообщения, равным `null`, в таблице `OutMessage` — исходящее сообщение). В итерации 2 этот сценарий использования детализирован тремя подтипами. `View All Unsent` (представление всех непосланных сообщений) обеспечивает функциональные возможности сценария использования итерации 1 `View Unsent Messages`.

Сценарий использования `View Department Unsent` (представление непосланных отделом сообщений) отображает все исходящие сообщения, намеченные для передачи по электронной почте любым служащим отдела, в котором работает текущий пользователь (то есть, зарегистрированный в EM служащий). Это подразумевает, что концептуальные классы (заканчивающиеся схемой БД) должны иметь информацию об исходящих сообщениях всех служащих соответствующих отделов и «намеченных отделам» исходящих сообщениях.

Сценарий использования `View Own Unsent` (представление собственных непосланных сообщений) отображает исходящие сообщения, намеченные для передачи по электронной почте текущим служащим (то есть, зарегистрированным в EM служащим). Это подразумевает, что концептуальные классы должны иметь информацию о «намеченных служащим» исходящих сообщениях.

Так как сценарий использования `View Unsent Messages` полностью определяется тремя сценариями использования подтипов, непосредственно сам он становится **абстрактным сценарием использования**. Он также является *основным сценарием использования* для двух других сценариев входной и выходной фильтрации видимых в настоящее время исходящих сообщений. Эти два сценария использования — `Filter by Date` (фильтр по дате) и `Filter by Contact` (фильтр по деловому партнеру). Третий основной фильтр, `Filter by Department` (фильтр по отделу), лишь расширяет сценарий использования `View All Unsent`. Этот фильтр был добавлен к учебному примеру в образовательных целях, его практическая ценность сомнительна, потому что сценарий использования `View Department Unsent` дает практически тот же результат.

Сценарий использования `Toggle Filter` (выключатель фильтра) включает или выключает фильтр для отображенного в настоящее время списка непосланных исходящих сообщений. Согласно сценарию использования `View Unsent Messages`, текущий список может представлять все сообщения, сообщения отдела или собственные сообщения. Первоначально фильтр выключен. Фильтр может быть установлен по намеченной дате (то есть дате, намеченной для передачи по электронной почте) и/или по деловому партнеру (то есть, по предназначению электронного сообщения) и/или по отделу (то есть, по отделу, ответственному за передачу электронных сообщений). Фильтр по дате определяется не конкретной датой, а датой по сравнению с сегодняшней датой. Это означает, что фильтр по дате может находить прошлые, сегодняшние и будущие исходящие сообщения.

Сценарии использования `Create New Message` и `Delete Message` расширяют функциональные возможности итерации 1, позволяя создавать и удалять исходящие сообщения в БД. Предположение, что все исходящие сообщения в итерации 1 созданы отдельной системой «формирования» (и переданы EM-приложению), в итерации 2 не используется. Теперь можно управлять содержанием БД непосредственно из EM-приложения.

Два оставшихся сценария использования — `Display Message Text` (отображение текста сообщения) и `Email Message` (сообщение по электронной почте). Эти сценарии использования имеют те же самые функциональ-

ные возможности, что и в итерации 1, но представление этих функциональных возможностей использует рабочий стол и/или допустимые Web-технологии GUI.

14.2. Документ сценариев использования

Объем документа сценариев использования для итерации 2 представляет собой непосредственное объединение требований старой итерации 1 и требований новой итерации 2. Исходные сценарии использования, которые включают обе итерации, представлены на рис. 8.1 — `Manage Email` (управление электронной почтой). Как требует итеративная и пошаговая разработка, итерация 2 обеспечивает новый шаг поверх итерации 1.

14.2.1. Краткое описание, предусловия и постусловия

Краткое описание

Итерация 2 сценария использования `Manage Email` (управление электронной почтой) позволяет служащему создавать, удалять, отображать и передавать по электронной почте сообщения деловым партнерам. Диаграмма сценариев использования имеет дело только с отсылаемыми сообщениями. Поэтому ради обеспечения точности такие сообщения называются исходящими.

Сценарий использования может показывать и передавать по электронной почте исходящие сообщения, предварительно размещенные в БД другими процессами. Он также позволяет создавать новые исходящие сообщения, отображать их или передавать по электронной почте, если потребуется. Размещенные сообщения могут быть также удалены, когда в них больше нет потребности. Отображение исходящих сообщений может быть ограничено рядом условий отображения (критериев поиска). Текущее отображение может быть далее отфильтровано рядом условий фильтрации.

Предусловия (предварительные условия)

1. Пользователь (актор) — это служащий, который работает в отделе по связи с клиентами, или иным образом авторизованный администратором системы для доступа к EM-приложению.
2. БД EM содержит исходящие сообщения, которые должны быть переданы по электронной почте деловым партнерам. Создание новых исходящих сообщений не является никакой заменой для исходящих сообщений, размещенных в БД другим процессом AEM-системы (глава 6).
3. Служащий связан с сервером электронной почты и является авторизованным пользователем БД.

Постусловия (заключительные условия)

1. Программа корректирует БД EM, чтобы отразить любую успешную передачу по электронной почте исходящих сообщений.
2. БД EM остается в неповрежденном состоянии, если произойдет любое исключение или ошибка.

3. Для служащего, покидающего приложение, все соединения с БД закрываются; закрываются также все окна приложения на рабочем столе или допускающие использование Web-технологий.

14.2.2. Основной поток

Следуя подходу, принятому в итерации 1, описание потока событий включает прототипы GUI-окон. Прототипы позволяют «изобразить ситуацию» и облегчить понимание текстовых спецификаций.

Основной поток

Сценарий использования начинается, когда служащий желает создать, просмотреть и/или переслать по электронной почте исходящее сообщение деловым партнерам.

Система отображает информационное сообщение и запрашивает служащего, чтобы он ввел пользовательское имя и пароль (рис. 14.2).

1. Система пытается соединить служащего с БД EM.
2. После успешного соединения приложение отображает главное окно приложения, через которое пользователь может взаимодействовать с системой (рис. 14.3).
3. Пользователь может выполнить следующие задачи:
   а) представление непосланных деловым партнерам исходящих сообщений (см. ниже «*S1 — View Unsent Messages*» — просмотр непосланных сообщений);
   б) фильтрация отображения исходящих сообщений по различным критериям (см. ниже «*S2 — Filter Messages*» — фильтрация сообщений);
   в) отображение текста выбранного исходящего сообщения (см. ниже «*S3 — Display Message Text*» — отображение текста сообщения);
   г) передача по электронной почте выбранного исходящего сообщения (см. ниже «*S4 — Email Message*» — передача сообщения по электронной почте);
   д) создание нового исходящего сообщения (см. ниже «*S5 — Create New Message*» — создание нового сообщения);
   е) удаление выбранного исходящего сообщения (см. ниже «*S6 — Delete Message*» — удаление сообщения);
   ж) завершение работы с приложением, после чего сценарий использования заканчивается.

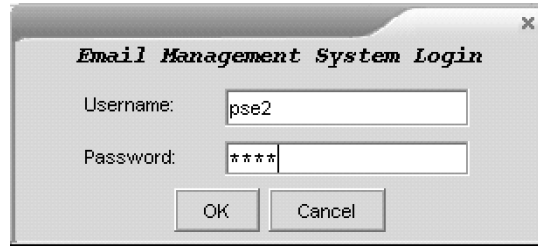


Рис. 14.2. Прототип окна для ввода регистрационного имени

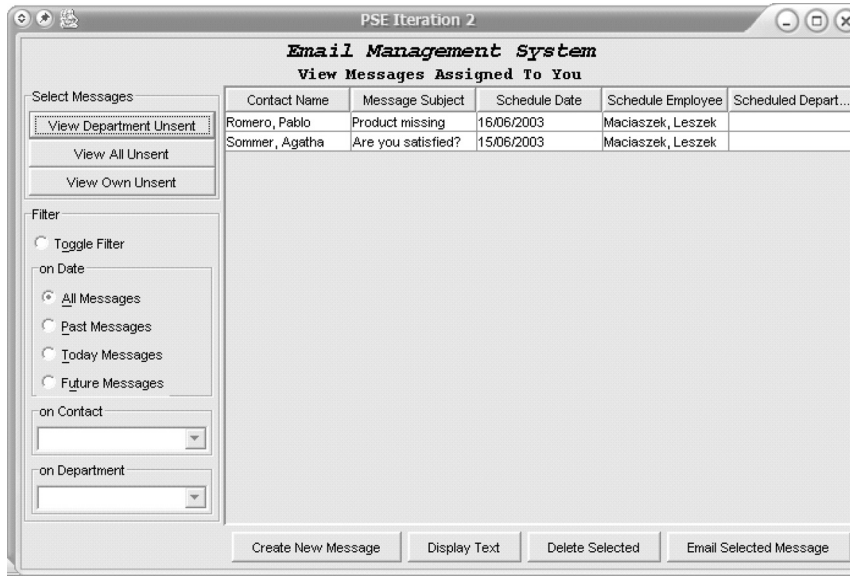


Рис. 14.3. Прототип основного окна приложения

14.2.3. Подпотоки

S1 — View Unsent Messages (просмотр непосланных сообщений)

Этот подпоток состоит из трех опций, доступных через группу из трех переключающихся кнопок или через подобные устройства (рис. 14.3):

- S1.1 — View All Unsent (просмотр всех непосланных сообщений);
- S1.2 — View Department Unsent (просмотр непосланных отделом сообщений);
- S1.3 — View Own Unsent (просмотр собственных непосланных сообщений).

S1.1 — View All Unsent (просмотр всех непосланных сообщений)

Когда пользователь выбирает опцию `View All Unsent`, приложение представляет полный список непосланных исходящих сообщений. В окне имеется четкий признак того, что отображаются все непосланные сообщения. Признак реализуется посредством нажатия переключающей кнопки `View All Unsent` и с помощью четкой надписи выше списка. Надпись содержит текст: `All unsent outmessages` (все непосланные исходящие сообщения).

- Список содержит именованные столбцы: `Contact Name` (имя делового партнера), `Message Subject` (тема сообщения), `Scheduled Date` (намеченная дата отправки), `Scheduled Employee` (намеченный служащий) и `Scheduled Department` (намеченный отдел):
  - `Contact Name` отображает фамилию и имя делового партнера;
  - `Scheduled Employee` отображает фамилию и имя служащего или ничего (если намеченный служащий неизвестен);

- `Scheduled Department` отображает название отдела или ничто (если намеченный отдел неизвестен).
  - Список может быть отсортирован, по крайней мере, по `Contact Name` и `Scheduled Date`.
  - Список может прокручиваться, чтобы учесть ситуации, когда число извлеченных строк превышает размер окна.
  - Исходящие сообщения в списке можно выбрать мышью. Должны поддерживаться обычные функциональные возможности: `Ctrl-Click` (нажатие левой клавиши мыши при нажатой клавише `Ctrl` клавиатуры), чтобы выбрать несколько несмежных исходящих сообщений, и `Shift-Click` (нажатие левой клавиши мыши при нажатой клавише `Shift` клавиатуры), чтобы выбрать диапазон смежных исходящих сообщений, хотя эти функциональные возможности не нужны для работы итерации 2.

S1.2 — View Department Unsent (просмотр непосланных отделом сообщений)

Когда пользователь выбирает `View Department Unsent`, приложение представляет список исходящих сообщений, намеченных для передачи по электронной почте служащими отдела, в котором работает текущий служащий (то есть, текущий пользователь). В окне имеется четкий признак того, что отображаются только исходящие сообщения, намеченные для конкретного отдела. Признак реализуется посредством нажатия переключающей кнопки `View Department Unsent` и с помощью четкой надписи выше списка. Надпись содержит текст: `Outmessages assigned to your department` (исходящие сообщения, назначенные для вашего отдела).

S1.3 — View Own Unsent (просмотр собственных непосланных сообщений)

Когда пользователь выбирает `View Own Unsent`, приложение представляет список исходящих сообщений, намеченных для отправки по электронной почте текущим служащим (то есть, текущим пользователем). В окне имеется четкий признак того, что отображаются только исходящие сообщения, намеченные для зарегистрированного служащего. Признак реализуется посредством нажатия переключающей кнопки `View Own Unsent` и с помощью четкой надписи выше списка. Надпись содержит текст: `Outmessages assigned to you` (исходящие сообщения, назначенные для вас).

S2 — Filter Messages (сообщения фильтра)

Этот подпоток состоит из четырех опций:

- S2.1 — Toggle Filter (выключатель фильтра);
- S2.2 — Filter by Date (фильтр по дате);
- S2.3 — Filter by Contact (фильтр по деловому партнеру);
- S2.4 — Filter by Department (фильтр по отделу).

S2.1 — Toggle Filter (выключатель фильтра)

Панель фильтров обеспечивает набор элементов управления для ограничения данных, выделенных для показа после того, как они были извлечены из БД подпотоком `View Unsent Messages` (просмотр непосланных сообщений). Панель не активна, пока не отображен список исходящих сообщений с помощью `View Unsent Messages` и не нажата кнопка `Toggle Filter`.

По умолчанию опция `Toggle Filter` выключена. Пользователь может включить ее, чтобы использовать другие опции фильтра: `Filter by Date`, `Filter by Contact` и/или `Filter by Department`. Опцию `Toggle Filter` (рис. 14.3) может реализовать копка-переключатель или панель выключателей.

Фильтр изображается активным, когда список отображенных исходящих сообщений ограничен условиями, заданными опциями `Filter by Date`, `Filter by Contact` и/или `Filter by Department`. Когда фильтр установлен, использование `Toggle Filter` может выключить его. Это приводит к повторному отображению списка исходящих сообщений, как требуется подпотоком `View Unsent Messages`, до того, как будет применен какой-либо фильтр.

S2.2 — Filter by Date (фильтр по дате)

Панель `Filter by Date` обеспечивает набор переключающих кнопок, чтобы отфильтровать исходящие сообщения с намеченной датой (то есть, намеченные для передачи по электронной почте) в прошлом, будущем, сегодняшней или все (с любой датой). В любое время может быть активна только одна из этих кнопок. По умолчанию, когда активизируется панель, активной становится кнопка `All` (все).

S2.3 — Filter by Contact (фильтр по деловому партнеру)

Панель фильтров реализуется комбинированной строкой ввода (полем со списком), которая позволяет выбрать делового партнера из выпадающего списка имен деловых партнеров или которая может быть оставлена незаполненной (рис. 14.3). Фильтр автоматически реализуется, когда отбирается деловой партнер. Это воздействует на список исходящих сообщений, отображаемых, чтобы показать лишь исходящие сообщения, предназначенные для выбранного делового партнера, исключая все исходящие сообщения, отфильтрованные текущим заданным фильтром даты.

S2.4 — Filter by Department (фильтр по отделу)

Здесь также используется комбинированная строка ввода, чтобы позволить пользователю отфильтровать список исходящих сообщений отдела, который выбирается из выпадающего списка названий отделов (рис. 14.3). Выводится только список исходящих сообщений, предназначенных конкретному отделу (который выбран с помощью комбинированной строки ввода). Пустое значение комбинированной строки ввода означает, что фильтрация по отделу не выполняется. Фильтрация выполняется автоматически, если отобрано название отдела. Фильтр изменяет список отображаемых исходящих сообщений, чтобы показать только те сообщения, которые назначены конкретному отделу, или, если быть точными, исходящие сообщения, которые назначены служащим выбранного отдела. Список исключает любые исходящие сообщения, отфильтрованные по заданной дате и по деловому партнеру.

S3 — Display Message Text (отображение текста сообщения)

Исходящее сообщение, выбранное (подсвеченное) в окне списка исходящих сообщений, может быть показано полностью в отдельном диалоговом окне (рис. 14.4). Это окно представляет полный текст исходящего сообщения

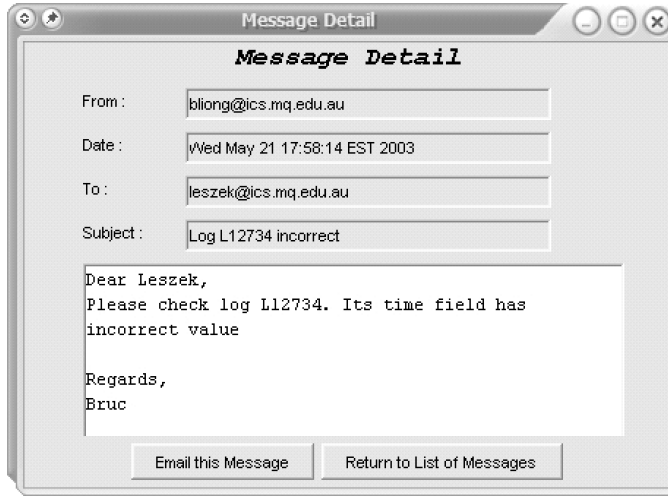


Рис. 14.4. Диалоговое окно, показывающее текст текущего выделенного сообщения

вместе с другой описательной информацией. Панель с текстом можно прокручивать. Окно содержит две кнопки:

- `Email this Message` (отправить по электронной почте это сообщение) — см. ниже подпоток S4;
- `Return to List of Messages` (вернуться к списку сообщений) — чтобы закрыть это окно и возвратиться к главному окну приложения.

S4 — Email Message (передача сообщения по электронной почте)

Диалоговое окно на рис. 14.4, которое позволяет передать по электронной почте исходящее сообщение, может быть создано подпотоком S3 или данным подпотоком. В подпотоке S3 двойное нажатие клавишей мыши на выбранном исходящем сообщении создает диалоговое окно. В подпотоке S4 то же самое получается путем нажатия кнопки `Email Selected Message` (передать по электронной почте выбранное сообщение).

Успешная передача по электронной почте завершается обновлением БД и информационным сообщением, отображаемым пользователю (рис. 14.5). Сообщение содержит следующий текст: «Ваш запрос был отправлен вашему

Рис. 14.5. Информационное сообщение после передачи по электронной почте

серверу электронной почты для его отсылки. Пожалуйста, используйте ваше ПО для электронной почты, чтобы проверить, что ваше сообщение действительно достигло назначения».

S5 — Create New Message (создание нового сообщения)

Итерация 2 обеспечивает опцию для создания нового исходящего сообщения и сохранения его в БД. Нажатие кнопки `Create New Message` (создание нового сообщения) в основном окне приложения создает диалоговое окно, показанное на рис. 14.6.

Диалоговое окно позволяет выбрать делового партнера и наметить служащего и/или отдел, которые будут ответственны за передачу. Тема и содержание исходящего сообщения могут быть напечатаны в соответствующих полях. Окно содержит четыре командные кнопки:

- `Save` (сохранить) — сохранить новое исходящее сообщение в БД, но оставить диалоговое окно открытым, чтобы иметь возможность создать следующее исходящее сообщение;
- `OK and Return` (все в порядке и вернуться) — сохранить новое исходящее сообщение в БД, а затем закрыть это окно и вернуться в главное окно приложения;
- `Clear` (очистить) — очистить текущее содержимое диалогового окна и разрешить пользователю заново создать новое исходящее сообщение;
- `Cancel and Return` (отменить и вернуться) — отменить операцию, закрыть это окно и вернуться к главному окну приложения.

S6 — Delete Message (удаление сообщения)

Когда исходящие сообщения больше не нужны, возможно, из-за коррекций, сделанных другими служащими, они могут быть удалены из БД. Нажатие кнопки `Delete Message` (удалить сообщение), когда исходящее сообщение выбрано в главном окне, активизирует этот подпоток. В этом случае отображается простое окно, чтобы запросить пользователя подтвердить удаление, как показано на рис. 14.7.

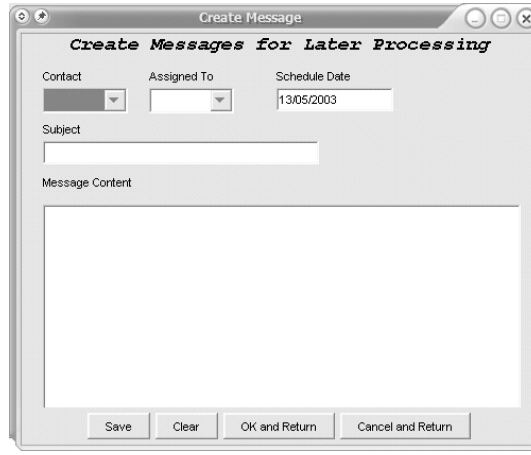


Рис. 14.6. Диалоговое окно для создания новых исходящих сообщений

Рис. 14.7. Подтверждение удаления сообщения

Если пользователь решит отменить удаление, никаких изменений не произойдет.

14.2.4. Потоки исключений

Любое исключение, возникающее в течение времени выполнения, должно быть выявлено и передано пользователю посредством информационного окна сообщения. Программа не должна нарушить свою работу при любом исключении, поступающем от БД, из-за сетевых отказов электронной почты или отказов приложения.

E1 — Incorrect username or password (неправильное пользовательское имя или пароль)

Если в основном потоке актор вводит неправильное пользовательское имя или неправильный пароль, система отображает сообщение об ошибке (рис. 14.8). Сообщение выглядит следующим образом: «Invalid login. Please try again. This will be your second attempt (Неправильное регистрационное имя. Пожалуйста, попробуйте снова. Это будет ваша вторая попытка)». Сценарий использования продолжается и позволяет пользователю заново ввести пользовательское имя и пароль. Вторая неудачная попытка завершается сообщением об ошибке: «Invalid login. Please try again. This will be your last attempt (Неправильное регистрационное имя. Пожалуйста, попробуйте снова. Это будет ваша последняя попытка)». Сценарий использования продолжается и позволяет пользователю заново ввести пользовательское имя и пароль.

Пользователю даются три возможности, чтобы задать правильное пользовательское имя и пароль. Если все три раза будут неудачны, система выводит сообщение: «Exceeded the maximum of three attempts to login. Application will quit (Превышен максимум из трех попыток ввода регистрационного имени. Приложение будет завершено)» — рис. 14.9. Сценарий использования завершается.

Рис. 14.8. Окно сообщения для неправильного регистрационного имени

Рис. 14.9. Окно сообщения после трех неудачных попыток ввести регистрационное имя

E2 — Email could not be sent (сообщение по электронной почте не может быть послано)

Если в подпотоке S4 почтовый сервер возвращает информацию о том, что сообщение по электронной почте не могло быть послано, система сообщает актору, что сообщение не было послано (рис. 14.10). (В окне сообщения выводится текст: «Failed in emailing the message. No network connection or invalid email address — Ошибка при передаче сообщения по электронной почте. Нет сетевого соединения или неправильный адрес электронной почты».) Сценарий использования продолжается после того, как пользователь подтверждает получение сообщения об ошибке, нажав кнопку `OK`.

14.3. Концептуальные классы и реляционные таблицы

Концептуальная модель классов для итерации 2 содержит четыре класса: `Employee` (служащий), `Department` (отдел), `Contact` (деловой партнер) и `OutMessage` (исходящее сообщение) — рис. 14.11. На линиях ассоциаций приведены имена ассоциаций, роли ассоциаций и множественности на обеих их сторонах. Первичные атрибуты (идентификаторы) напечатаны полужирным шрифтом.

`Employee` может быть `employed_by` (служащим) максимум одного `Department`. `Department` `employs` (нанимает) многих `Employee`, но он может существовать, даже если в нем не служит ни один `Employee`. `Department` может иметь намеченные `OutMessages` (исходящие сообщения). Может быть максимум один `Department` для каждого `OutMessage`.

Рис. 14.10. Окно сообщения после ошибки при передаче сообщения по электронной почте

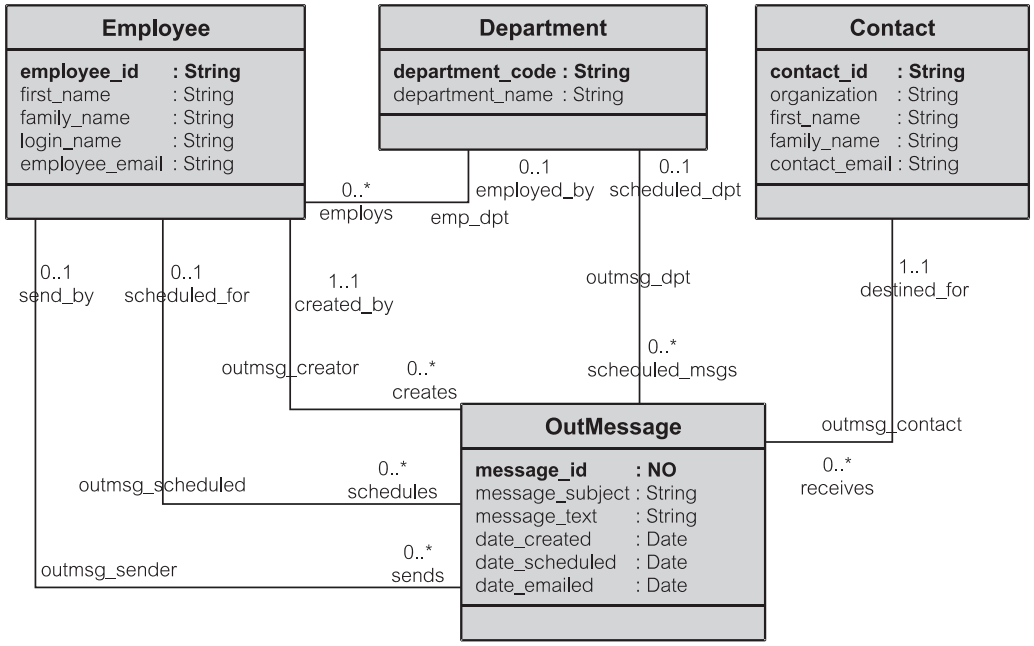


Рис. 14.11. Концептуальные классы для итерации 2

Имеется точно один `Contact` для каждого `OutMessage`. `Contact` может быть связан с нулем или многими `OutMessage`.

Ассоциация по имени `outmsg_creator` (создатель исходящего сообщения) связывает `OutMessage` с `Employee`, который создал это `OutMessage` в БД. Ассоциация по имени `outmesg_scheduled` (намеченный для исходящего сообщения) определяет `Employee`, который намечен для передачи исходящего сообщения по электронной почте. Эта ассоциация связывается с нулем `Employee`, если `OutMessage` не намечено никакому `Employee`. Ассоциация по имени `outmesg_sender` (отправитель исходящего сообщения) определяет `Employee`, который передает по электронной почте `OutMessage`. Эта ассоциация связана с нулем `Employee`, пока `OutMessage` не отослано по электронной почте.

Большинство атрибутов имеет тип `string` (строка). Тип атрибута `message_id` (идентификатор сообщения) — `number` — число (показанное как `NO`). Три атрибута дат имеют тип `date` (дата).

Согласно правилам объектно-реляционного отображения, объясненным в главе 10, концептуальная модель классов может быть автоматически преобразована в реляционную модель БД. Рис. 14.12 показывает реляционную модель, полученную из концептуальной модели на рис. 14.11. Эта модель очевидна. Область индексов содержит индексы, созданные на первичных ключах (`PK` — primary keys), на внешних ключах (`FK` — foreign keys) и на уникальных столбцах (`UN` — unique columns).

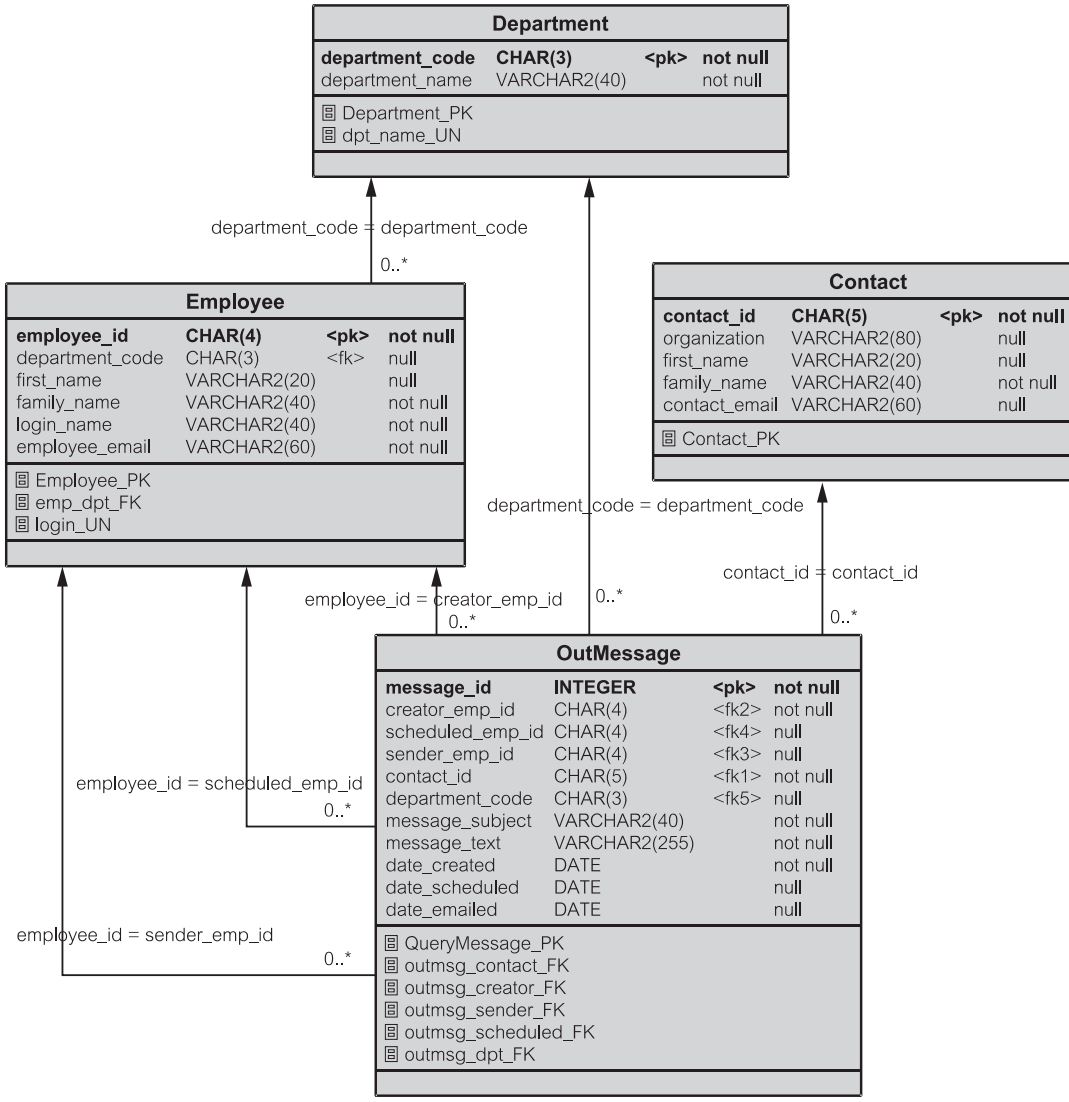


Рис. 14.12. Реляционные таблицы для итерации 2

14.4. Дополнительная спецификация

Дополнительные спецификации существенно не отличаются от подобных спецификаций в итерации 1. Изменения являются результатом добавления к программе интерфейса в виде рабочего стола и/или Web-доступного GUI-интерфейса и результатом дополнительных функциональных возможностей.

- Функциональные возможности
  - EM — многопользовательское приложение.

- Авторизация пользователя и права доступа к различным средствам приложения управляются централизованно из БД, с которой соединена прикладная программа.

- Применимость
  - Для грамотного в отношении к компьютерам пользователя не требуется никакого обучения, чтобы иметь возможность использовать итерацию 2 программы. Достаточно простого объяснения цели и основных особенностей приложения, чтобы использовать программу.
  - Итерация 2 представляет собой GUI-программу с использованием рабочего стола и/или Web-доступного интерфейса, которая обеспечивает интуитивный подход к использованию различных опций приложения. Одновременно может потребоваться только одна опция. Создание всех вторичных окон, **модальных** в отношении к первичному окну приложения, обеспечивает это требование. (*Модальное окно* не позволяет переключаться на другие окна приложения без первоначального завершения обработки в этом окне и закрытия его.)
  - Выполнение итерации 2 может быть развернуто на **GUI-рабочем столе** ОС Windows или с помощью **GUI на основе Web-технологии**.

- Надежность
  - Приложение должно быть доступно 24 часа каждый день недели. Не должно быть никакого связанного с БД времени простоя. О любом намеченном отключении для профилактики сервера электронной почты ЕМ-пользователи электронной почты должны быть уведомлены, по крайней мере, за 24 часа.
  - Отказ программы не должен нарушать правильность и целостность БД. Пользователь должен иметь возможность повторно начать программу после отказа и удостовериться, что информация БД является непротиворечивой и не пострадала в результате отказа.

- Выполнение функций
  - Нет никакого верхнего предела числа параллельных пользователей.
  - Время отклика системы не изменяется, если число параллельных пользователей — 100 или менее.
  - Время отклика для подпотоков S1 и S5 должно быть меньше 10 секунд в 90 процентах случаев.
  - Время отклика для подпотоков S2 и S3 должно быть меньше 5 секунд в 90 процентах случаев.
  - Время отклика для подпотока S4 должно быть меньше 10 секунд в 90 процентах случаев для сообщений по электронной почте, не превышающих 1 МБ в размере (включая любые добавленные документы). Однако обратите внимание, что итерация 2 не позволяет использовать приложения в электронной почте.

- Возможность сопровождения
  - Структурное проектирование системы должно соответствовать PCMEF-шаблону, чтобы обеспечить надлежащее удобство сопровождения и масштабируемость.

- Для создания кода используется управляемая тестированием разработка. Для контроля кода используются приемочные испытания. Испытательные единицы, полученные при управляемой тестированием разработке и в процессе приемочных испытаний, используются для регрессионного тестирования, если код итерации 2 будет изменен.
- Другие ограничения
  - Проект должен использовать БД Oracle, но он должен быть легко перестраиваемым на другие реляционные БД.
  - Итерация 2 должна использовать Java и JDBC или SQLJ для доступа из программы к БД Oracle.

Резюме

1. Итерация 2 добавляет GUI-внешний интерфейс к подсистеме EM. Кроме рабочего стола GUI, в итерации 2 имеются версии апплетов и сервлетов, позволяющие осуществлять доступ к системе с помощью браузера Интернета.
2. Итерация 2 представляет дополнительные критерии, включая фильтры, по которым могут отбираться исходящие сообщения. Она также позволяет пользователю создавать новые исходящие сообщения в интерактивном режиме.
3. Концептуальная модель классов для итерации 2 содержит один новый класс (`Department` — отдел) и ряд новых ассоциаций.

Ключевые термины

Обзорные вопросы

1. Объясните зависимости реализации между сценариями использования `View Unsent Messages` (просмотр непосланных сообщений) и `Filter Messages` (фильтрация сообщений).
2. Что делает элемент управления *Toggle Filter* (переключатель фильтра)?
3. Объясните необходимость четырех командных кнопок в подпотоке *S3 — Create New Message* (создать новое сообщение).
4. Атрибут `sheduled_emp_id` (идентификатор намеченного служащего) в `OutMessage` (исходящее сообщение) — рис. 14.11 — не является обязательным (то есть, может принимать значение `null`). Может ли этот атрибут быть смоделирован как обязательный? Объясните.
5. Объясните взаимозависимости между подпотоками *View Department Unsent* (представление непосланных отделом сообщений) и *Filter by Department* (фильтр по отделу).

Глава

15 Структурный рефакторинг

Итерация 1 из учебного примера EM была завершена работающим исполняемым продуктом, который можно было продемонстрировать клиентам проекта и другим заинтересованным сторонам. Ясная и устойчивая PCMEF-структура (глава 9) — особое достоинство итерации 1 подлежащего сдаче продукта. Следовательно, ПО *приемлемо*, то есть, оно понятно, ремонтопригодно и расширяемо.

Однако справедливо сказать, что из этих трех особенностей итерация 1 подчеркивает понятность ПО, представляемую минимизацией связей объектов. Минимизация связей объектов может легко привести к слабому объединению объектов и по этой причине к трудностям с удобством сопровождения и масштабируемостью. Есть настоятельная необходимость в лучшем балансе между всеми тремя особенностями. Этот лучший баланс может быть достигнут рефакторингом.

«**Рефакторинг** — процесс изменения системы ПО таким способом, что при этом не изменяется внешнее поведение кода, но при этом все же улучшается его внутренняя структура» [30]. Рефакторинг — нечто вроде чистки кода после того, как он был написан. Пока его цели — решение потенциальных проблем в проекте. Фаулер называет эти проблемные области «дурно пахнущими в коде».

Рефакторинг очень хорошо сочетается с **быстрой разработкой** [66]. Он может проводиться в любом месте итерации, но наиболее эффективно выполнять его либо в конце текущей итерации, либо в начале следующей. Рефакторинг может существенно улучшить внутреннюю структуру кода без изменения его внешнего поведения.

Можно сказать, что практика управляемой тестированием разработки (раздел 12.2) является частичной заменой рефакторинга. Действительно, управляемая тестированием разработка использует разновидность рефакторинга — разновидность, которая применяется для улучшения скорее самого проекта, а не кода. Управляемая тестированием разработка — итеративный и пошаговый процесс, объединенный с написанием прикладного кода. Рефакторинг может предугадать «дурно пахнущий код» и устранить его до того, как это случится.

В этой книге рефакторинг используется в начале итерации 2, чтобы реструктурировать код итерации 1 для удобства сопровождения и масштабируемости и подготовить его к выполнению итерации 2. Главные изменения будут на уровне domain (предметная область); в меньшей мере — на уровнях control (управление) и foundation (основание). Уровень presentation (представление) не улучшается, потому что итерация 2 эффективно заменяет уровень presentation итерации 1 (и любой рефакторинг был бы сразу же потерян).

Обратите также внимание, что разработка и код итерации 1 условно оптимальны по педагогическим причинам. Многие важные объектно-ориентированные технологии не использовались в итерации 1 из-за заложенной в книге педагогики и принятой последовательности представления тем. Соответственно, обсуждение рефакторинга в этой главе также условно оптимально.

15.1. Цели рефакторинга

Фаулер [30] определяет и называет весь диапазон «дурно пахнущего кода». Это — один вид **целей рефакторинга**. Другие цели — не столько результат «дурного запаха», сколько желание далее улучшить код, в особенности, улучшить его структуру [31].

Несколько наиболее интересных целей рефакторинга, вызванных «дурным запахом», следующие:

- дублированный код — одни и те же части кода в нескольких местах;
- длинная подпрограмма-метод — метод, который делает слишком много;
- большой класс — класс, который делает слишком много и/или имеет слишком много элементов данных;
- длинный список параметров — слишком большое количество данных передается в качестве параметров (вместо того, чтобы запросить данные от других объектов);
- расширяющееся изменение — когда класс должен быть изменен в результате больше чем одного вида изменения;
- эффект дробовика — когда одно изменение воздействует на несколько классов;
- излишняя зависимость — метод, который обращается ко многим другим объектам с помощью сообщений `get` (получить), чтобы получить данные для собственных вычислений;
- группы данных — данные (элементы данных, параметры), которые обычно используются вместе во многих местах и которые следует преобразовать в объект.

Цель рефакторинга имеет множество *методов и паттернов рефакторинга*, чтобы устранить «дурные запахи» или внести структурные усовершенствования. Иногда один и тот же метод или паттерн рефакторинга может использоваться, чтобы решать (или частично решать) больше чем одну цель рефакторинга.

15.2. Методы рефакторинга

В жизненном цикле ПО время и трудозатраты на сопровождение кода существенно превышают время и трудозатраты на первоначальное написание кода. При сопровождении кода необходимо прочитать и попробовать понять его, чтобы затем изменить или расширить. Любой рефакторинг кода в процессе его создания, независимо от того, сколь он мал, может принести существенную пользу эксплуатационному персоналу ПО.

Методы рефакторинга (или просто **рефакторинги**) — основные принципы и лучшая практика изменения кода с целью улучшения его понятности, удобства сопровождения и масштабируемости. Изменения кода небольшие, по одному шагу, но усовершенствования могут быть весьма впечатляющими. К сожалению, можно также и ухудшить код, если рефакторинги сделаны не должным образом.

В современной практике CASE- и инструментальные средства разработки программ могут существенно помочь в реализации рефакторинга. Многие инструментальные средства содержат каталоги поддерживаемых рефакторингов. Фаулер [30] является основным источником ссылок, который перечисляет и документирует более шестидесяти методов рефакторинга. Следующее обсуждение иллюстрирует использование методов рефакторинга, рассматривая только три из них:

- *Класс извлечения (Extract Class);*
- *Метод подключения (Subsume Method);*
- *Интерфейс извлечения (Extract Interface).*

15.2.1. Класс извлечения

Одна из целей использования рефакторинга называется *большим классом* — классом, который делает слишком много и/или имеет слишком много элементов данных. Большие классы могут возникнуть из-за чрезмерной минимизации связей между классами. В случае большого класса в коде итерации 1 уместны два метода: *Класс извлечения* (Extract Class) и *Интерфейс извлечения* (Extract Interface) [30].

Рефакторинг Класс извлечения определяется так: «Создайте новый класс и переместите соответствующие поля и методы из старого класса в новый» [30]. Главная трудность заключается в определении, как разбить большой класс на ряд меньших классов. Идея заключается в том, чтобы извлечь непротиворечивые и объединенные части функциональных возможностей в отдельный класс (классы).

Как только поля и методы будут перемещены в новый класс (классы), должна быть установлена связь ассоциации от старого класса к новому. Это может быть односторонняя связь, если нет очевидной потребности в обратной передаче сообщений. Конечно, перемещение методов должно влиять на классы клиента, которые зависели от этих методов. Это требует изменений в обращениях к методам в этих классах.

Рис. 15.1 показывает, как рефакторинг *Класс извлечения* мог бы быть использован для класса `CActioner` — класс «исполнитель» пакета `control`

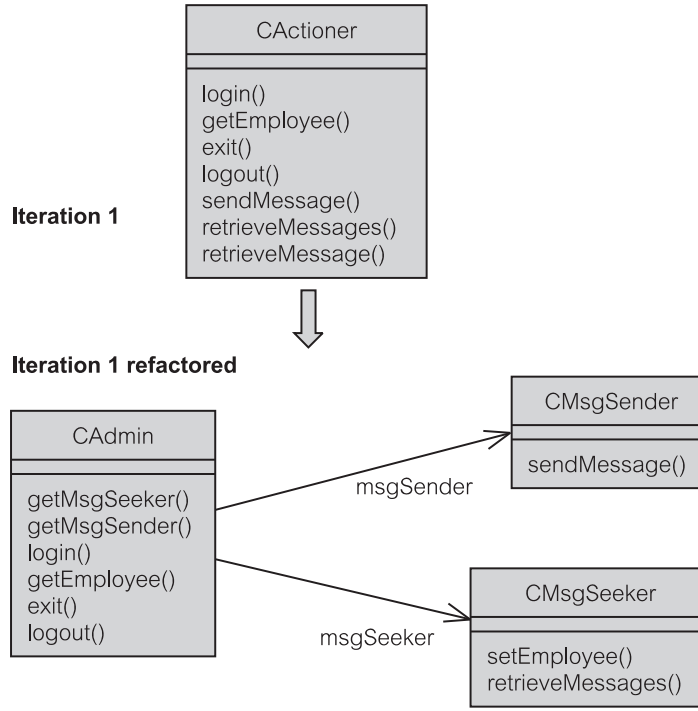


Рис. 15.1. Рефакторинг Класс извлечения

(раздел 13.4.1). Класс `CActioner` включен в две совершенно несопоставимые задачи: извлечение исходящих сообщений, требуемых пользователем, и посылка исходящих сообщений (передача по электронной почте). Логично извлечь эти две задачи в отдельные классы: `CMsgSeeker` (класс «поиск сообщения» пакета `control`) и `CMsgSeeker` (класс «передача сообщения» пакета `control`). Чтобы избежать терминологического беспорядка, `CActioner` переименован в `CAdmin` (класс «администратор» пакета `control`). Конструктор и методы, не являющиеся общедоступными, не рассматриваются.

Как ожидается, с помощью рефакторинга *Класс извлечения* `CAdmin` обеспечивает связи ассоциации с новыми классами. Связи поддерживаются двумя методами: `getMsgSeeker()` (получение поиска сообщения) и `GetMsgSender()` (получение передачи сообщения). Первый получает объект `CMsgSeeker`, который отвечает за извлечение исходящих сообщений для уровня `presentation` (представление). Второй получает объект `CMsgSender`, ответственный за исходящие сообщения, передаваемые по электронной почте.

Рефакторинг дает один новый метод (`setEmployee()` — задать информацию о служащем). Метод `setEmployee()` используется, чтобы установить ассоциацию от `CMsgSeeker` к `EEmployee` (класс «служащий» пакета `entity`). Это требуется, потому что извлечение исходящего сообщения требует данных о служащем. Метод `setEmployee()` получает объект `EEmployee` от метода `login()` (регистрационное имя) класса `CAdmin()`.

15.2.2. Метод подключения

Дублированный код (те же самые части кода в нескольких местах) является частой целью рефакторинга. В зависимости от своей природы к дублированному коду можно подойти различными способами. Типичный рефакторинг для дублированного кода — *Метод извлечения*, то есть, преобразование кода, дублированного в нескольких методах, в отдельный метод [30]. Другой подходящий рефакторинг, не перечисленый Фаулером, — Метод подключения (Subsume Method). **Рефакторинг Метод подключения** устраняет метод включением его функциональных возможностей в другой существующий метод.

Рефакторинг *Класс извлечения* на рис. 15.1 показывает, что метод `retrieveMessage()` (извлечение сообщения) класса `CActioner` (класс «исполнитель» пакета `control`) отсутствует в улучшенном коде. Метод `retrieveMessage()` удален не рефакторингом *Класс извлечения*. Это реализуется *Методом подключения*, который объединяет методы `retrieveMessage()` в метод `retrieveMessages()` (извлечение сообщений).

Неудачно, что код итерации 1 может быть запущен двумя разными опциями меню, доступными пользователю: *View Unsent Messages* (просмотр непосланных сообщений) и *Display Text of a Message* (отображение текста сообщения) — рис. 12.12. Первая опция приводит к извлечению и отображению нескольких исходящих сообщений; вторая — к извлечению и отображению одного исходящего сообщения, которое пользователь предназначает для передачи по электронной почте. В обоих случаях извлекается полное содержание данных объектов `EOutMessage` (класс «исходящее сообщение» пакета `entity`), но отображение информации пользователю изменяется.

Суть проблемы в том, что имеется большое количество дублированного кода, вытекающее из независимой обработки этих двух действий меню, как видно в разделе 13.6.1. Конечно, извлекая несколько исходящих сообщений и извлекая единственное исходящее сообщение, используется схожая обработка. Рис. 15.2 демонстрирует две последовательности передачи сообщений для итерации 1 и то, как они объединены в единственный путь, начинающийся с уровня `control` (управление). Различие между несколькими исходящими сообщениями и единственным исходящим сообщением очевидно только на уровне `presentation` (представление).

15.2.3. Интерфейс извлечения

Цель рефакторинга Интерфейс извлечения двойная и определяется так: «Несколько клиентов используют то же самое подмножество интерфейса класса или два класса содержат общую часть своих интерфейсов» [30]. Метод **рефакторинга Интерфейс извлечения** используется, чтобы «выделить подмножество в интерфейс» (там же). Идея относительно этого рефакторинга связана с самой природой интерфейсов (раздел 9.1.7).

Подобно *Классу извлечения* (раздел 15.2.1) *Интерфейс извлечения* часто является ответом на цель, связанную с большим классом. Это — периодически повторяющаяся цель рефакторинга в итерации 1. Класс `PConsole` (класс

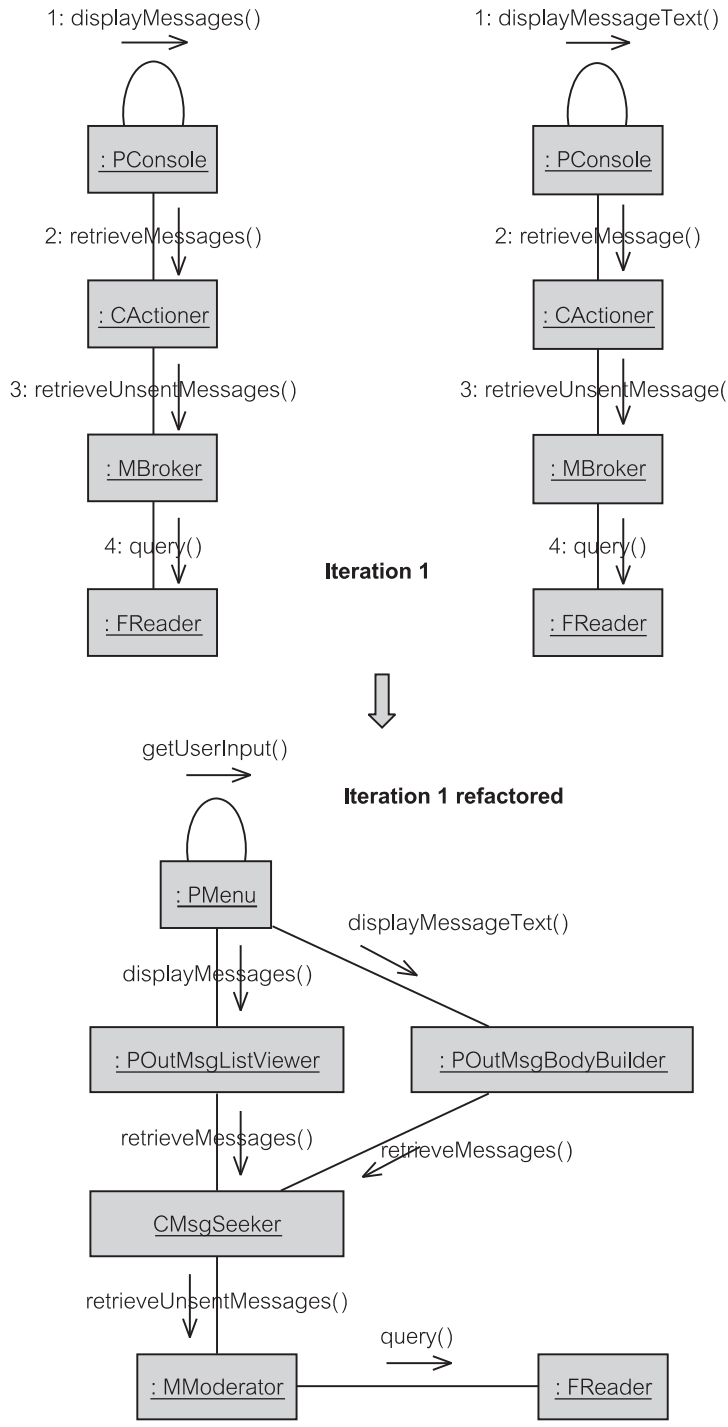


Рис. 15.2. Рефакторинг Метод подключения

«консоль» пакета `presentation`) — большой класс, который делает слишком много. В зависимости от команды меню, выбранной пользователем, `PConsole` исполняет различные задачи. Например, может потребоваться показать список исходящих сообщений или детали отдельного исходящего сообщения.

В итерации 1 выполнение этих различных задач инкапсулировано методом `displayMenu()` (отобразить меню) — раздел 13.3.2, листинг 13.16. Этот метод содержит приватный (private) метод `getUserInput()` (получить введенную пользователем информацию), который получает запрос пользователя и останавливается на задаче, вызывая соответствующий метод `PConsole`, наподобие `displayMessages()` (отобразить сообщения). Класс `PConsole` имеет только два общедоступных (public) метода: `displayLogin()` (отобразить регистрационное имя) и `displayMenu()`. Все остальное инкапсулировано (скрыто). Инкапсуляция имеет преимущества, но вызывает «запахи» большого класса, которые нелегко обслуживать и устранить.

Рефакторинг класса `PConsole` может сначала использовать метод *Класс извлечения*, чтобы создать, например, два отдельных класса для отображения списка исходящих сообщений и отдельного исходящего сообщения (рис. 15.3). Соответствующие обязанности от `PConsole` (переименованного теперь в `PMenu` — класс «меню» пакета `presentation`) берут на себя два новых класса `POutMsgListViewer` (класс «просмотр списка исходящих сообщений» пакета `presentation`) и `POutMsgBodyBuilder` (класс «формирование тела исходящего сообщения» пакета `presentation`).

Естественный следующий шаг должен подтвердить, что два новых класса уровня `presentation` (представление) в улучшенном коде используют один и тот же метод `display()` (отобразить), чтобы показать строку на экране. Ясно, что код улучшит удобочитаемость и документирование, если метод `display()` будет извлечен в интерфейс, реализован в `PMenu` и использован при необходимости новыми классами уровня `presentation`.

Рис. 15.3 представляет улучшенный проект, используя *Класс извлечения* и *Интерфейс извлечения* на уровне `presentation`. Рефакторинг этого уровня для итерации 2 имеет ограниченный объем. Переход в итерации 2 к графическому пользовательскому интерфейсу означает весьма радикальную реорганизацию уровня `presentation`. Даже в рассматриваемом случае наличие интерфейса и новых классов облегчит этот переход. В некоторых случаях был бы оправдан более радикальный рефакторинг. Например, класс `POutMsgBodyBuilder` может быть разбит на два класса, чтобы отделить отображение исходящего сообщения от подготовки его для передачи по электронной почте.

15.3. Паттерны рефакторинга

Рефакторинг неотделим от структурного проектирования и разработки шаблонов. Существенные структурные усовершенствования могут быть достигнуты скоординированным применением многих методов рефакторинга. Начальное проектирование системы управляется структурными паттернами, как рассмотрено в разделе 9.3. Структурные паттерны также полезны в рефакторингах, чтобы получить более существенные исправления или расширения кода.

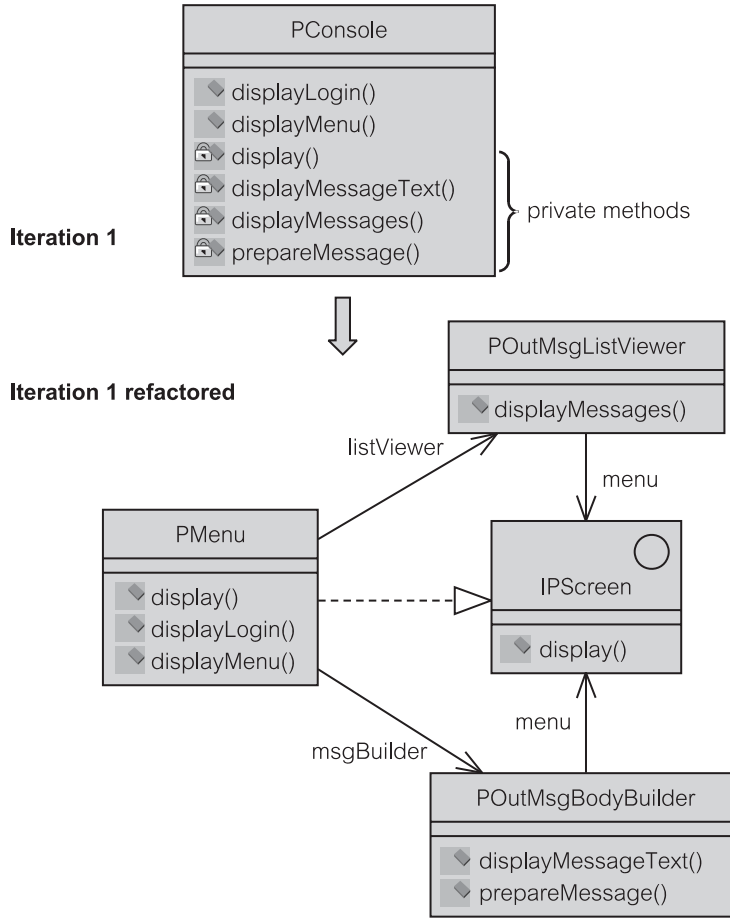


Рис. 15.3. Рефакторинг Интерфейс извлечения

Паттерны рефакторинга — структурные паттерны, используемые в рефакторинге кода. Фаулер [31] называет их паттернами структуры промышленного приложения. Они являются целью тех аспектов системы, которые делают ее промышленным приложением в противоположность маломасштабным настольным приложениям. Они имеют дело с такими проблемами, как связь с БД, кэши сохраняемых объектов, расположенные в памяти, управление транзакциями и параллелизмом, представление Web-технологий, работа с распределенными объектами и т. д.

Паттерны рефакторинга обеспечивают системе более легкую масштабируемость, когда она перерастает в крупномасштабное решение. Итерация 1 учебного примера ЕМ подчеркивает присутствие БД в приложении при сохранении весьма примитивного текстового пользовательского интерфейса. Итерация 2 делает шаг к графическому пользовательскому интерфейсу. Паттерны рефакторинга, рассматриваемые в данной главе, все еще применяются

к итерации 1 и поэтому концентрируются на проблемах, отличных от проблем пользовательского интерфейса.

Следующее рассмотрение иллюстрирует структурные паттерны, используемые для рефакторинга кода итерации 1 в пределах уровня `domain` (предметная область). Большинство этих паттернов описано у Фаулера [31]. Книга Фаулера является источником ссылок, находящимся на современном уровне, и которая перечисляет и документирует более пятидесяти структурных паттернов. Паттерны представлены следующими группами:

- *Коллекция идентичности объектов* (Identity Map);
- *Преобразователь данных* (Data Mapper);
- *Загрузка по требованию* (Lazy Load);
- *Единица работы* (Unit of Work).[1]

15.3.1. Коллекция идентичности объектов

Итерация 1 подтверждает важность назначения идентификаторов объектов (object identifier — OID) объектам пакета `entity` — сущность (раздел 13.5) и важность поддержания кэшей пакета `entity` с помощью пакета `mediator` — посредник (раздел 13.6). Это, конечно, хорошо, но имеющийся в настоящее время класс `MBroker` (класс «посредник» пакета `mediator`) «пахнет» большим классом, поэтому можно использовать ряд *паттернов рефакторинга*, чтобы улучшить управление идентификацией и кэшированием. Один такой паттерн — **Коллекция идентичности объектов** (Identity Map) [31].

Паттерн *Коллекция идентичности объектов* «гарантирует, что каждый объект загружается только однажды, храня ссылку на каждый загруженный объект в коллекции. При использовании объектов просматривается коллекция, содержащая ссылки на них» [31]. Объект *Коллекция идентичности объектов* поддерживает одну или несколько коллекций (например, хэш-коллекции), которые содержат идентификаторы объектов (раздел 12.1.2). *Коллекция идентичности объектов* имеет два основных назначения:

- гарантировать, что одна и та же запись БД не загружена несколько раз в различные объекты приложений (это создало бы хаос, если приложение изменило бы один из этих объектов, а другие — нет);
- избежать ненужной загрузки одних и тех же данных несколько раз (это воздействовало бы на выполнение функций приложения).

Фаулер [31] различает явно заданные и общие коллекции идентичности объектов. К объектам в *явно заданной коллекции идентичности объектов* можно иметь доступ (`get` — получить), их можно зарегистрировать (`put` — поместить) и снять с них регистрацию (`remove` — удалить) с использованием различных методов для каждого класса кэшированных объектов. Например, для получения доступа к объекту:

```
getEEmployee (new Integer(empOID));
```

[1] Здесь также можно отметить, что русская терминология паттернов еще не устоялась, и порой используются названия, не соответствующие английским названиям. Так, например, паттерн Identity Map часто называют Коллекцией объектов, что, на наш взгляд, искажает его суть. Мы выбрали здесь наиболее подходящую по нашему мнению терминологию. (*Прим. перев.*)

Для объектов в *общей коллекции идентичности объектов* доступ (`get`), регистрация (`put`) и снятие регистрации (`remove`) выполняются единственным методом для всех классов. Параметр метода определяет класс. Например, для получения доступа к объекту:

```
get ("EEmployee", new Integer(OID));
```

Общая коллекция может использоваться только в тех случаях, если все идентификаторы объекта формируются одним и тем же алгоритмом и глобально уникальны для всех объектов всех классов. Общая коллекция позволяет легко создавать и поддерживать единую регистрацию идентификаторов объектов, так что другие объекты могут использовать ее, чтобы найти объект по его OID.

Общая коллекция имеет преимущество удобочитаемости и документирования. Все доступные коллекции статически объявлены в коде. Дополнения и удаления в коллекциях ограничены единственным классом, так что добавление и удаление новых методов не является проблемой.

Далее Фаулер [31] различает одну коллекцию на класс и одну коллекцию на сеанс. *Одна коллекция на сеанс* требует глобальных уникальных идентификаторов объектов. В этом смысле она идет рука об руку с общей коллекцией идентификации объектов. *Одна коллекция на класс* может использовать идентификаторы объектов, уникальные в пределах класса, но здесь возникает проблема, как управлять классами в дереве наследования. В обоих случаях для коллекций идентичности обновляемых объектов должна быть обеспечена защита в виде транзакций и они должны быть помечены как измененные, когда объекты идентичности выходят из синхронизации с соответствующими записями БД.

В случае итерации 1 результатом паттерна *Коллекция идентичности объектов*, так же как и метода *Класс извлечения*, является рефакторинг, который приводит к новому классу `EIdentityMap` — класс «коллекция идентичности объектов» пакета `entity` (листинг 15.1). `EIdentityMap` определяет два вида коллекций (строки 19–20). Одни коллекции содержат `OID` пакета `entity`, а другие коллекции содержат `msgID` (идентификатор объектов-сообщений) к `OID EOutMessage` (класс «исходящее сообщение» пакета `entity`). Итерация 1 не требует других коллекций, но они могут быть добавлены всякий раз, когда это будет необходимо. Пример показывает методы, необходимые, чтобы управлять размещением объектов `EContact` (класс «деловой партнер» пакета `entity`).

Листинг 15.1. Класс EIdentityMap — фрагмент

Класс EIdentityMap — фрагмент

```
18:  public class EIdentityMap {
21:     private Map OIDToObj; //OID -> Obj
22:     private Map msgPKToOID; //msgPK -> OID
23:
24:     public EIdentityMap() {
25:       OIDToObj = new HashMap();
```

```
28:       msgPKToOID = new HashMap();
29:     }
30:
31:     /** Получить хранимого делового партнера */
32:     public IAContact findContact(int contactOID) {
33:       return (IAContact) OIDToObj.get(new
                                    Integer(contactOID));
34:     }
35:     /** Разместить делового партнера с заданным OID */
36:     public void registerContact(IEObjectID oidObject) {
37:       OIDToObj.put(new Integer(oidObject.getOID()),
                                                 oidObject);
41:     }
42:     /** Снять регистрацию с зарегистрированного делового
                                          партнера */
43:     public void unregisterContact(IEObjectID oidContact) {
44:       OIDToObj.remove(new Integer(oidContact.getOID()));
46:     }
113: }
```

`EIdentityMap` — *явно заданная коллекция идентичности объектов* — она имеет отдельные методы для каждого класса пакета `entity`. Класс представляет стратегию *одной коллекции на сеанс* — он использует глобальные уникальные идентификаторы `OID`.

15.3.2. Преобразователь данных

Паттерн Преобразователь данных (Data Mapper) определяется как «уровень *Преобразователей* (Mappers), который перемещает данные между объектами и БД, когда они хранятся независимо друг от друга, и непосредственно сам преобразователь» [31]. Общий паттерн *Преобразователь*, упомянутый в определении, является «объектом, который устанавливает связь между двумя независимыми объектами» [31]. Паттерн *Преобразователь* отличается от паттерна *Посредник* (раздел 9.3.5) в том, что объекты, выделенные объектом-преобразователем, не ведают друг о друге, в то время как объекты, использующие посредника, знают об этом.

В PCMEF-шаблоне *Преобразователь данных* принадлежит пакету `mediator` (посредник). В итерации 1 класс `MBroker` — класс «посредник» пакета `mediator` (раздел 13.6.1) отвечает, среди прочего, за знание объектов пакета `entity` (сущность), расположенных в кэше, и за требование извлечения данных из БД, если объект не находится в кэше или если кэш изменен. Эти обязанности *Преобразователя данных* необходимо извлечь из `MBroker` в отдельный класс (классы), используя рефакторинг *Класс извлечения* (раздел 15.2.1). `MDataMapper` (класс «преобразователь данных» пакета `mediator`) — один такой новый класс в улучшенном коде итерации 1.

Рис. 15.4 представляет UML-**диаграмму деятельности** (раздел 2.2.5), которая показывает, как `MDataMapper` отделяет пакет `entity` (ответственный за кэш) от пакета `foundation` — основание (ответственного за доступ

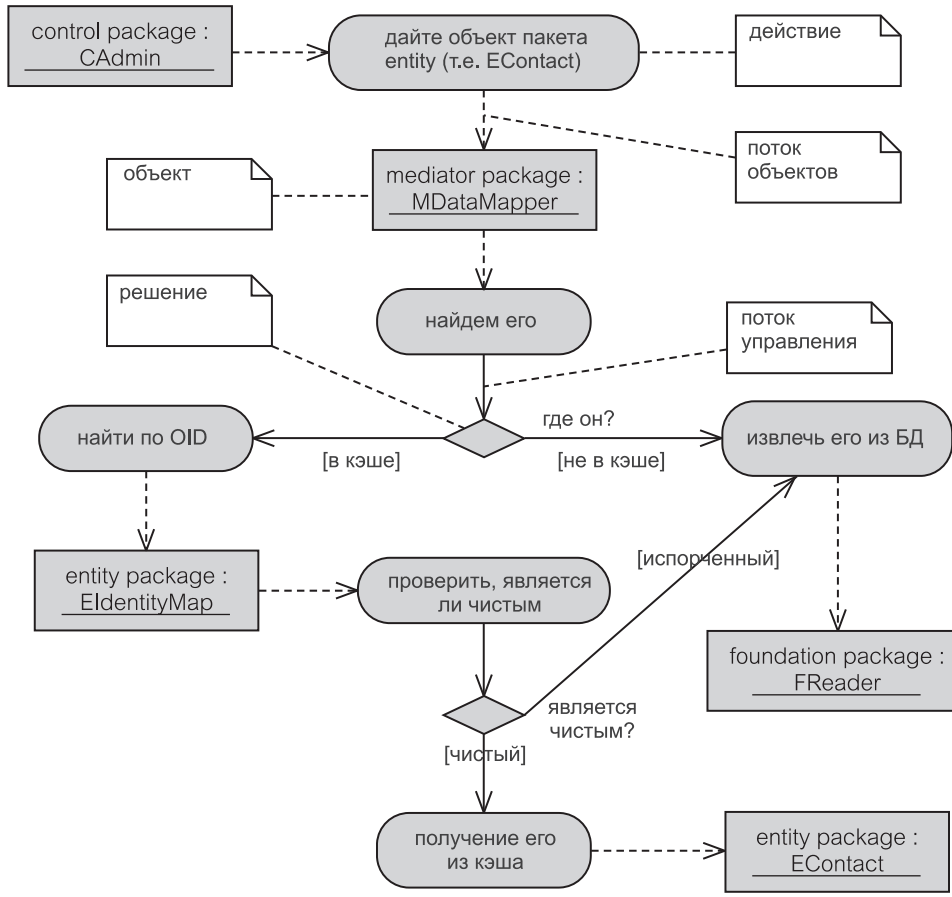


Рис. 15.4. Диаграмма деятельности для MDataMapper

к БД). Диаграмма деятельности используется здесь скорее для удобства, нежели как демонстрация ее обычной применимости.

Семантике модели действий на рис. 15.4 легко следовать. Значение графических объектов объясняется в примечаниях UML. Прежде всего объект управления (`CAdmin` — класс «администратор» пакета `control`) запрашивает `MDataMapper`, чтобы тот получил объект пакета `entity`. `CAdmin` может выполнить этот запрос рядом способов:

- `CAdmin` может знать OID объекта и передать его объекту `MDataMapper` или
- `CAdmin` может знать значения некоторых атрибутов объекта, например, значение первичного ключа, и передать эту информацию объекту `MDataMapper` как условие поиска или
- `CAdmin` может запросить ссылочный поиск, обратившись к `MDataMapper`, чтобы он нашел объекты, которые связаны с объектом, известным объекту `CAdmin`.

MDataMapper должен установить, где находится объект (объекты). Находится ли он в кэше памяти, или он должен быть извлечен из БД? Если он находится в кэше памяти, то является ли он измененным? Если да, то он должен быть извлечен из БД. Объект EIdentityMap — класс «коллекция идентичности объектов» пакета entity (листинг 15.1) знает, находится ли нужный объект в кэше. Если EIdentityMap содержит объекты, и те не изменены, они будут возвращены объекту MDataMapper и затем объекту CAdmin. Если же нет, MDataMapper формирует соответствующие SQL-запросы и передает их объекту FReader (класс «чтение» пакета foundation) для доступа к БД.

Загрузка — импорт

Рис. 15.4 не содержит полный набор операций MDataMapper. Когда записи данных будут извлечены их БД, MDataMapper превращает их в объекты (то есть, MDataMapper создает новые объекты пакета entity). Затем новые объекты регистрируются в EIdentityMap. Это — операция **загрузки** (записи БД извлекаются и преобразуются в объекты памяти). С точки зрения БД загрузка является операцией **импорта из БД — check-out** (записи импортируются в программу). Процесс загрузки также называется **материализацией** [55].

Фрагмент диаграммы последовательности действий на рис. 15.5 иллюстрирует возможный сценарий для загрузки объекта в улучшенном коде итерации 1. Диаграмма касается взаимодействия «Регистрационное имя» (раздел 11.4.1). Она показывает, как MDataMapper извлекает и создает новый объект EEmployee (класс «служащий» пакета entity), когда его запрашивают,

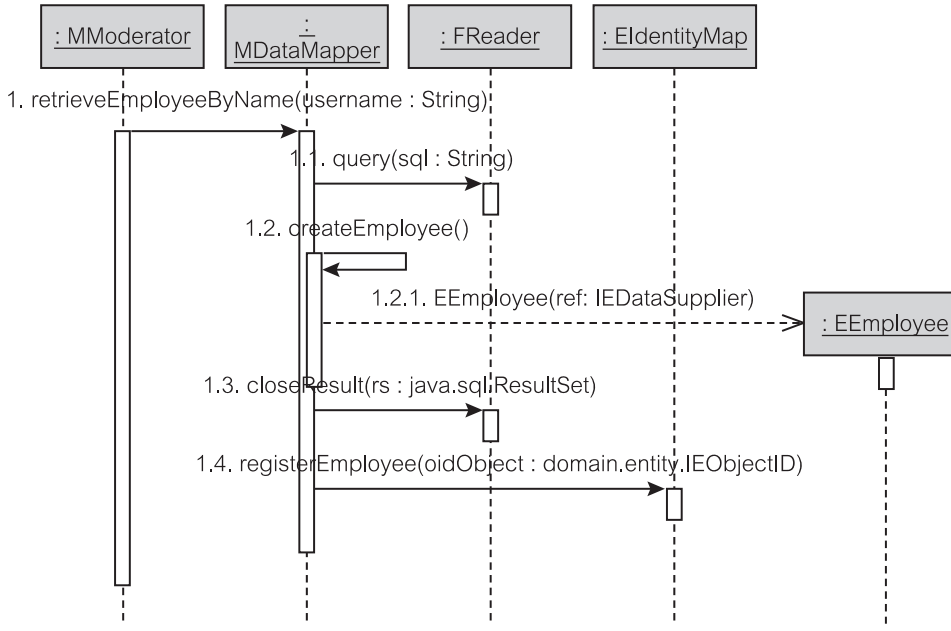


Рис. 15.5. Диаграмма последовательности для загрузки объекта

чтобы получить имя служащего. Эта операция включает формирование OID для EEmployee, который затем передается объекту EIdentityMap в параметре метода registerEmployee() (регистрация служащего).

Выгрузка — экспорт

Вполне естественно, что если *Преобразователь данных* ответствен за загрузку объектов из БД, он должен также обеспечивать и противоположную операцию — выгрузку объектов в БД. С точки зрения БД это операция **экспорта в БД — check-in**. Процесс **выгрузки** также известен как **пассивация** или **дематериализация** [55]. Операция выгрузки требуется, когда:

- приложением создан новый объект пакета entity, и необходимо его постоянно хранить в БД или
- объект пакета entity обновляется приложением, и изменения должны постоянно храниться в БД или
- объект пакета entity удален приложением и поэтому соответствующая запись должна быть удалена и из БД.

Выгрузка является темой исследования бизнес-транзакций, а также операций commit (фиксация) и rollback (откат). Тема управления транзакциями подробно рассматривается в части 4 книги в контексте итерации 3 учебного примера.

Итерация 1 не создает и не удаляет объекты пакета entity, но она должна корректировать объекты EOutMessage (класс «исходящее сообщение» пакета entity), чтобы указать, что они были переданы по электронной почте. Это — задача взаимодействия «Сообщение, передаваемое по электронной почте» (раздел 11.4.5). Фрагмент диаграммы последовательности действий на рис. 15.6 иллюстрирует, как «скорректированная» выгрузка может быть реализована в улучшенном коде итерации 1.

CMsgSeeker (класс «поиск сообщения» пакета control) посылает исходящее сообщение и затем обращается к MModerator (класс «координатор» пакета mediator), чтобы тот выполнил метод updateMessage() (скорректировать сообщение). MModerator делегирует эту задачу объекту MDataMapper, который создает оператор корректировки SQL. FWriter (класс «запись» пакета foundation) делает изменение в БД. Это разрешает объекту MDataMapper выполнить метод unregisterMessage() (снять регистрацию сообщения) в объекте EIdentityMap. Частью этой операции является выполнение метода flagCache() (установить флаг кэша), который фиксирует, что кэш изменен. Пример предполагает, что MDataMapper имеет собственную коллекцию OID (идентификатор объекта).

15.3.3. Альтернативные стратегии Преобразователя данных

В улучшенной итерации 1 используется единственный класс *Преобразователя данных* (MDataMapper — класс «преобразователь данных» пакета mediator), чтобы размещать и удалять строки данных у всех объектов пакета entity (сущность), известных в приложении (EEmployee — класс «служащий» пакета entity, EContact — класс «деловой партнер» пакета entity и EOutMessage — класс «исходящее сообщение» пакета entity).

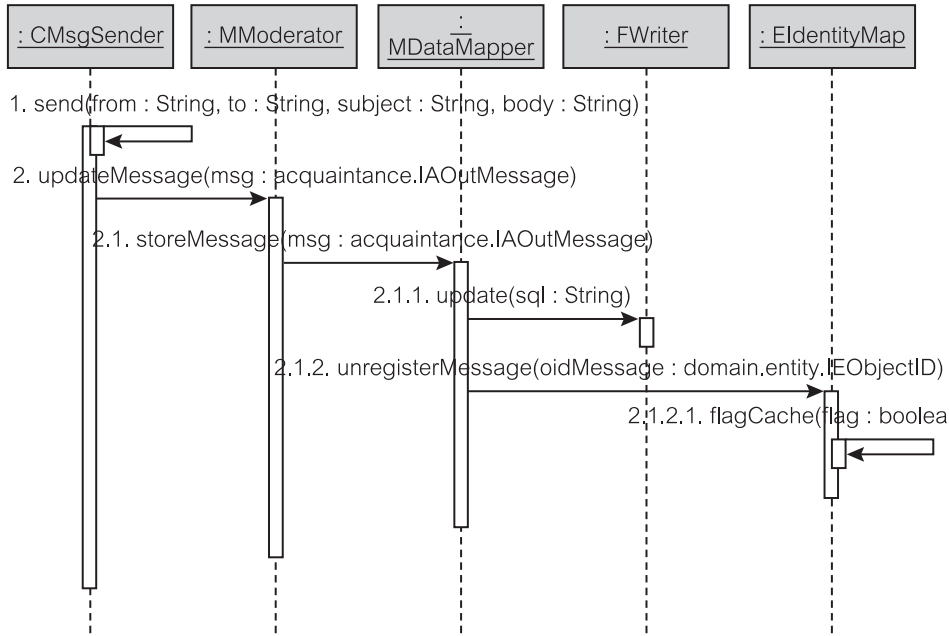


Рис. 15.6. Диаграмма последовательности для операции экспорта класса MDataMapper

`MModerator` (класс «координатор» пакета `mediator`) служит Фасадом (см. 9.3.1) для пакета `mediator` (посредник). `MDataMapper` обеспечивает ассоциации к `EIdentityMap` (класс «коллекция идентичности объектов» пакета `entity`), с одной стороны, и к `FReader` (класс «чтение» пакета `foundation`) и `FWriter` (класс «запись» пакета `foundation`), с другой стороны. Это показано на рис. 15.7.

Более сложные системы или последовательные итерации могут требовать альтернативных решений, основанных на различных паттернах рефакторинга. Следующий список точно определяет общие альтернативные стратегии [55, 31]:

- Несколько *Преобразователей данных*, возможно, по одному на каждый класс пакета `entity` (сущность).
- Использование метаданных и Java-отображения, чтобы динамически формировать в приложении *Преобразователи данных* по мере необходимости.
- Экспорт по требованию, который размещает только данные, необходимые в настоящее время приложению, и, если возможно, создает пустые объекты (модули) для данных, не извлеченных из БД, но которые связаны с уже извлеченными данными.

Несколько Преобразователей данных

`MDataMapper` в улучшенной итерации 1 имеет отдельные методы для управления запросами клиента, направленные на различные объекты пакета `entity`. Это необходимо, потому что имеется много различных способов,

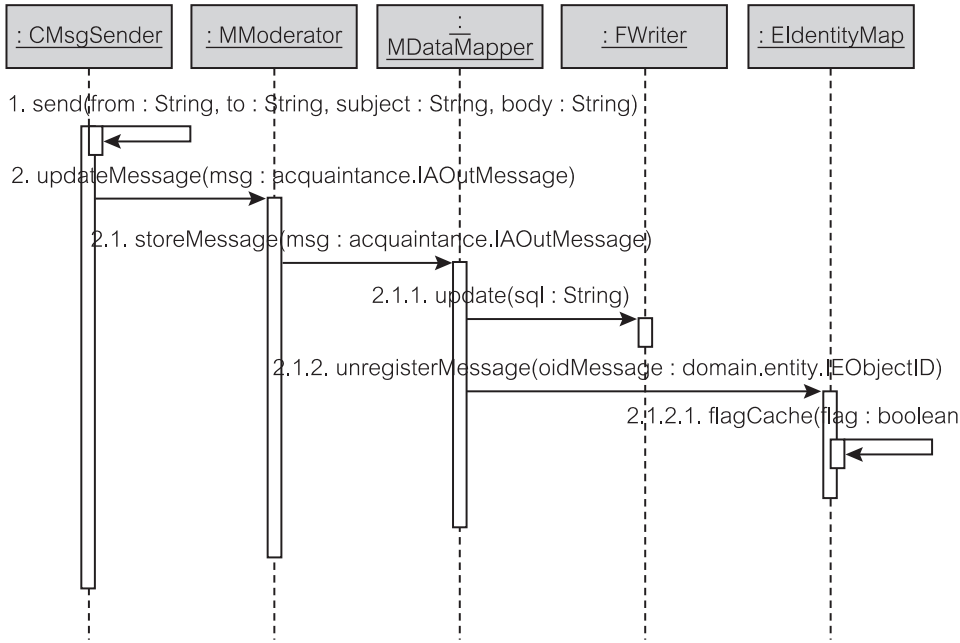


Рис. 15.7. MDataMapper в улучшенной итерации 1

которыми *Преобразователь данных* запрашивает объекты пакета `entity`. Кроме того, каждый объект пакета `entity` имеет свои собственные уникальные элементы данных и ассоциации к другим объектам. В случае большого числа классов пакета `entity`, управляемых приложением, единственный *Преобразователь данных* может стать слишком разросшимся классом.

Рис. 15.8 показывает альтернативное решение итерации 1 с несколькими преобразователями данных — один *Преобразователь данных* на каждый класс пакета `entity`. Три преобразователя реализуют, каждый уникальным способом, интерфейс `IMDataMapper` (интерфейс «преобразователь данных» пакета `mediator`). `MModerator` обеспечивает ассоциацию «один ко многим» к объектам `IMDataMapper` [55].

Назначение методов следующее:

- `getByOID()` — получение объекта пакета `entity`, если дан его `OID` (идентификатор объекта); извлечение его из БД, если кэш изменен.
- `retrieve()` — извлечение объекта пакета `entity` из кэша (если он там) или из БД; если имеет место второй случай, создание нового объекта пакета `entity` и размещение его в кэше.
- `insert()` — преобразование нового объекта пакета `entity` в строку данных и размещение его в БД.
- `update()` — сохранение изменений объекта пакета `entity` в БД.
- `delete()` — удаление объекта пакета `entity` из БД и устранение его из кэша.

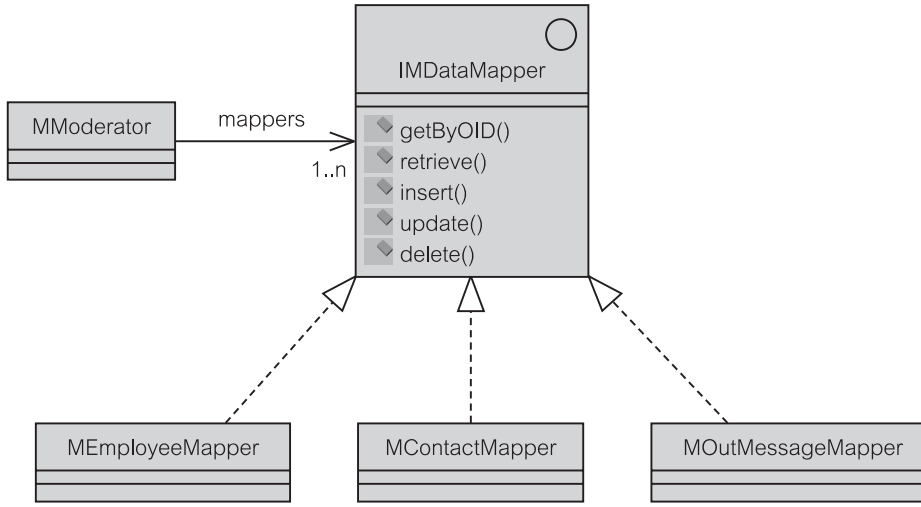


Рис. 15.8. Несколько Преобразователей данных

Обратите внимание, что если для каждого класса пакета `entity` используется отдельный *Преобразователь данных*, то может быть также желательно иметь одну *Коллекцию идентичности объектов* (раздел 15.3.1) на каждый класс пакета `entity`. Тогда каждый преобразователь будет обслуживать свой собственный кэш.

Преобразование метаданных

Цель **паттерна Преобразователь метаданных** [31] — динамически формировать объектно-реляционное преобразование, основанное на **метаданных** (данные относительно данных). Метаданные хранят знания относительно преобразований таблиц и классов, столбцов и элементов данных, внешних ключей и связей ассоциации и т. д. Раздел 10.2.2 описывает виды преобразований, которые могут находиться в метаданных.

Программа может опрашивать метаданные во время выполнения и динамически строить SQL-операторы, с одной стороны, и реализовывать получение/задание объектов пакета `entity`, с другой стороны. С точки зрения масштабируемости *Преобразователь метаданных* позволяет легко адаптировать приложение к изменениям схемы БД и к изменениям, определяемым данными, в частности, к дополнениям классов пакета `entity`.

Важное решение, связанное с *Преобразователем метаданных*, состоит в том, где хранить метаданные. Имеются три возможности [31]:

- непосредственно в *исходном коде* приложения (в пакете `mediator` в случае PCMEF-шаблона);
- во *внешнем файле*, предпочтительно в XML-файле (приложение читает файл во время инициализации и создает соответствующие преобразования в структуре программы);
- в *БД* (это позволяет использовать одни и те же метаданные несколькими приложениями и допускает автоматическую проверку самых последних изменений в схеме БД).

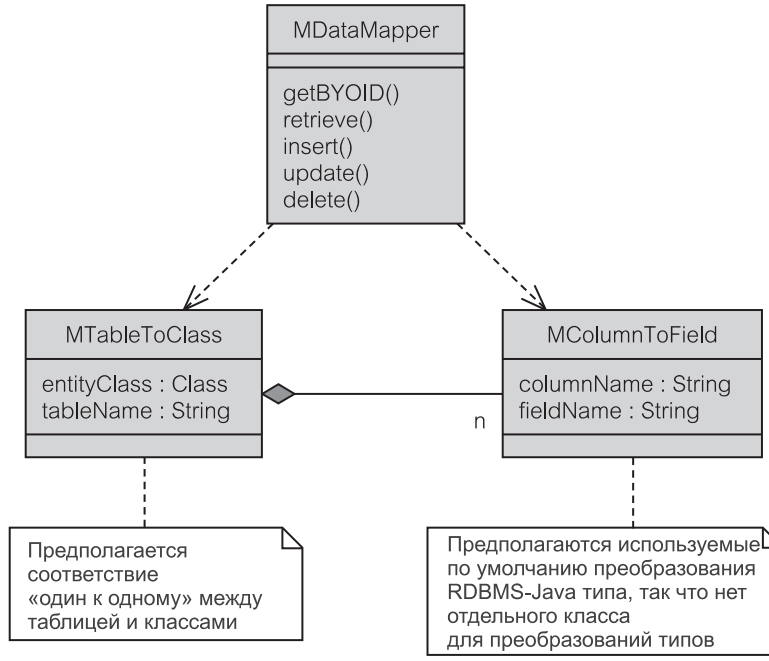


Рис. 15.9. Паттерн Преобразователь метаданных

Имеются две основные стратегии включения информации размещения в выполняемом коде [31]:

- *формирование кода*, который читает информацию метаданных и формирует классы, чтобы выполнить преобразование (это делается во время процесса формирования кода до компиляции программы);
- *рефлексивная программа*, которая позволяет настраивать приложение к любой новой информации метаданных во время выполнения.

В PCMEF классы, которые могли бы формировать *Преобразователь метаданных*, принадлежат пакету `mediator`. Рис. 15.9 представляет простую модель классов для *Преобразователя метаданных*. Однако модель не реализована в учебном примере EM.

`MDataMapper` зависит от этих двух классов метаданных для определения информации преобразования, необходимой для выполнения его методов. Методы, которые извлекают информацию из БД, типа `retrieve()`, будут запрашивать `MTableToClass` (класс «таблицу в класс» пакета `mediator`), чтобы получить `entityClass` (класс пакета `entity`), передающий `tableName` (имя таблицы), из которой извлекаются данные. Методы, которые пишут информацию в БД, типа `insert()`, будут запрашивать `MTableToClass`, чтобы получить `tableName` и `MColumnToField` (класс «столбец в поле» пакета `mediator`) для определения `columnNames` (имена столбцов).

15.3.4. Загрузка по требованию

Записи БД и соответствующие объекты пакета `entity` (сущность) в приложении находятся в тесной взаимосвязи. Запрос загрузить объект пакета `entity` может привести к необходимости загрузить длинный ряд находящихся во взаимосвязи объектов пакета `entity`. Эти взаимосвязанные объекты пакета `entity` соответствуют записям БД, связанным ограничениями ссылочной целостности (то есть отображениям внешних ключей в записи с первичными ключами в других таблицах).

В большинстве случаев приложению нельзя разрешать загрузку всех взаимосвязанных объектов. Во-первых, это может быть невозможно из-за ограниченного размера кэша памяти. Во-вторых, это может быть недопустимо из-за требования параллельной работы с БД, которая должна позволить многим приложениям использовать одни и те же записи бесконфликтным способом. В-третьих, загрузка объектов, не нужных приложению, может отрицательно сказаться на выполнении функций.

Имеются два основных вида операций извлечения, которые целесообразно рассмотреть:

- **Идентифицирующая загрузка** (Identity load) — извлечение конкретных объектов, имеющих заданную величину `OID` (идентификатор объекта) или первичного ключа.
- **Загрузка на основе логического условия** (Predicate load) — извлечение нескольких объектов, выполняя поиск по запросу на основе логических условий, возможно, в соответствии с ограничениями ссылочной целостности.

Имеются три основные стратегии загрузки, которые управляют тем, сколько объектов из результирующего набора, полученного операцией извлечения, будут фактически размещены в кэше:

- **Замкнутая загрузка** (Closure load) — загружаются все объекты, доступные из указанного объекта.
- **Простая загрузка** (Flat load) — загружаются только указанные объекты без объектов компонентов.
- **n-уровневая загрузка** (n-levels load) — загружаются объекты, достижимые из указанного объекта в пределах данного числа уровней.

Как утверждают, замкнутая загрузка, называемая иногда *активной загрузкой*, обычно неприемлема. Остальные стратегии — варианты *загрузки по требованию*. Паттерн **Загрузка по требованию** определяется как «объект, который не содержит все данные, в которых вы нуждаетесь, но знает, как получить их» [31]. Некоторые подходы, реализующие Загрузку по требованию, следующие:

- Инициализация по требованию (Lazy Initialization);
- Виртуальный заместитель (Virtual Proxy);
- Заместитель идентификатора объекта (OID Proxy).

Инициализация по требованию

Паттерн Инициализация по требованию [5] — самый простой вариант паттерна *Загрузка по требованию*. По запросу от объекта-клиента *Преобразова-*

тель данных, который является ответственным за поддержание кэша пакета `entity`, ищет кэш данных и, если данных там нет, загружает их из БД.

Реализация *Инициализации по требованию* отличается в зависимости от ряда факторов. Попытка *Преобразователя данных* получить данные из кэша может быть выполнена по значению элемента данных в объекте пакета `entity`. Если значение `null` — пустое (и `null` недопустим согласно бизнес-правилам), то *Преобразователь данных* должен инициализировать элемент данных, обращаясь к БД. Но здесь может быть несколько сложностей.

Элемент данных может быть простого типа данных, ссылкой на класс или коллекцией ссылок на классы. Каждый случай требует различных механизмов реализации. Кроме того, значение `null` может быть законной величиной, а не заместителем отсутствующего объекта, который должен быть загружен. Наконец, структура зависимости принятого структурного шаблона может не позволить объектам пакета `entity` зависеть от объектов `foundation` (основание), чтобы получить данные из БД (PCMEF используют `MDataMapper` — класс «преобразователь данных» пакета `mediator`, чтобы решить эту задачу, но все еще есть проблема зависимости между пакетами `entity` и `mediator` — посредник).

Листинг 15.2 демонстрирует, как *Инициализация по требованию* может использоваться в `EOutMessage` (класс «исходящее сообщение» пакета `entity`), в предположении, что приложение содержит преобразователь данных, в которых имеется `contactOID` (OID делового партнера) к загруженному объекту `EContact` (класс «деловой партнер» пакета `entity`), и элемент данных, содержащий `contactID` (идентификатор делового партнера), который соответствует внешнему ключу в БД. Когда `EOutMessage` запрашивает `getContact()` (получить делового партнера), тот проверяет, является ли `contactOID` пустым. Если это так, то `EContact` не загружается. Чтобы загрузить его, `EOutMessage` посылает сообщение `retrieveContact()` (извлечь делового партнера) объекту `MDataMapper`.

Листинг 15.2. Метод getContact() в EOutMessage

```
Метод getContact() в EOutMessage

public IAContact getContact()
     { if (contactOID == null)
        contact = MDataMapper.retrieveContact(contactID);
        return contact;
}
```

Виртуальный заместитель

Паттерн Виртуальный заместитель [32] — более мощная альтернатива *Инициализации по требованию*. Заместитель заменяет объект и выступает в роли реального объекта. Он получает сообщения, предназначенные для реального объекта (который называется реальным субъектом в *Виртуальном заместителе*) и создает реальный объект, если он еще не существует. В *Загрузке по требованию* заместитель обозначает объект, который может потребоваться загрузить при попытке обратиться к нему.

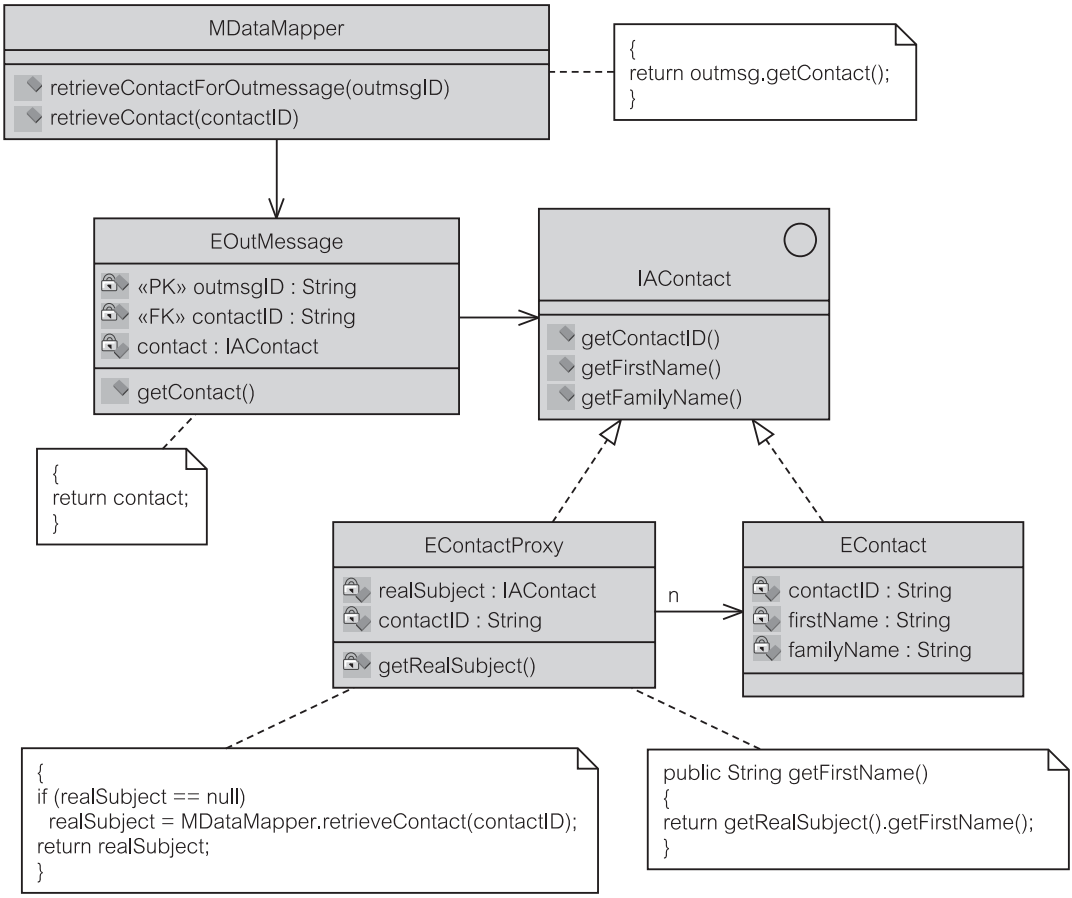


Рис. 15.10. Паттерн Виртуальный заместитель

Рисунки 15.10 и 15.11 показывают, как *Виртуальный заместитель* может использоваться в контексте кода итерации 1. Объект `MDataMapper` ответствен за кэш пакета `entity`. В учебном примере EM объекты пакета `entity` различаются идентификаторами, размещенными в приложении. Чтобы проиллюстрировать *Виртуальный заместитель*, предположим, что объекты пакета `entity` идентифицированы своими первичными ключами (`PK`). Предположим, что объект `EOutMessage` должен быть загружен и что `EContactProxy` (класс «заместитель делового партнера» пакета `entity`) знает `PK` (первичный ключ) своего реального субъекта. Когда `MDataMapper` создает `EContactProxy`, он инициализирует его поле `contactID` соответствующим значением внешнего ключа (`FK`) в `EOutMessage`.

Как видно из рис. 15.10, `EOutMessage` обеспечивает ассоциацию с деловым партнером. Ассоциация изображена с интерфейсом `IAContact` (интерфейс «деловой партнер» пакета `acquaintance`). В действительности же связь имеется с `EContactProxy`. Когда `MDataMapper` запрашивает делового партнера (`getContact()`), `EOutMessage` возвращает `EContactProxy`.

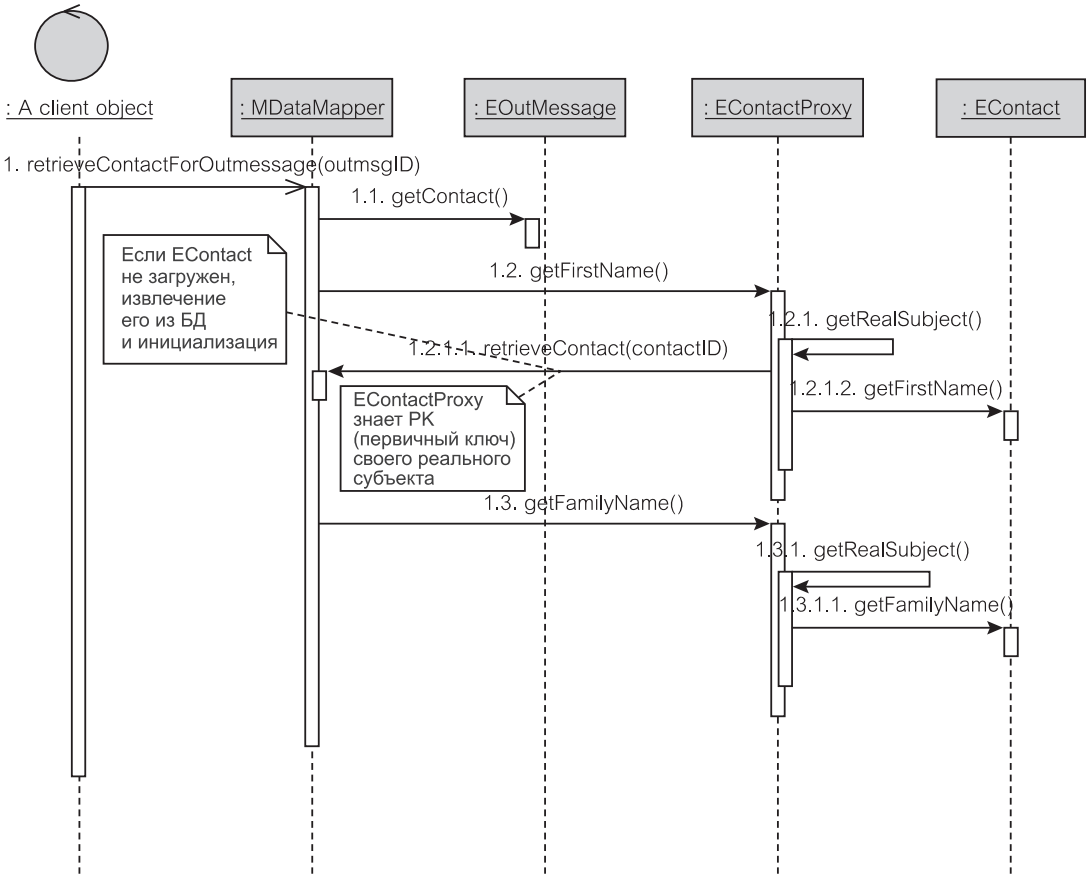


Рис. 15.11. Виртуальный заместитель — диаграмма последовательности для retrieveContactForOutmessage() (извлечение делового партнера для исходящего сообщения)

Когда на следующем шаге `MDataMapper` запрашивает, например, `firstName` (имя) делового партнера (рис. 15.11), `EContactProxy` проверяет, существует ли реальный субъект (`EContact`). Если его нет, запрашивается `MDataMapper`, чтобы инициализировать его. Затем `EContactProxy` может возвратить `firstName`. Следующий запрос относительно `familyName` (фамилия), выполняется подобным же образом за исключением того, что `EContact` уже загружен.

Интерфейс пакета `entity`, реализованный с помощью `MDataMapper` и используемый заместителями, устранил бы цикл между `MDataMapper` и `EContactProxy` (раздел 9.1.7). В общем случае может быть несколько заместителей для одного и того же реального субъекта. Организация того, что все эти заместители используют один и тот же интерфейс, даст дополнительные преимущества.

Заместитель идентификатора объекта

Паттерн *Виртуальный заместитель* — популярный способ реализовать *Загрузку по требованию*, но он потенциально избыточен в системах, которые назначают идентификаторы объектов (OID) объектам пакета entity при их загрузке. Среды программирования объектов имеют внутренние механизмы, гарантирующие идентификацию объектов, загружаемых в память, и используют эти механизмы для связи объектов. Механизмы различны для разных языков.

В Java этот механизм скрыт от программиста. Связь между объектами представляется программисту как включение. Например, если EOutMessage имеет поле contact (деловой партнер), то программист может использовать это поле для доступа к связанному объекту EContact, как будто бы EContact физически содержался в EOutMessage. В C++ поле contact было бы указателем в памяти на EContact.

Ни один из этих механизмов не является заменой идентификаторов объектов (OID), явно назначаемых программой объектам пакета entity, когда они инициализируются в памяти. Во всех тривиальных ситуациях сложные передачи объектов пакета entity между приложением и БД требуют от программы явного назначения OID для объектов, размещаемых в памяти. Это обеспечивает дублирование между внутренними механизмами сред программирования, обеспечивающими идентификацию объектов, и явным заданием OID.

Дублирование не всегда плохо. Тот факт, что обе технологии связывают объекты в памяти, могут быть преимуществом в *Загрузке по требованию*. Результатом является то, что *Загрузка по требованию* может быть полностью реализована с помощью замено-подобного окольного пути без необходимости использования отдельных замещающих классов. Одноэлементный класс, который обеспечивает задание OID объектам (называемый EIdentityMap — класс «коллекция идентичности объектов» пакета entity — в улучшенном коде итерации 1) выполняет функции замещающих классов. Такой подход может быть зарегистрирован в паттерне — **паттерне Заместитель идентификатора объекта**.

EIdentityMap знает, загружен ли объект пакета entity. После загрузки объекту задается OID, и все его элементы данных инициализируются. Инициализация включает значения внешних ключей (FK), полученные из БД. Основанные на OID связи ассоциации, соответствующие внешним ключам, инициализируются значением null, если связанный объект не был еще загружен.

Дополнительное преимущество *Заместителя идентификатора объекта* над *Виртуальным заместителем* — легкость, с которой может быть определено измененное/не измененное состояние объекта пакета entity. Для объекта пакета entity, загруженного первым, устанавливается флаг неизмененного объекта. Если содержание данных объекта выходит из синхронизма со своим соответствующим содержанием в БД, flag (флаг) устанавливается в состояние «измененный». Объект пакета entity всегда знает свое состояние и может вызывать перезагрузку из БД, чтобы обновить содержание.

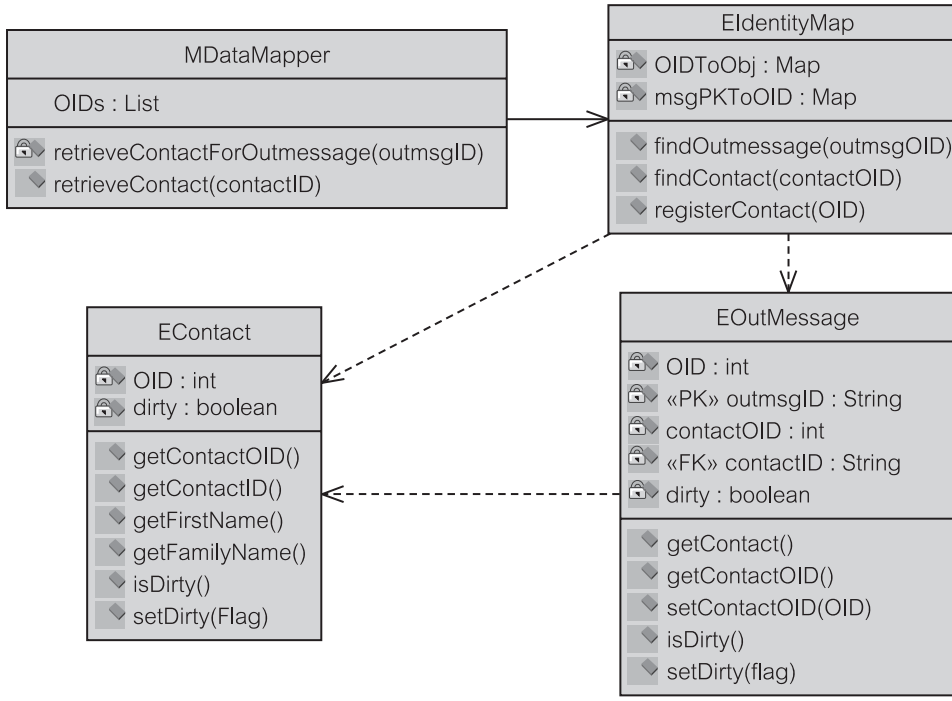


Рис. 15.12. Паттерн Заместитель идентификатора объекта — навигация по коллекции идентичности

Имеются две разновидности *Заместителя идентификатора объекта*, отличающиеся тем, как программа обеспечивает навигацию между связанными объектами пакета `entity`:

- навигация по коллекции идентичности объектов;
- навигация по классам пакета `entity`.

Навигация по коллекции идентичности объектов

Коллекция идентичности объектов содержит каждый загруженный объект пакета `entity`. Это размещение поддерживает связь между `OID` и его объектом. Другие коллекции, которые связывают значения первичных ключей с `OID`, также могут поддерживаться, чтобы обеспечивать запросы на поиск объектов, основанные на значениях их полей. Эти коллекции, в общем, достаточны, чтобы обеспечить навигацию по объектам, которые связаны в БД ограничениями ссылочной целостности.

Навигация по коллекции идентичности объектов требует, чтобы *Коллекция идентичности объектов* опросила объекты пакета `entity` для установления связи ассоциации. Связь ассоциации в объекте пакета `entity` представляется двумя способами:

- значением внешнего ключа (`FK`), полученным из БД;
- значением `OID` связанного объекта в памяти.

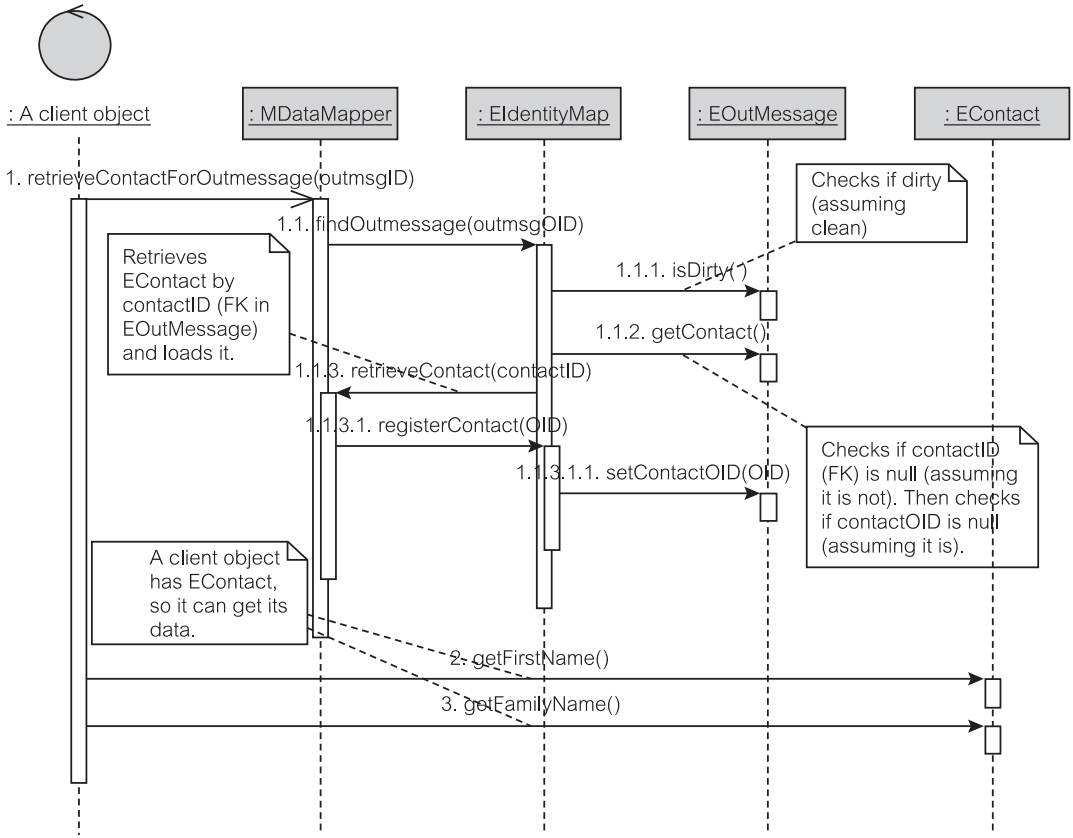


Рис. 15.13. Заместитель идентификатора объекта: навигация по коллекции идентичности объектов — диаграмма последовательности для retrieveContactForOutmessage()

И `FK`, и `OID` элементов данных могут иметь значение `null`. Значение `null` внешнего ключа (`FK`) имеет тот же самый смысл, что и в БД — объект не связан. Значение `null OID` означает, что связанный объект не загружен.

Рис. 15.12 показывает модель класса, которая использует *Заместителя идентификатора объекта* вместо *Виртуального заместителя*. Класс `EOutMessage` содержит элемент данных `contactOID`, который служит заместителем объекта `EContact`, с которым он связан. Как и у паттерна *Загрузка по требованию*, объект `EOutMessage` не содержит данные, необходимые, когда вызывается `getContact()`, но он знает, как получить их.

`MDataMapper` содержит список `OID` всех объектов, загруженных в памяти. Когда передается `OID` какого-либо объекта, по нему можно сразу же проверить, загружен ли такой объект. Когда передается значение первичного ключа (`PK`) объекта (например, `outmsgID` — идентификатор исходящего сообщения), может быть запрошен `EIdentityMap` о соответствующем `OID`.

Рис. 15.13 объясняет последовательность сообщений для того же сценария, что и на рис. 15.11. Вместо *Коллекции идентичности объектов* каждый объект пакета `entity` знает свое состояние (изменен он или нет). Состояние проверя-

ется методом isDirty() (изменен ли). Предполагая, что EOutMessage не является измененным, EIdentityMap запрашивает EOutMessage выполнить метод getContact(). EOutMessage проверяет, является ли значение внешнего ключа к EContact пустым (этого не должно быть, потому что в итерации 1 запрещается в БД значение contactID, равное null). Затем он проверяет, является ли contactOID равным null.

Объект EContact может быть, а может и не быть загружен в память. EOutMessage содержит знание об этом в contactOID. Если contactOID равен null, то EContact не был загружен. Диаграмма последовательности действий на рис. 15.13 предполагает, что дело обстоит именно так. EIdentityMap запрашивает MDataMapper, чтобы тот извлек EContact на основе значения внешнего ключа.

Результатом будет то, что EContact загрузится (по требованию), MDataMapper корректирует его список OID, EIdentityMap корректирует его коллекции, а contactOID устанавливается в EOutMessage. EContact теперь загружен и возвращается к client object (объект-клиент), который может теперь непосредственно запросить его относительно данных (getFirstName() — получение имени, getFamilyName() — получение фамилии и т. д.).

EIdentityMap отвечает за все навигации между объектами пакета entity. При этом подходе объекты пакета entity не должны содержать связанные объекты. Объекты пакета entity содержатся в коллекциях EIdentityMap. Объекты пакета entity непосредственно имеют только OID к другим объектам.

Навигация по классам пакета entity

Навигация по *Коллекции идентичности объектов* затеняет объектно-реляционное отображение. Ограничения ссылочной целостности видимы в объектах пакета entity, и они подкреплены (дублированы) значениями OID к связанным объектам. Однако объект пакета entity не может непосредственно осуществлять навигацию по этим связям. *Коллекция идентичности объектов* — единственное место, которое знает, как получить объект по его OID. В системе с большим количеством классов пакета entity *Коллекция идентичности объектов* делает слишком много.

Обязанность осуществлять навигацию между объектами пакета entity может быть поручена этим объектам. **Навигация по классам пакета entity** требует, чтобы объект пакета entity не содержал лишь OID к другому объекту, но также содержал и этот объект (подобно примеру *Виртуального заместителя*, представленного на рис. 15.10).

Рис. 15.14 представляет диаграмму последовательности действий, соответствующую рис. 15.13, но которая использует Навигацию по классам пакета entity. Многое из логики обработки findContactForOutmessage() (найти делового партнера для исходящего сообщения) перемещено от MDataMapper к EOutMessage.

После определения, что EOutMessage не изменен, MDataMapper сообщает это методу getContact(). Если EContact не загружен, EOutMessage инициирует загрузку. Он передает свой outmsgOID (идентификатор

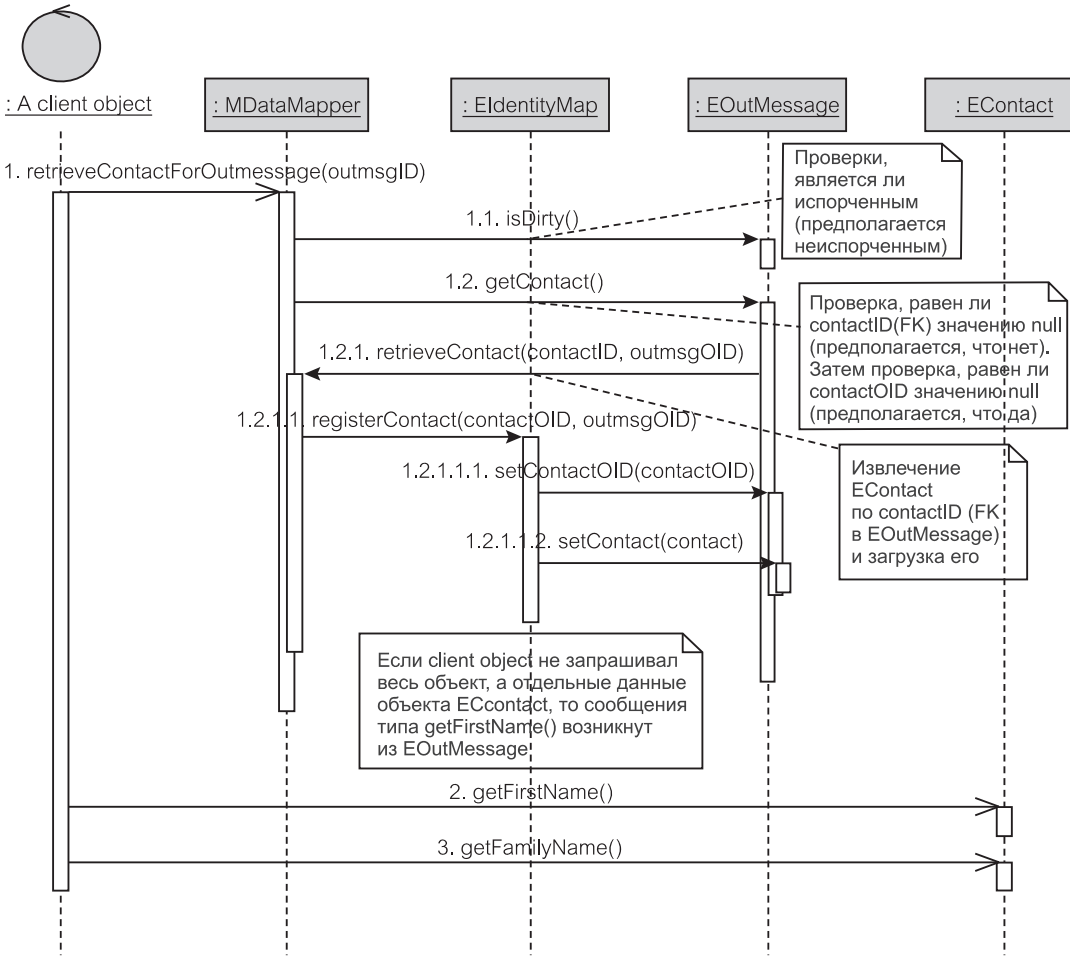


Рис. 15.14. Заместитель идентификатора объекта: навигация по классам пакета entity — диаграмма последовательности для retrieveContactForOutmessage()

фикатор исходящего сообщения) объекту `MDataMapper`, чтобы новый `EContact` мог быть связан с ним, когда будет зарегистрирован в `EIdentityMap`.

В сценарии, где `client object` запрашивает `MDataMapper` для всего `EContact`, `client object` может непосредственно получить данные от `EContact` (`getFirstName()`, `getFamilyName()` и т. д.). Однако если бы `EContact` был загружен ранее, `EOutMessage` получил бы данные от `EContact` (он также проверил бы, что `EContact` не был изменен перед получением данных).

Навигация по классам пакета `entity` обеспечивает более интуитивную и более изящную реализацию паттерна *Заместитель идентификатора объекта*. Семантика отношений между сохраняемыми бизнес-объектами реализуется непосредственно в этих объектах. Навигация по коллекции идентичнос-

ти объектов требует продвижения по объектам пакета entity вперед и назад, чтобы найти данные в связанных объектах. Навигация по классам пакета entity позволяет реализовать линейное продвижение по объектам этого пакета. *Коллекция идентичности объектов* необходима лишь для того, чтобы найти начальный объект, от которого может продолжаться дальнейшая навигация.

15.3.5. Единица работы

Объекты пакета entity (сущность) должны быть загружены (импортированы) из БД по двум причинам. Во-первых, потому что приложение нуждается в них, и они не были загружены. Во-вторых, потому что приложение находит, что они стали измененными (то есть противоречивыми по отношению к состоянию БД).

В итерации 1 единственные объекты, которые могли быть помечены как измененные, это объекты после того, как они были переданы по электронной почте — объекты EOutMessage (класс «исходящее сообщение» пакета entity). Итерация 1 не делает никаких изменений объектов EEmployee (класс «служащий» пакета entity) и EContact (класс «деловой партнер» пакета entity). Для упрощения итерация 1 объявляет весь кэш (все объекты пакета entity) измененным, как только будет передано по электронной почте любое EOutMessage (раздел 13.6.1, листинг 13.56). Флаг измененного состояния находится в классе MBroker.

Улучшенный код итерации 1 обеспечивает более гибкое управление кэшем. Одним из усовершенствований является представление флага измененного состояния в каждом объекте пакета entity. При загрузке флаг объекта очищается. Флаг устанавливается измененным, когда объект изменится. Это проиллюстрировано в листинге 15.3. После того как сообщение EOutMessage будет передано по электронной почте, MModerator (класс «координатор» пакета mediator) посылает сообщение setDirty() (установить измененным) измененному объекту EOutMessage (строка 99).

Листинг 15.3. Метод updateMessage() в MModerator

Метод updateMessage() в MModerator

```
87:     public boolean updateMessage(IAOutMessage msg) {
91:        msg.setSentDate(new java.sql.Date(System.
                                    currentTimeMillis()));
92:
93:        boolean b = mapper.storeMessage(msg);
94:        if (b) {
96:           msg.setContact(null);
97:           msg.setCreatorEmployee(null);
98:           msg.setSenderEmployee(null);
99:           msg.setDirty(true);
100:       }
```

Подход, где каждый объект пакета `entity` знает, является ли он измененным или нет, позволяет осуществить индивидуальную загрузку и выгрузку объектов. Однако многие бизнес-транзакции, управляемые приложением, делают многочисленные изменения у разнообразных объектов в кэше перед попыткой записать эти изменения в БД. Измененные объекты должны быть соответственно помечены как измененные, когда БД подтверждает, что транзакция успешно завершена. Эти проблемы быстро становятся очень сложными.

Паттерн Единица работы «обслуживает список объектов, на которые воздействует бизнес-транзакция, и координирует перезапись изменений и решение проблем параллелизма» [31]. Изменения могут быть следствием трех операций модификации: размещения нового объекта пакета `entity` в БД, удаления объекта из БД или обновления элементов данных объекта.

Паттерн *Единица работы* требует создания нового класса в пакете `mediator` (посредник). Класс можно назвать `MWorkUnit` (единица работы) или другим подобным образом. *Единица работы* представлена в итерации 2, но ее полная применимость демонстрируется в итерации 3 (глава 21).

15.4. Улучшенная модель классов

Улучшенный код итерации 1 состоит из увеличенного числа классов по сравнению с первоначальным проектом (раздел 13.1, рис. 13.1). Новые классы улучшают единство проекта, не влияя при этом на связанность проекта. Зависимости между классами в улучшенном коде соответствуют PCMEF-шаблону.

Улучшенная модель классов представлена здесь в трех диаграммах классов. Первая диаграмма (рис. 15.15) представляет классы на уровнях `presentation` (представление) и `control` (управление). Раздел 15.2 объясняет рефакторинги, которые приводят к модели, представленной на рис. 15.15.

Вторая диаграмма для улучшенной модели (рис. 15.16) концентрируется на уровне `domain` (предметная область). Примечания в диаграмме дают основные пояснения. Детали паттернов рефакторинга, используемых на уровне `domain`, обсуждены в разделе 15.3. Использование интерфейсов Java-коллекций в моделировании ассоциаций таково, как было обсуждено в разделе 12.1.2.

Уровень `foundation` (основание) — единственный PCMEF-уровень, на который не воздействует рефакторинг итерации 1. Однако зависимости от `mediator` к `foundation` слегка изменяются из-за замены класса `MBroker` (класс «посредник» пакета `mediator`) двумя классами: `MModerator` (класс «координатор» пакета `mediator`) и `MDataMapper` (класс «преобразователь данных» пакета `mediator`). Это показано на рис. 15.17.

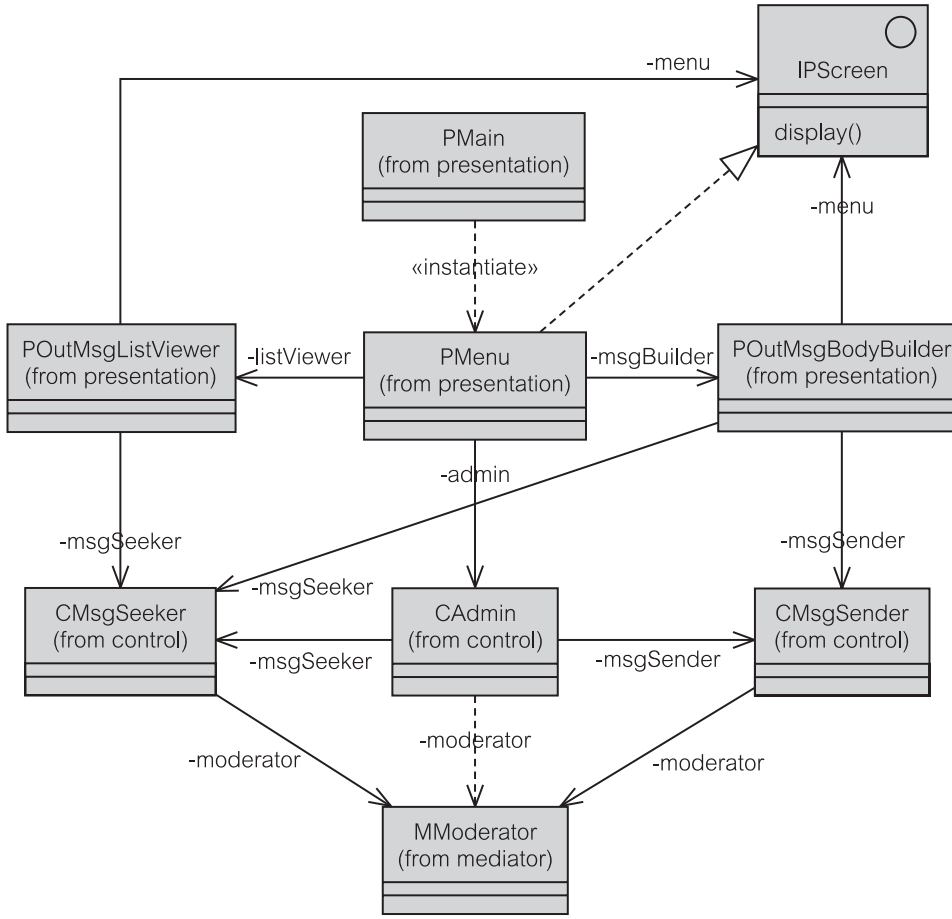


Рис. 15.15. Уровни presentation и control в улучшенной итерации 1

Резюме

1. *Рефакторинг* — процесс чистки и улучшения внутренней структуры кода без изменения его внешнего поведения.
2. Есть две основные *цели рефакторинга*: устранение «дурного запаха в коде» (очистка) и структурные усовершенствования кода, приводящие к лучшей его структуре.
3. *Методы рефакторинга* (или просто *рефакторинги*) — основные принципы и лучшая практика изменения кода для улучшения возможности его сопровождения (понятность, удобство сопровождения и масштабируемость).
4. Рефакторинг *Класс извлечения* расщепляет большой класс на ряд меньших классов.

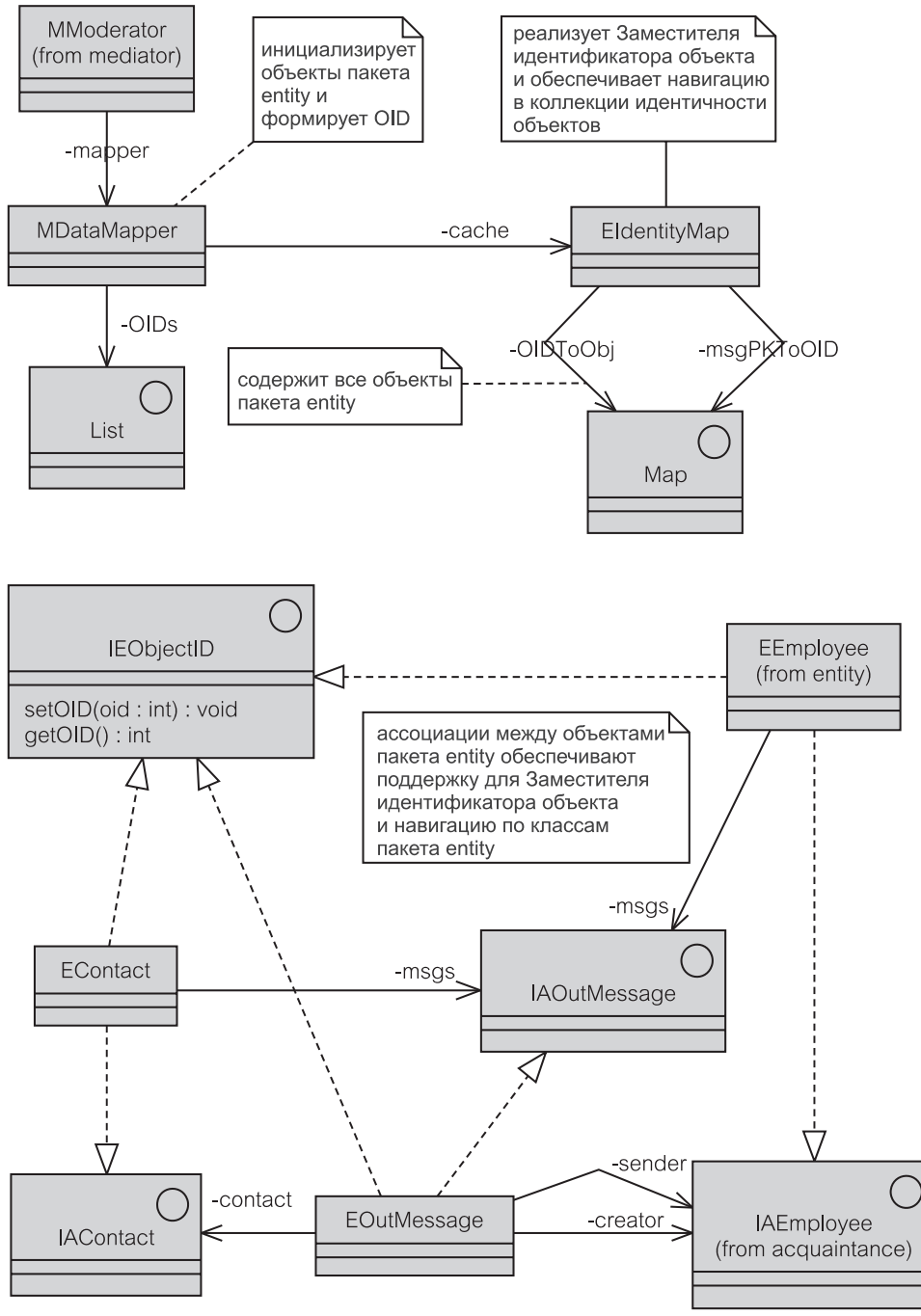


Рис. 15.16. Уровень domain в улучшенной итерации 1

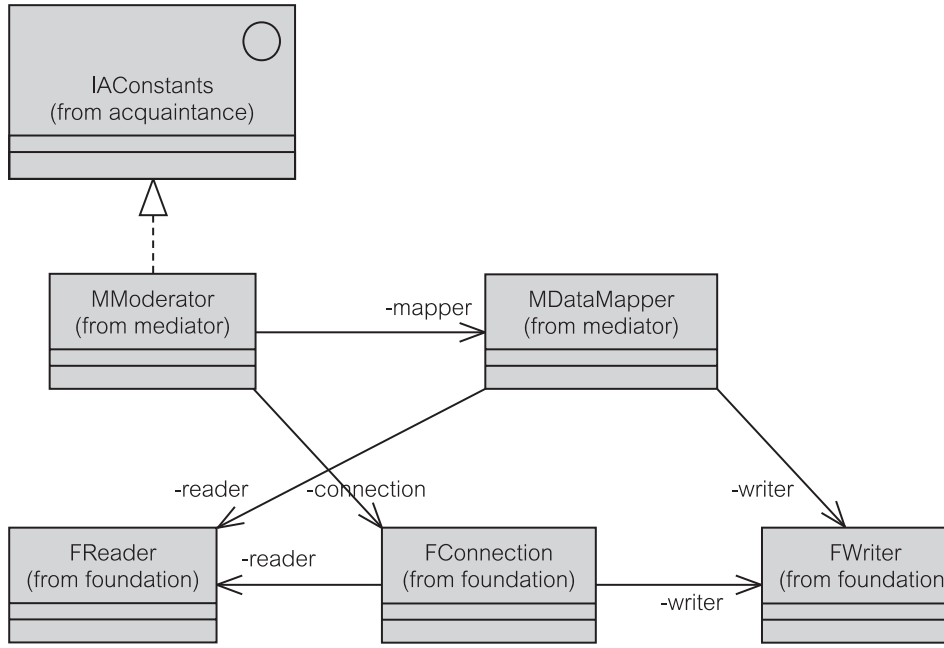


Рис. 15.17. Пакеты mediator и foundation в улучшенной итерации 1

5. Рефакторинг *Метод извлечения* преобразует дублированный код в отдельный метод.
6. Рефакторинг *Метод подключения* устраняет метод включением его функциональных возможностей в другой существующий метод.
7. Рефакторинг *Интерфейс извлечения* выделяет множество сигнатур методов, дублированных в нескольких классах или используемых несколькими клиентами, в интерфейс.
8. *Паттерны рефакторинга* — структурные паттерны, используемые в рефакторинге кода.
9. Паттерн *Коллекция идентичности объектов* назначает идентификаторы объектам и поддерживает коллекции, чтобы найти объекты, находящиеся в памяти, на основе идентификаторов объектов.
10. Паттерн *Преобразователь данных* отделяет объекты пакета `entity` (сущность), находящиеся в памяти, от их постоянного представления в БД и отвечает за поддержание кэшей памяти объектов этого пакета.
11. UML-диаграммы деятельности служат технологией спецификации поведения, способной изобразить управление и потоки данных.
12. *Загрузка* (импорт, материализация) — процесс извлечения записей из БД и преобразования их в объекты памяти. Противоположная операция называется *выгрузкой* (экспортом, пассивацией, дематериализацией).

13. Паттерн *Загрузка по требованию* загружает только отобранные объекты из БД в память, но он может загружать оставшиеся и связанные объекты, когда это необходимо. Подходами к *Загрузке по требованию* являются Инициализация по требованию, Виртуальный заместитель и Заместитель идентификатора объекта.
14. Паттерн *Инициализация по требованию* загружает объекты по определенному запросу от объекта-клиента, ответственного за поддержание кэша пакета `entity`. Загрузка для клиентов не прозрачна.
15. Паттерн *Виртуальный заместитель* использует объект-заместитель (заместитель), который является заменой реального объекта и может загрузить реальный объект способом, прозрачным для клиента.
16. Паттерн *Заместитель идентификатора объекта* — удобная замена паттерна *Виртуальный заместитель* в приложениях, которые поддерживают коллекции OID объектов. Коллекции могут использоваться вместо классов-заместителей, чтобы загрузить объекты в память способом, прозрачным для клиента. Имеются две разновидности Заместителя идентификатора объекта, которые отличаются тем, как программа реализует навигацию между связанными объектами пакета `entity`: Навигация по коллекции идентичности объектов и Навигация по классам пакета `entity`.
17. *Навигация по коллекции идентичности объектов* использует класс Коллекция идентичности объектов каждый раз, когда приложение должно осуществить навигацию к связанному объекту пакета `entity` (то есть связанному ссылочной целостностью). Если объект `X` связан с объектом `Y`, Коллекция идентичности объектов должна запросить `X`, чтобы получить связь с `Y`, и затем осуществляет доступ к `Y`.
18. *Навигация по классам пакета entity* использует класс Коллекция идентичности объектов, чтобы получить первый объект пакета `entity`, но затем использует классы пакета `entity`, чтобы осуществить навигацию по связанным классам. Если объект `X` связан с объектом `Y`, Коллекция идентичности объектов позволила бы клиенту осуществить доступ к `X`, но затем передает управление к `X`, чтобы продолжить навигацию к `Y`.
19. Паттерн *Единица работы* обеспечивает приложение знанием проблем о бизнес-транзакциях и параллелизме. Он хранит последовательность изменений объектов пакета `entity` и то, действительно ли изменения были переданы в БД.

Ключевые термины

Обзорные вопросы

Вопросы для обсуждения

1. Сравните применимость рефакторинга в быстрой разработке и в Unified Process.
2. Сравните цели рефакторинга *Класс извлечения* и *Интерфейс извлечения*. Используйте примеры, отличающиеся от примеров в тексте, чтобы показать сходства и различия.
3. Рефакторинг *Метод подключения* используется нечасто. Почему? Сравните его с *Методом извлечения*.
4. Объясните характеристики паттерна *Коллекция идентичности объектов*. Как Коллекция идентичности объектов связана с объектно-реляционным преобразованием?
5. Объекты характеризуются как имеющие состояние, поведение и идентификацию. Обсудите, как различные виды коллекции идентичности объектов поддерживают понятие идентификации объекта (OID). Обсудите четыре вида коллекций идентичности объектов: явно заданная коллекция идентичности объектов, общая коллекция идентичности объектов, одна коллекция на сеанс и одна коллекция на класс.
6. Обсудите преимущества и недостатки стратегии преобразования данных, основанной на многих преобразователях данных.
7. Паттерн *Преобразователь метаданных* не определяет, где метаданные должны храниться (размещаться). Каковы здесь возможности? Как их сравнить?
8. Приведите пример, объясняющий три варианта загрузки по требованию: замкнутая загрузка, простая загрузка и n-уровневая загрузка.
9. Сравните паттерны *Виртуальный заместитель* и *Заместитель идентификатора объекта*.
10. Сравните *Навигацию по коллекции идентичности объектов* и *Навигацию по классам пакета entity*.

Вопросы учебного примера

1. Рассмотрите рефакторинг *Класс извлечения* на рис. 15.1. Обратите внимание на метод `setEmployee()` (задать информацию о служащем). Что могло бы быть возможным прототипом этого метода? (Вспомните, что прототип метода — это несколько больше, чем сигнатура метода, поскольку включает возвращаемый тип.)
2. Рассмотрите рефакторинг *Класс извлечения* на рис. 15.1 и рефакторинг *Метод подключения* на рис. 15.2. Обратите внимание на метод `retrieveMessages()` (извлечь сообщения). Что могло бы быть возможным прототипом этого метода?
3. Что необходимо сделать на уровне `control` (управление), если мы не хотим, чтобы многие из классов уровня `control` подвергались воздействию уровня `presentation` (представление)?
4. Предположим, что EM-приложение позволяет пользователю фильтровать исходящие сообщения. Это означает, что пользователь может применять критерии фильтра, чтобы показать только отобранные подмножества исходящих сообщений, например, исходящие сообщения, намеченные для передачи по электронной почте в будущем, исходящие сообщения, намеченные для конкретного отдела, и т. д. Как мы могли бы смоделировать это?

Примеры задач

1. Обратимся к рефакторингу *Интерфейс извлечения* на рис. 15.3. Могли бы вы улучшить код по-другому? Покажите новую модель.
2. Обратимся к вопросу 4 учебного примера, приведенному выше. Предположим, что используется объект `MessageFilter` (фильтр сообщений), чтобы реализовать фильтрацию исходящих сообщений приложением, и который передает объект соответствующим классам уровня `domain` (предметная область). Изобразите фрагмент диаграммы последовательности действий, который модифицирует модель для взаимодействия «Просмотр непосланных сообщений» (рис. 11.12), чтобы включить фильтрацию объектом `MessageFilter`.
3. Большинство EM-операций инициализируется уровнем `control` (управление) и делегируется пакету `mediator` (посредник). Пакет `mediator` далее управляет объектами пакета `entity` (сущность), которые он считает пригодными. Мы можем смоделировать это по-другому, используя паттерн Издания-подписки на объекты пакета `entity`. Когда объект пакета `entity` изменяется, он уведомляет своих подписчиков относительно этого факта. Представьте диаграмму последовательности действий для этого случая.
4. Как показано на рис. 15.4, `MDataMapper` (класс «преобразователь данных» пакета `mediator`) ответствен за нахождение объектов пакета `entity` и, если необходимо, загрузку их в память. Приведите альтернативный проект, в котором `MModerator` (класс «координатор» пакета `mediator`) играет более важную роль в процессе. Какие методы рефакторинга и паттерны, если таковые вообще будут, обеспечат эту модификацию?

Глава 16

Проектирование и программирование пользовательского интерфейса

Основным «шагом» итерации 2 после итерации 1 учебного примера ЕМ является замена текстового консольного интерфейса графическим и основанным на Web-технологии интерфейсом. Итерация 1 сосредоточена на введении системы надежного структурного проектирования. Это проектирование определяет внутреннюю структуру и внутренние операции системы. Однако в ПО ценятся **вид и поведение**, а не его внутренняя организация. Итерация 2 концентрируется на придании подсистеме ЕМ надлежащего впечатления и ощущения от ее использования.

Как видно из вводных комментариев к главе 9, структурное проектирование имеет аналогии в строительной индустрии. Дом не может быть построен, если его сначала не спроектировал архитектор. Дом не может быть продан, если покупателю не понравятся его вид и комфортность. Усилия по проектированию и программированию **пользовательского интерфейса** (user interface — UI) нацелены на создание привлекательного и полезного впечатления и ощущения от использования системы путем определения объектов пользовательского интерфейса и действий, которые позволяют пользователю реализовывать функции системы.

Промышленные приложения неизменно являются **клиент/серверными** (client/server — C/S) решениями. Часто они являются **многоярусными** решениями. Ярусы определяют ряд уровней серверов, типа Web-сервера, сервера приложения и сервера БД. Клиентом может быть рабочий стол, портативный компьютер, карманный компьютер, мобильный телефон, терминал ввода/вывода и т. д. Приложение может поддерживать многих клиентов.

Один из способов классификации клиентов состоит в разделении их на программируемых клиентов и клиентов-браузеров [94]. **Программируемый клиент** предполагает, что программа находится и выполняется на клиенте и имеет доступ к ресурсам, хранящимся на клиенте. Также важно, что программируемый клиент может загружать данные с сервера, отключаться от источника данных, если это потребуется, локально кэшировать данные, обрабатывать их и передавать их своему пользовательскому интерфейсу. Программируемый клиент иногда называется **толстым клиентом** или **богатым клиентом**.

Клиент-браузер нуждается в сервере, чтобы загрузить требуемые данные и получить инструкции для предоставления данных в пользовательском интерфейсе на основе Web-технологии. Кроме простых проверок пользова-

тельского входа клиент-браузер не имеет возможности своей собственной обработки. Данные представляются как Web-страницы, отформатированные на языке разметки гипертекстов (HyperText Markup Language — HTML). Клиент-браузер также называется **тонким клиентом**, **Web-клиентом** или **HTML-клиентом**.

Приложение типа учебного примера ЕМ может быть развернуто:

- *локально* с программируемым клиентом (клиент Java в случае ЕМ);
- на **Web-сервере** с клиентом-браузером и обращением с помощью сервлетов и серверных страниц на Java (Java Server Pages — JSP);
- на **сервере приложения** типа сервера Enterprise JavaBeans (EJB) с программируемым клиентом или клиентом-браузером.

Учебный пример ЕМ предполагает, что программируемый клиент Java развернут локально. Однако чтобы объяснить клиенты-браузеры и различные опции развертывания, части итераций 2 и 3 ЕМ-решения были перепрограммированы. В этой главе обсуждается проектирование и программирование пользовательского интерфейса для клиента Java-приложения, который использует набор базовых классов Java (Java Foundation Classes — JFC) с библиотекой Swing API. Она также представляет пользовательский интерфейс для приложения, разработанного с использованием бизнес-компонент, и такой, у которого бизнес-логика «вытекает» из исходного проекта БД. Такое приложение обычно разворачивается на сервере приложения.

16.1. Основные принципы проектирования пользовательского интерфейса

Интерфейс — термин в разработке систем, который может иметь разный смысл. В объектной технологии интерфейс — объявление множества особенностей, которые непосредственно не инстанциируемы[1]. **Предоставленный интерфейс** класса — множество интерфейсов, реализованных классом. Множество интерфейсов, требуемых классом от других классов, обычно называется **требуемым интерфейсом**. Множество общедоступных методов в классе называется его **общедоступным интерфейсом** (или протоколом). **Проектирование интерфейса** означает также все, что должно установить связь между компонентами программного/аппаратного обеспечения. Ясно, что понятие *пользовательского интерфейса* не связано с другими понятиями интерфейса.

Жизненный цикл проектирования пользовательского интерфейса — неотъемлемая часть жизненного цикла разработки системы. Ушли те дни, когда проектирование пользовательского интерфейса было последним в проектировании системы/программы. Они ушли с исчезновением принципа процедурного программирования, воплощенного системами КОБОЛа. Современные системы переключили этот принцип с «управления программой» к «управлению пользователем», то есть пользователь решает, *что* программа делает, *какие* действия она исполняет и *когда* эти действия выполняются.

[1] Инстанциируемый структурный тип — тип, обладающий конструктором для создания объекта, не инстанциируемый структурный тип — тип, не обладающий конструктором. (*Прим. перев.*)

Соответственно, проектирование пользовательского интерфейса переместилось от преимущественно задач проектирования/реализации системы к операциям, начинающимся на самых ранних стадиях анализа требований. В современной практике, которой следует эта книга, эскизы окон пользовательского интерфейса часто включаются в документы сценария использования (главы 8 и 14). Эти ранние эскизы чрезвычайно полезны для документации и проверки функциональных требований пользователя.

Включение эскизов пользовательского интерфейса в документы сценария использования помогает в определении функциональных требований разрабатываемой системы. Это — только исходный и дополнительный аспект проектирования пользовательского интерфейса. **Проект пользовательского интерфейса** обеспечивает пользователю доступ к функциональным возможностям системы, но как автономная операция, он в первую очередь концентрируется на **применимости** системы. Применимость — одна из особенностей FURPS+ (раздел 8.4).

Определение применимости включает впечатление и ощущение от использования пользовательского интерфейса, но его главные характеристики — простота использования системы, простота изучения, обеспечение пользовательским интерфейсом эффективности работы пользователя, скорость действий, сопротивляемость к ошибкам и восстанавливаемость после них, адаптируемость среды пользовательского интерфейса. Короче говоря, рискуя некоторой неточностью, можно сказать, что удобная система — это дружественная система.

Проект пользовательского интерфейса должен следовать ряду важных директив (принципов), чтобы тот считался «хорошим проектом». Директивы стоят выше детальных технических соображений. А применимость находится в центре этих руководящих требований.

Исторически, наиболее технически обоснованные директивы проектирования пользовательского интерфейса были представлены в контексте конкретных вида и поведения. Цель состояла в том, чтобы показать, как директива могла бы быть реализована в виде и поведении Win32 (Windows), Macintosh, Motif (Unix) и т. д. В современной практике использования нескольких платформ и настраиваемых во время выполнения вида и поведения могут быть представлены директивы сами по себе, а соответствующие вид и поведение могут быть выбраны позже.

Директивы проектирования пользовательского интерфейса можно объяснять с различными степенями детализации и совмещения [57, 61, 96]. В данной книге они сгруппированы в следующие категории:

- пользователь в управлении;
- непротиворечивость интерфейса;
- снисходительность интерфейса;
- адаптируемость интерфейса.

16.1.1. Пользователь в управлении

Директива «Пользователь в управлении» подобна предписанию, которое существенно возвышается над всеми другими директивами. Эта директива

охватывает характер взаимодействий между пользователем и программой. При современных графических и основанных на Web-технологии пользовательских интерфейсах пользователь инструктирует программу относительно каждого следующего шага, который необходимо выполнить. Даже в то время, когда программа занята выполнением некоторой обработки, а пользователь ожидает получение результата, пользователь должен сохранять ощущение управления.

Точности ради следует сказать, что директива «Пользователь в управлении» на самом деле лишь обеспечивает ощущение управления пользователем. Это иногда называется *непорождающим* принципом. Пользователь не должен иметь чувства, что требуется постоянно узнавать у программы, что делать. В любое время пользователь должен иметь возможность инструктировать программу, что делать — выбирать следующий шаг, изменять направление действия, прерывать текущую задачу и т. д.

Директиве «Пользователь в управлении» непосредственно помогает объектная технология. Фактически, современная популярность объектной технологии может быть приписана тому совпадению, что она в 1980-х годах была обеспечена появляющимися графическими пользовательскими интерфейсами. Объектно-ориентированная программа состоит из большого количества сотрудничающих объектов. Каждый объект — маленький программный модуль с набором услуг, которые могут быть выполнены, когда запрашиваются другими объектами или когда активизируются непосредственным событием, поступающим от пользователя. За каждым взаимодействием пользователя с программой стоит объект, который понимает запрос пользователя.

Чтобы пользователь чувствовал управление, программа должна быть соответствующим образом разработана. Пользовательский интерфейс должен быть удобен. Он должен использовать терминологию бизнес-области, а не компьютерный жаргон. Это должно минимизировать изменения интерфейса в результате изменений в состоянии системы. Пользователь не должен постоянно удивляться непригодностью пунктов меню, командных кнопок и других средств управления в ситуациях, когда это средство управления было бы доступно при выполнении подобных задач в подобных окнах. Любое руководство, предоставляемое программой пользователю, должно быть сдержанно и корректно.

Директива «Пользователь в управлении» предполагает надлежащую и своевременную обратную связь от программы, когда та выполняет задачу, которая не может быть прервана. Это — преимущественно проблема выполнения функций. Пользователь начинает беспокоиться, если программа выполняет некоторую обработку более долго, чем обычно, и пользователь не информирован относительно того, что происходит. Могут иметься серьезные причины, почему программа работает медленно. Это может быть из-за необычной рабочей нагрузки или из-за потребности выполнить системой административную задачу, типа реорганизации индексов в реляционной БД. Ни один из этих факторов не освобождает проектировщика пользовательского интерфейса от необходимости обеспечить пользователя своевременной информационной обратной связью.

16.1.2. Непротиворечивость интерфейса

Непротиворечивость при проектировании интерфейса означает соответствие промышленным стандартам и стандартам организации в отношении именования, кодирования, сокращений, размещения визуальных объектов, использования средств управления, включая меню, кнопки, функции клавиатуры, и т. д. Непротиворечивость интерфейса не является особенностью отдельной изолированной прикладной программы. Непротиворечивость интерфейса относится ко многим приложениям в пределах системы предприятия и/или ряда приложений, выполненных на одной и той же платформе создания пользовательского интерфейса.

Индивидуальные творческие убеждения, склонности к изобретательности, отходы от стандартов ради создания более привлекательных вещей и т. д. — не приветствуются при проектировании пользовательского интерфейса. Они только смущают пользователей и отрицательно воздействуют на директиву «Пользователь в управлении». Стандарты могут быть не лучшим отражением состояния мастерства, но проектировщик должен помнить, что практически единственная вещь, относительно которой должен заботиться пользователь, состоит в том, чтобы сделать работу самым простым, самым быстрым и наиболее знакомым путем.

Однако непротиворечивость не означает повторения снова и снова одних и тех же ошибок. Имеются пределы непротиворечивости и стандартов. Стандарты имеют тенденцию узаконить и хорошую, и плохую практику и препятствовать прогрессу. Усовершенствования ПО системы и аппаратных средств ЭВМ, новые вычислительные способности, создание приложений и даже социальные, политические, юридические или экономические факторы могут потребовать новшеств и несоблюдения стандартов в новых приложениях.

Непротиворечивость интерфейса подытоживает эстетику интерфейса. Можно предположить, что непротиворечивый интерфейс, соответствующий стандартам, приведет к эстетическому интерфейсу — интерфейсу, привлекательному визуально, с ощущением баланса и симметрии, с хорошим использованием цвета, привлекательным выравниванием и интервалом между элементами, приятной группировкой связанных элементов, без специальных требований к перемещению взгляда человека и т. д.

16.1.3. Снисходительность интерфейса

Приложение должно быть снисходительным к своим пользователям, выполняющим недопустимые действия, активизирующим недопустимые события, вводящим ошибочные данные и т. д. **Снисходительный интерфейс** подразумевает эластичный интерфейс — интерфейс, который может изящно приспосабливаться к исключениям и ошибкам. Снисходительный интерфейс подразумевает интерфейс, с которым легко экспериментировать, — интерфейс, который поощряет пользователя экспериментировать с неизвестными опциями и который обеспечивает принцип «Пользователь в управлении».

Снисходительный интерфейс обеспечивает надлежащую обратную связь пользователю и дружественную обработку исключений. Точка зрения программы на пользователя должна быть такой, как точка зрения продавца на

клиента: она не зависит от того, прав ли клиент. Пользователь может делать любую ошибку в суждении, выполнять любое действие, активизировать любую команду меню, нажимать любую командную кнопку, и при всем этом программа не должна нарушить свою работу. Глупое действие не должно выполняться. Недопустимое действие должно быть обработано.

Невыполнение действия пользователя означает, прежде всего, простую задачу забыть то, что делал пользователь. Это действительно имеет место в однопользовательском автономном приложении типа Microsoft Word. Опция «Отменить ввод» в Microsoft Word позволяет удалить не только самое последнее, но и большое количество последних изменений в документе Word. Не так обстоит дело в случае типичного промышленного приложения, обращающегося к БД и выполняющего бизнес-транзакции к сохраняемому размещению БД. Выполненная транзакция не может быть уничтожена. В лучшем случае она может быть компенсирована.

Как скомпенсировать транзакцию? Предположим, что пользователь забрал деньги со счета банка. Ясно, что, будучи совершенной, эта транзакция не может быть уничтожена. Но ей можно дать компенсацию. Пользователь должен иметь возможность поместить деньги назад на счет. Компенсирующая транзакция не является автоматической операцией системы. Пользователь должен вызвать ее. Но даже и тогда компенсирующая транзакция может потерпеть неудачу.

Представьте обратную банковскую транзакцию, в которой пользователь внес деньги в объединенный счет. Представьте затем, что пользователь решает «удалить» транзакцию вскоре после этого и забрать депонированные деньги. Транзакция изъятия может и не состояться, если тем временем другой держатель счета обратился к счету с другого места и забрал деньги, только что депонированные.

Снисхождение включает реакцию пользовательского интерфейса на недопустимое действие пользователя. Во-первых, недопустимое действие не должно нарушить работу системы. Во-вторых, недопустимое действие нельзя разрешить. В-третьих, недопустимое действие должно быть обработано мягко. Пользователь должен быть информирован относительно характера проблемы, должен быть «просвещен», почему система не может выполнить действие, и должно быть сказано, как исправить проблему и какое альтернативное действие может быть позволено.

16.1.4. Адаптируемость интерфейса

Пользовательский интерфейс должен иметь возможность приспособить любого пользователя. Разнообразие пользователей имеет множество измерений. Во-первых, пользователь может быть пользователем-новичком или опытным пользователем. Во-вторых, пользователи могут прибывать с различных мест действий. В-третьих, пользователь может быть человеком-инвалидом.

Проект пользовательского интерфейса должен быть **адаптирован** так, чтобы его особенности могли быть изменены в зависимости от знания его пользователем. Знание имеет два аспекта. Первый аспект касается знания пользовательского интерфейса как механизма взаимодействия с системой и способности «перемещаться» и «продвигаться». Второй аспект касается зна-

ния, как «получить нужное» в смысле бизнеса. Этот аспект предполагает пользователя с хорошим знанием бизнес-области приложения.

Хороший пользовательский интерфейс должен обеспечить настройку и персонализацию средства обслуживания, чтобы позволить пользователю изменять особенности пользовательского интерфейса в зависимости от уровня компьютерных и/или бизнес-знаний. Настройка — операция объявления программе уровня знаний о ней пользователя, чтобы пользовательский интерфейс мог бы изменить ее внешний вид и поведение. Персонализация имеет меньше дела со знаниями пользователя и больше с предпочтениями пользователя выполнять операции.

Проектирование пользовательского интерфейса должно адаптироваться так, чтобы его особенности могли быть изменены в зависимости от места действия пользователя [57]. Информация о месте действия — особенность современных Интернет-систем. Приложение может запросить операционную систему, на которой оно выполняется, чтобы получить информацию о месте действия, и затем может быть приспособлено к этому месту действия. Адаптация включает изменение языка пользователя (например, с английского на итальянский), набора символов и шрифтов, валюты, времени, порядка сортировки, формата даты, форматов для почтовых адресов и номеров телефона, способа, как называются люди и как на них ссылаются, пиктограмм и модельных представлений (типа значения цвета в различных культурах) и т. д.

Проектирование пользовательского интерфейса должно быть адаптируемо так, чтобы инвалиды могли использовать его средства. Различные нарушения требуют различных мер адаптируемости. Слепые люди требуют, чтобы приложение использовало систему Брайля или речь. Глухие люди требуют визуальной замены для любой звуковой продукции. Физически ущербные люди могут потребовать, чтобы в систему была включена способность распознавания голоса. Адаптированный пользовательский интерфейс должен поддерживать альтернативные устройства ввода и вывода типа терминалов Брайля, считывателя экрана, луп экрана и подобных устройств.

16.2. Компоненты пользовательского интерфейса

Программирование пользовательского интерфейса пользуется преимуществом широко доступных **библиотек классов**, которые содержат «оконные» компоненты GUI и определяют вид и поведение приложения. В мире Java библиотека классов называется Java™ Foundation Classes (JFC). Большая часть JFC состоит из набора GUI-компонентов по имени Swing. Набор обеспечивает GUI-компоненты для использования в приложениях и апплетах Java.

Набор компонентов Swing позволяет поставлять приложения со *сменным впечатлением и ощущением от использования*. Сменность имеет несколько значений. Она может означать соответствие с GUI-платформой, на которой выполняется программа (Windows, Unix и т. д.). Она может означать единую взаимную платформу впечатления и ощущения от использования, которая имеет одинаковое проявление на любой платформе выполнения. Swing вызывает это впечатление и ощущение от использования языка Java. Наконец, она

может означать настраиваемое программистом впечатление и ощущение от использования, уникальное для приложения.

Чтобы достичь **сменности**, большинство компонентов Swing сами являются платформо-независимыми или **легкими**, иногда называемыми бесподобными. Легкий компонент программируется без какого-либо использования платформо-зависимого (специального) кода. К сожалению, не все компоненты Swing легкие — некоторые являются **тяжеловесными**. В большинстве случаев, когда используются тяжеловесные компоненты, Swing обеспечивает обходные пути для скрытия специального кода, чтобы сменность впечатления и ощущения от использования была бы все еще достижима.

В Swing большинство конкретных классов, представляющих компоненты пользовательского интерфейса, называются, начиная с буквы J, например, `JDialog`, `JMenuBar`, `JButton`. Компоненты очень переплетены. Они могут быть составлены различными способами, чтобы обеспечить проектирование требуемого пользовательского интерфейса. Расклассифицировать компоненты пользовательского интерфейса в последовательные группы нелегко. Одной из грубых классификаций может быть следующая:

- контейнеры (например, `JInternalFrame` — класс «внутренняя структура» пакета Swing, `JTabbedPane` — класс «панель с закладками» пакета Swing);
- меню (например, `JPopupMenu` — класс «всплывающее меню» пакета Swing, `JRadioButtonMenuItem` — класс «элемент меню в виде переключателя» пакета Swing);
- элементы управления (например, `JRadioButton` — класс «переключатель» пакета Swing, `JScrollBar` — класс «полоса прокрутки» пакета Swing).

16.2.1. Контейнеры

Контейнеры — прямоугольные области на рабочем столе GUI, которые содержат другие компоненты, включая другие контейнеры, меню и элементы управления. В зависимости от предназначенной им цели и роли по отношению к другим контейнерам они называются окнами, диалоговыми окнами, частями окон, панелями и подобными терминами. Они определяют основные вид и поведение приложения.

Swing содержит разнообразные классы, которые могут сформировать контейнерные объекты. Четыре из этих классов — **тяжеловесные классы**: `JWindow` (класс «окно» пакета Swing), `JFrame` (класс «структура» пакета Swing), `JDialog` (класс «диалог» пакета Swing), и `JApplet` (класс «апплет» пакета Swing). В большинстве случаев основной контейнер программы — экземпляр тяжеловесного контейнера. Легкие контейнерные классы нуждаются в тяжеловесных компонентах для художественного оформления экрана и обработки событий. **Легкие классы** включают: `JInternalFrame` (класс «внутренняя структура» пакета Swing), `JDesktopPane` (класс «панель рабочего стола» пакета Swing), `JOptionPane` (класс «панель опций» пакета Swing), `JPanel` (класс «панель» пакета Swing), `JTabbedPane` (класс «панель с закладками» пакета Swing), `JScrollPane` (класс «прокручиваемая

панель» пакета Swing), JSplitPane (класс «расщепляющаяся панель» пакета Swing), JTextPane (класс «текстовая панель» пакета Swing), и JTable (класс «таблица» пакета Swing).

Часто приложение имеет только одно основное **первичное окно** и ряд **вторичных окон** (то есть окон, которые зависят от других окон; они нуждаются в родительском окне) [61]. JWindow — основной контейнер без «художественного оформления» — никаких границ, заголовка, строк меню и полос прокрутки. Из-за их отсутствия для реализации выпадающих окон обычно используются подклассы JWindow типа JFrame или JPanel. Объект JFrame имеет «художественное оформление». JPanel может использоваться, чтобы реализовать *диалоговое окно* — вторичное окно, предназначенное для того, чтобы показать сообщение и/или получать входную информацию от пользователя.

Рис. 16.1 является примером первичного и вторичного окон. Первичное окно по имени Figure 16-1 (рисунок 16.1) имеет тип JFrame. Вторичное окно по имени About (насчет) — типа JPanel.

Первичное окно на рис. 16.1 имеет строку меню (JMenuBar), панель инструментов (JToolBar) и кнопки в верхнем правом углу, чтобы сворачивать и закрывать окно. Элементы строки меню — экземпляры JMenuItem (класс «элемент меню» пакета Swing). Кнопки панели инструментов — экземпляры JButton (класс «кнопка» пакета Swing). У первичного окна можно изменять размеры с помощью любой стороны или угла его границы, а само окно можно перетащить (переместить) с помощью строки заголовка. Вторичное окно обычно имеет фиксированный размер.

JWindow, **JFrame** и **JPanel** — тяжеловесные компоненты. Чтобы иметь возможность добавлять легкие компоненты к тяжеловесному контейнерному объекту, контейнер должен использовать специальный метод getContentPane() (получить содержимое панели), например:

```
aFrame.getContentPane().add(toolBar, BorderLayout.NORTH);
```

JDialog — другой тяжеловесный компонент для создания *диалоговых окон* (диалоговых устройств). Еще один компонент Swing — JOptionPane — обеспечивает ряд стандартных диалоговых окон. Все стандартные диалоговые окна являются *модальными*. Когда модальное окно активно (видимо в настоящий момент), пользователь не может использовать другие окна, пока не будет закончена обработка в модальном окне и окно не будет закрыто. Немодальное окно позволяет вводить в него информацию так же, как и в первичное окно, лишь поместив желаемое окно поверх остальных.

Приложение может запрашивать диалоговое окно, вызывая соответствующий метод в **JOptionPane**, например:

```
void helpAbout_ActionPerformed(ActionEvent e)
  {
    JOptionPane.showMessageDialog(this, new
      Frame1_AboutBoxPanel1(),
      "About", JOptionPane.PLAIN_MESSAGE);
  }
```

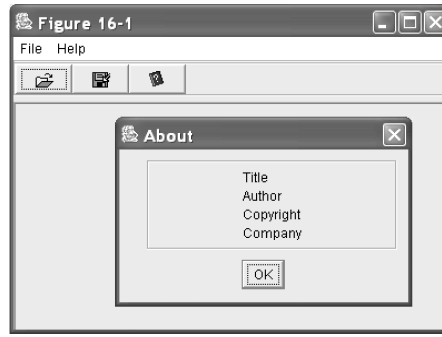


Рис. 16.1. Первичное и вторичное окна

JApplet — тяжеловесный компонент для создания **апплетов** — программ, которые выполняются внутри Web-страницы (глава 17). Апплеты загружаются с сервера и могут быть кэшированы в Web-браузере для выполнения. `JWindow`, `JDialog` и `JApplet` зависят от `JFrame`. Следовательно, минимизация `JFrame` минимизирует его окно, диалоговое окно или апплет. Удаление `JFrame` уничтожает также его окно, диалоговое окно или апплет и т. д.

Swing обеспечивает большое количество легких панелей, чтобы использовать их в других контейнерах. Одна из них — `JOptionPane`, которая реализует набор стандартных окон сообщений типа выдачи предупреждения, запроса подтверждения действия, объяснения события и т. д. Другие очень полезные панели включают `JTabbedPane`, `JScrollPane`, `JTextPane` и `JSplitPane`.

Объект **`JTabbedPane`** дает окно с рядом «страниц/панелей с закладками». **Страницы с закладками** накладываются друг на друга, так что в любой момент времени может быть видна только одна страница. Нажатие на закладку переводит ее страницу на передний план. Страницы с закладками очень полезны для работы с большим объемом данных в одном диалоговом окне. Они занимают особое место в программировании апплетов, когда нельзя использовать выпадающие диалоговые окна [25].

Рис. 16.2 демонстрирует панель с закладками, содержащую две закладки с именами `Movie` (фильм) и `Actor` (актер). Добавление объекта `JPanel` к объекту `JTabbedPane` создает каждую закладку.

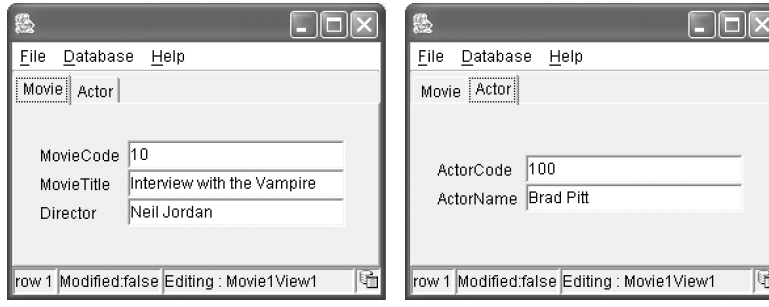


Рис. 16.2. Панель с закладками

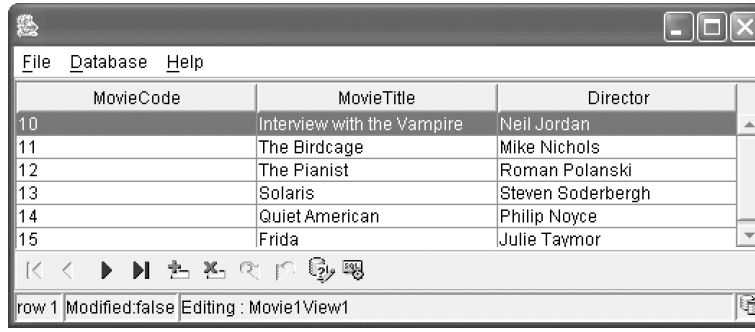


Рис. 16.3. Таблица

Объект **`JScrollPane`** обеспечивает возможность горизонтальной и вертикальной прокрутки окна. Можно выбрать различное сочетание **полос прокрутки**. Объект **JTextPane** отображает текст. Это — хороший кандидат, чтобы быть «прокручиваемым клиентом», то есть он может получить возможность прокрутки от `JScrollPane`. Расширяя (наследуя) `JEditorPane` (класс «панель редактирования» пакета Swing), `JTextPane` получает основные способности редактирования.

Объект **`JSplitPane`** обеспечивает *расщепляющееся окно* с двумя или большим количеством зависимых панелей, то есть действие на одной панели приводит к визуальным или другим изменениям в зависимой панели. Панели могут быть отделены вертикально или горизонтально. Windows Explorer является примером окна с «расщепляющейся панелью».

Один важный компонент Swing, который не поддается легкой классификации — **`JTable`**. Хотя он не совсем окно или панель, объект `JTable` является контейнером данных. `JTable` представляет таблицу строк и столбцов. Он может использоваться для всех видов сложных табличных манипуляций, включая функциональные возможности, наподобие электронных таблиц. Однако основное использование `JTable` в виде сетки — показать записи данных, полученные из реляционной БД.

Рис. 16.3 — пример окна, содержащего объект `JTable`. Средство прокрутки обеспечивается объектом `JScrollPane`. `JScrollPane` использует объект `JViewport` (класс «порт просмотра» пакета Swing), обеспечивающий «порт просмотра» в источнике данных — содержании таблицы `Movie` БД, как показано в следующем фрагменте кода:

```
private JTable tableMovie1View1 = new JTable();
private JScrollPane scroller = new JScrollPane();
scroller.getViewport().add(tableMovie1View1, null);
```

Управление расположением

Swing использует **менеджер расположения**, чтобы разместить компоненты внутри контейнера [25]. Если абсолютное расположение не запрограммировано явно, компоненты динамически размещаются в пределах контейнера и корректируются в соответствии с измерениями контейнера. Размещение су-

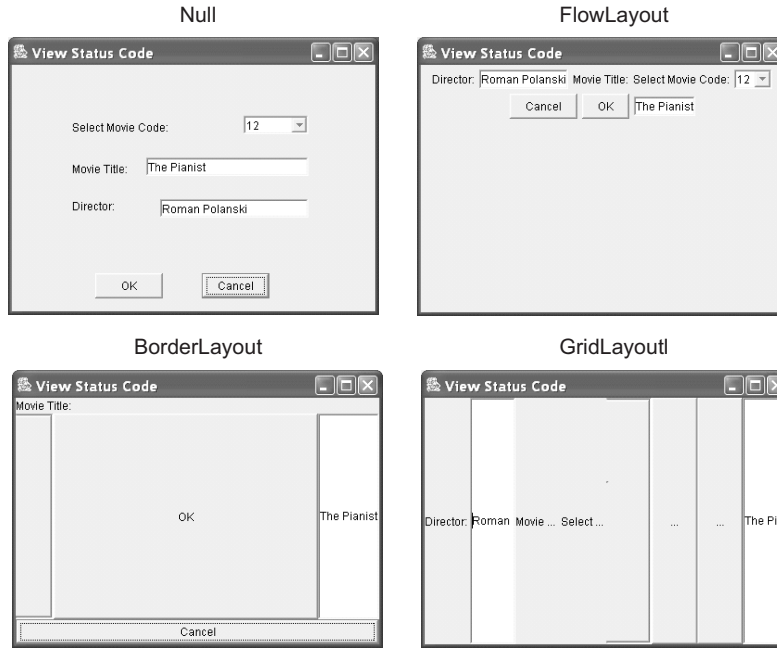


Рис. 16.4. Менеджеры расположения, примененные к одному и тому же набору компонентов

щественно отличается в зависимости от выбранного менеджера расположения. Метод `setLayout()` (установить расположение) позволяет выбрать желаемого менеджера, например:

```
aFrame.getContentPane().setLayout(new GridLayout(6,5));
```

Расположения, обеспечиваемые библиотекой Swing, включают `BorderLayout` (расположение границы), `FlowLayout` (расположение потока), `GridLayout` (расположение по сетке), `BoxLayout` (расположение в прямоугольнике) и `GridBagLayout` (расположение набора по сетке). **`BorderLayout`** — схема расположения, используемая по умолчанию для большинства контейнеров, кроме `JPanel` (который использует по умолчанию `FlowLayout`). Ручное расположение графических компонентов внутри контейнеров может быть запрограммировано с менеджером расположения, установленным в значение `null`:

```
aContainer.getContentPane().setLayout(null);
```

Объяснение детальных характеристик этих расположений вне возможностей данной книги [25]. Рис. 16.4 показывает, как отличаются расположения, когда применяются различные менеджеры расположения к одному и тому же множеству компонентов. Расположение в верхнем левом углу — задаваемое вручную `null`-расположение. Остальные расположения были получены применением различных схем к исходному `null`-расположению.

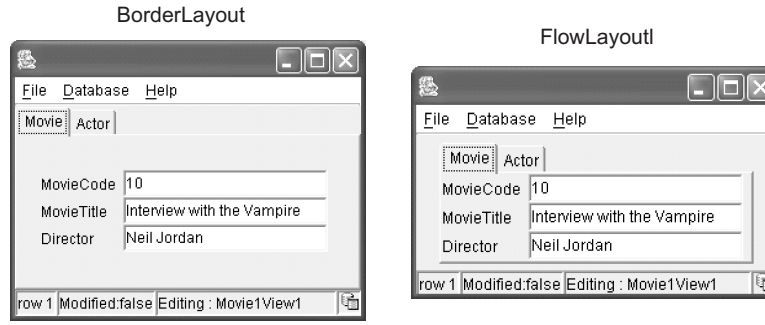


Рис. 16.5. Менеджеры расположения, примененные к панели с закладками

Рис. 16.4 может создать впечатление, что некоторые популярные расположения, типа `BorderLayout`, хуже других. На самом деле это не так. Выбор расположения должен быть таким, чтобы он соответствовал контейнеру. Рис. 16.5 показывает расположения `BorderLayout` и `FlowLayout`, примененные к панели с закладками.

Управление выбором уровней

Приложение может открыть много окон, которые затем накладываются друг на друга. Класс Swing по имени **`JLayeredPane`** (класс «панель с уровнем» пакета Swing) добавляет глубину к контейнерам, поддерживая выбор уровней окон и других компонентов. Этот класс обеспечивает методы перемещения компонентов на передний план, на задний план или в соответствующее положение относительно друг друга.

Большинство компонентов помещается на стандартный (используемый по умолчанию) уровень. Компоненты **уровня, используемого по умолчанию**, соответственно накладываются друг на друга в зависимости от выбора пользователем этих компонентов (например, нажатие мышью окна делает его окном, расположенным поверх остальных). `JLayeredPane` позволяет также объявлять специальные характеристики уровней, которые заставят их вести себя предопределенным образом по отношению к другим уровням. Специальные случаи следующие:

- **уровень палитры**, который всплывает поверх уровня, используемого по умолчанию (например, всплывающая панель инструментов);
- **модальный уровень**, который помещается поверх всех других активных окон, панелей инструментов и палитр приложения и не допускает переключения к этим другим окнам, если он сам непосредственно не будет закрыт (например, модальное диалоговое окно);
- **выпадающий уровень**, который отображается временно в своем собственном уровне выше других уровней (например, комбинированная строка ввода, контекстное окно);
- **уровень перетаскивания**, который делает компонент видимым, когда он перетаскивается до того, как он будет опущен на уровень назначения.

Каждый уровень имеет отличительный целый номер (от 1 до 5). Уровень для компонента устанавливается, когда он добавляется к контейнеру. Текущее положение (глубина) компонента внутри уровня также обозначается целой величиной. Эта величина может быть определена непосредственно (`setLayer()` — установить уровень). Положение может быть изменено путем вызова `moveToFront()` (переместить на передний план) и `moveToBack()` (переместить на задний план).

16.2.2. Меню

Окна и некоторые другие компоненты могут содержать меню. Классы библиотеки Swing для поддержки меню включают **`JMenuBar`** (класс «строка меню» пакета Swing), **`JMenu`** (класс «меню» пакета Swing), **`JMenuItem`** (класс «элемент меню» пакета Swing), **`JCheckBoxMenuItem`** (класс «отмечаемый элемент меню» пакета Swing) и **`JRadioButtonMenuItem`** (класс «элемент меню в виде переключателя» пакета Swing). Модель классов, связанных с меню компонентов Swing, показана на рис. 16.6.

`JPopupMenu` (класс «всплывающее меню» пакета Swing), `JMenuBar`, `JMenu` и `JMenuItem` реализуют один и тот же интерфейс `MenuElement` (элемент меню). `JMenuItem` расширяет `AbstractButton` (абстрактная кнопка) — это проясняет, что `JMenuItem` является элементом управления (раздел 16.2.3). `JCheckBoxMenuItem` и `JRadioButtonMenuItem` — два вида `JMenuItem`.

Модель классов Swing для меню искажает факт, что объект `JMenuItem` содержится в объекте `JMenu`, а объект `JMenu` содержится в объекте `JMenuBar`. Никакое отношение включения (агрегирования) не используется в модели, чтобы представить этот факт. Возможно, представление, где `JMenu` является подклассом `JMenuItem`, — пример неподходящего использования наследования реализации (раздел 9.1.5).

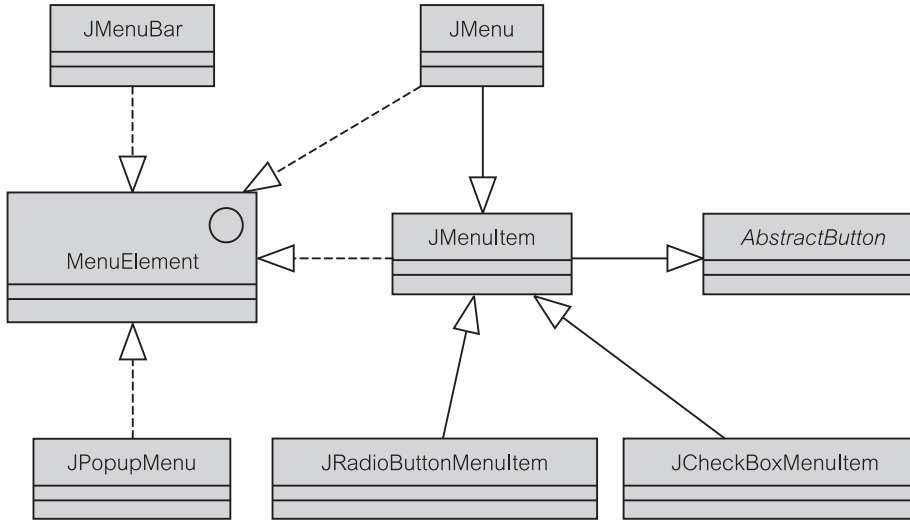


Рис. 16.6. Меню библиотеки Swing

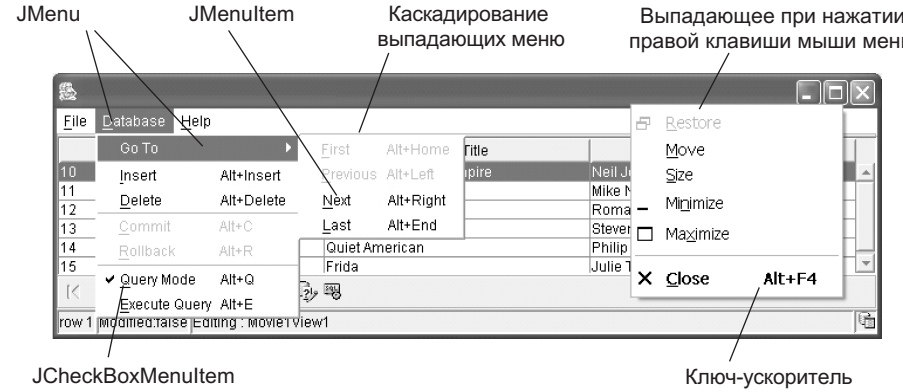


Рис. 16.7. Разные виды меню

Рис. 16.7 дает визуальное представление меню Swing. Строка меню (`JMenuBar`) содержит объекты `JMenu`. Элемент выпадающего объекта `JMenu` может быть объектом `JMenuItem` (например, `Insert` — добавить) или другим объектом `JMenu` (например, элемент меню `Go To` — перейти к...). Последний приводит к **каскадированию** («отображению справа») *всплывающего* меню (`JPopupMenu`). Другой пример `JPopupMenu` — **всплывающее при нажатии правой клавиши мыши** меню. Объект `JMenuItem` может быть пунктом `JCheckBoxMenuItem` (например, элемент `QueryMode` — режим запроса) или выбором `JRadioButtonMenuItem`. Каждое меню может иметь ключ-ускоритель.

Рис. 16.8 объясняет использование панелей инструментов в проектировании пользовательского интерфейса. Как правило, **панель инструментов** содержит действия и элементы управления, которые дублируют функциональные возможности нескольких наиболее полезных элементов меню. Это позволяет осуществлять более быструю активизацию данных элементов меню. Контейнер Swing по имени **`JToolBar`** (панель инструментов) помогает в проектировании панелей инструментов пользовательского интерфейса. Панели инструментов могут иметь установленное местоположение в рамке окна или они могут быть плавающие, то есть, могут быть выделены из структуры в отдельное окно.

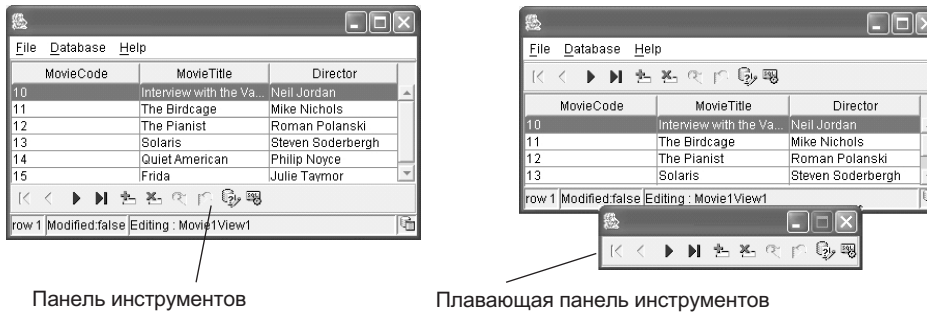


Рис. 16.8. Панели инструментов

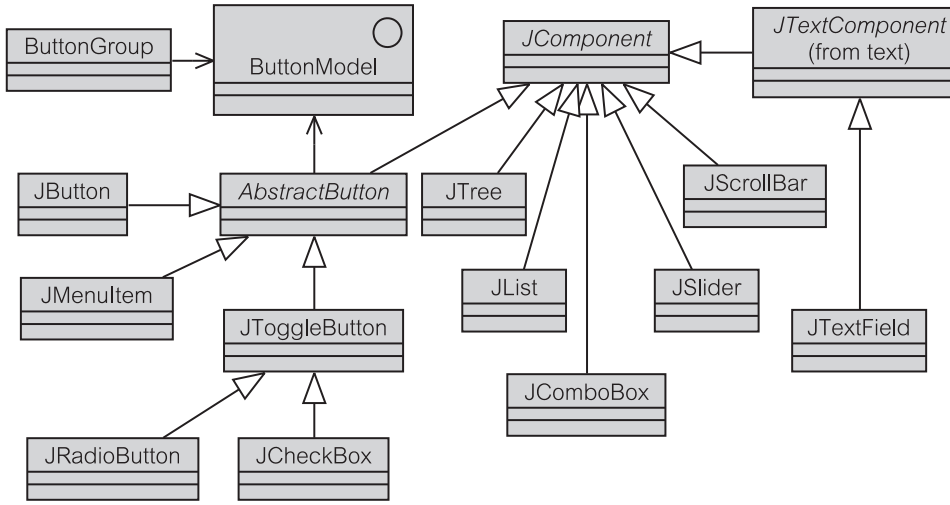


Рис. 16.9. Элементы управления Swing

16.2.3. Элементы управления

Элементы управления — сердце модели событий Swing (Swing event model). Они перехватывают, понимают и реализуют действия пользователя. Вообще говоря, они делятся на:

- командные *кнопки*, которые унаследованы от абстрактного класса по имени **`AbstractButton`** (абстрактная кнопка);
- *другие элементы управления*, которые унаследованы непосредственно от корневого абстрактного класса по имени `JComponent` (класс «компонент» пакета Swing).

Рис. 16.9 представляет модель классов для элементов управления Swing. Модель показывает, что имеются четыре категории кнопок: **`JButton`** (класс «кнопка» пакета Swing), **`JRadioButton`** (класс «кнопка-переключатель» пакета Swing), **`JCheckBox`** (класс «выключатель» пакета Swing), и **`JMenuItem`** (класс «элемент меню» пакета Swing), рассмотренная в предыдущем разделе. Чтобы реализовать находящееся во взаимосвязи поведение набора кнопок, они должны быть добавлены в компонент `ButtonGroup` (группа кнопок) [25].

Рис. 16.10 дает визуальное представление кнопок и других элементов управления, перечисленных на рис. 16.9. Элементы управления реализованы с моделью событий Swing. Каждый элемент управления может иметь одного или большее количество объектов-подписчиков, которые «прислушиваются» к нему. Например, объект `JTextField` (класс «текстовое поле» пакета Swing) может публиковать события типа изменения текста, нажатия кнопки `Delete` (удалить) клавиатуры или выбора элемента Copy (копировать) меню.

`JButton` представляет собой обычную кнопку. В отличие от `JButton` кнопка `JToggleButton` (класс «кнопка-переключатель» пакета Swing) бу-

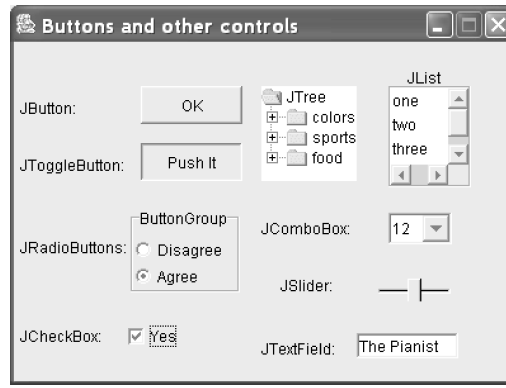


Рис. 16.10. Кнопки и другие элементы управления

дучи нажатой, остается нажатой (установленной в состояние true), пока она не нажата снова. Нажатая `JToggleButton` указывает, что кнопка активна. Две категории `JToggleButton` — это `JCheckBox` и `JRadioButton`.

`JCheckBox` изображается в виде квадрата и может быть установлен в true или false. Значение true **выключателя** обозначается «галочкой» на нем. `JRadioButton` используется, чтобы выбрать одну и только одну кнопку из множества кнопок, как, например, в случае выбора фильтра итерации 2 (рис. 14.3). Как и во многих механических устройствах, когда одна **кнопка-переключатель** «нажимается», все другие кнопки «отпускаются». Эти функциональные возможности получаются путем добавления каждой кнопки-переключателя в `ButtonGroup`.

`JComboBox` (класс «комбинированная строка ввода» пакета Swing) позволяет делать выбор из **выпадающего списка (комбинированная строка ввода)**. Комбинированная строка ввода часто называется на жаргоне **списком для выбора** (picklist). Объект `JComboBox` может быть редактируемым или нет (если нет, он не разрешает вводить собственные значения пользователя). По умолчанию объект `JComboBox` не редактируемый. `JComboBox` допускает выбор только одного значения.

`JList` (класс «список» пакета Swing) поддерживает выбор нескольких значений, используя типичные действия: `Ctrl-click` (Ctrl + нажатие клавиши мыши) и `Shift-click` (Shift + нажатие клавиши мыши). Выбор нескольких значений в `JList` возможен, потому что `JList` является окном со списком элементов (list box) фиксированного размера, хотя и прокручиваемым (обратите внимание, что прокрутка — не автоматическая особенность `JList`; она может быть получена включением `JList` в **`JScrollPane`** — класс «панель прокрутки» пакета Swing).

Значение и применимость компонентов **`JScrollBar`** (класс «линейка прокрутки» пакета Swing) и **`JSlider`** (класс «ползунок» пакета Swing) библиотеки Swing таковы, как это можно было бы ожидать. Полосы прокрутки могут быть добавлены к большинству контейнеров. Ползунки позволяют контролировать вход, наподобие регулятора громкости.

Последний, но не менее важный компонент, **`JTree`** (класс «дерево» пакета Swing) — очень полезный элемент управления и один из наиболее мощных в Swing [25]. В большинстве случаев простое представление дерева, показывающее иерархию различных элементов и контейнеров, как в Windows Explorer, является достаточным. Однако если приложение потребует более сложную реализацию дерева, в библиотеке Swing, вероятно, возникнет проблема.

16.3. Управление событиями пользовательского интерфейса

Пользовательский интерфейс описывает суть **обработки событий** (раздел 9.1.8) и **паттерн Наблюдатель** (раздел 9.3.4). PCMEF-шаблон рекомендует использовать обработку событий для восходящего уведомления уровней — UNP-принцип (раздел 9.2.2). Обработка событий в проекте пользовательского интерфейса касается только одного PCMEF-уровня — уровня `presentation` (представление).

Модель событий библиотеки Swing получается из MVC-шаблона (раздел 9.2.1). В Swing аспекты представления и диспетчера MVC объединены в отдельный компонент (типа `JButton` — кнопка). Для каждого такого компонента Swing определяет отдельный интерфейс модели (типа `ButtonModel` — модель кнопки) — рис. 16.7. Все, что должно сделать приложение для изменения предопределенного поведения компонента, это реализовать свой интерфейс модели своим собственным способом.

Любое действие пользователя через доступные ему устройства ввода (мышь, клавиатура и т. д.) вызывает в программе событие, которое должно быть перехвачено объектом-источником события — объектом **издателя**. Источник события уведомляет о возникновении события каждый объект, заинтересованный в этом событии. Эти целевые объекты являются **объектами-подписчиками** (приемниками, наблюдателями). Каждый источник события может иметь много подписчиков, и каждый подписчик может слушать много источников события.

Рис. 16.11 — абстрактный пример обработки события. Каждый класс подписчика реализует интерфейс приемника библиотеки Swing (типа `ActionListener` — приемник операций) или он расширяет класс, который реализует такой интерфейс. Код, который регистрирует объект подписчика в качестве приемника, может быть включен в класс издателя, или эту обязанность можно передать отдельному объекту-регистратору (рис. 9.20).

В примере, когда пользователь нажимает мышью компонент `cancelButton` (кнопка завершения), кнопка вызывает событие действия. Это вызовет метод `actionPerformed()` (выполненное действие) в объекте `Subscriber3` (подписчик 3) — единственный метод интерфейса `ActionListener` библиотеки Swing, который реализован объектом `Subscriber3`. Аргументом для `actionPerformed()` является объект `ActionEvent` (событие действия). `ActionEvent` сообщает объекту `Subscriber3` относительно события и об его источнике (объект `Publisher2` — издатель 2).

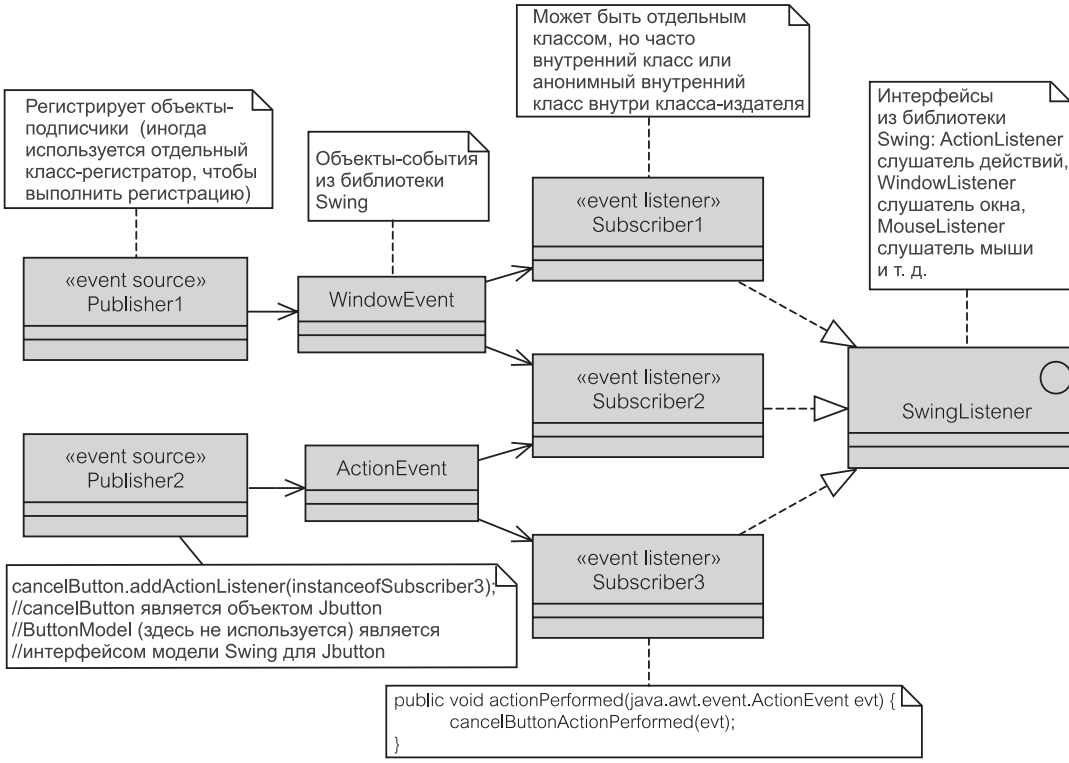


Рис. 16.11. Управление событиями в пакете Swing

Как отмечено на рис. 16.11, *класс подписчика* может быть отдельным классом. Это может быть хорошим вариантом в более сложной обработке событий. В PCMEF отдельный класс подписчика, вероятно, будет находиться на уровне `control` (управление) и слушать издателя на уровне `presentation`. В более простых случаях класс подписчика может быть **внутренним классом** класса издателя или даже **анонимным внутренним классом**. Листинги 16.1, 16.2 и 16.3 представляют три примера, которые приводят к одному и тому же результату: они реализуют `ActionListener` объекта-издателя.

Листинг 16.1. Подписчик как отдельный класс

Подписчик как отдельный класс

```
public class ShowTitleListener implements ActionListener{
   TitleChanger host; //ссылка на объект, который изменяет
                      заголовок
   public ShowTitleListener(TitleChanger host){
      this.host = host;
```

```
    }
    public void actionPerformed(ActionEvent evt){
                        //запрос изменить поступающий заголовок
       System.out.println("The title is now changed to " +
                                              host.changeTitle());
    }
}
public class TitleChanger{
    public void show(){
       Button b = new Button("Change Title");
       ...
       b.addActionListener(new ShowTitleListener(this));
                    //это то, где данная операция используется
```

Листинг 16.2. Подписчик как внутренний класс

Подписчик как внутренний класс

```
public class TitleChanger{
    public void show(){
       Button b = new Button("Change Title");
       ...
       b.addActionListener(new ShowTitleListener());
       ...
    }
    //объявление внутреннего класса
    private class ShowTitleListener implements ActionListener{
       public void actionPerformed(ActionEvent evt){
                          //запрос изменить поступающий заголовок
       System.out.println("The title is now changed to " +
       TitleChanger.this.changeTitle());
                                //прямая ссылка на внешний класс
       }
    }
}
```

Листинг 16.3. Подписчик как анонимный внутренний класс

Подписчик как анонимный внутренний класс

```
public class TitleChanger{
    public void show(){
       Button b = new Button("Change Title");
       ...
       b.addActionListener(new ActionListener(){
```

```
                    //создание анонимного внутреннего класса
            public void actionPerformed(ActionEvent evt){
               System.out.println("The title is now changed to " +
               TitleChanger.this.changeTitle());
                                  //прямая ссылка на внешний класс
            }
         });
         ...
      }
}
```

Все компоненты Swing имеют **интерфейс модели**, определенный в библиотеке. Например, интерфейс модели для кнопок называется `ButtonModel`, для `JList` (класс «список» пакета Swing) он называется `ListModel` (модель списка), для `JTable` (класс «таблица» пакета Swing) он называется `TableModel` (модель таблицы) и т. д. Интерфейсы модели могут быть реализованы для приложения уникальным способом согласно требуемой логике приложения. Однако, для компонентов, не требующих уникальной реализации модели, взаимодействие со стандартной моделью скрыто за методами класса компонента. Это выполняется методом компонента, делегирующим сообщение к его модели, например:

```
public String getTitle() {
  return getModel().getTitle();
}
```

Рис. 16.12 показывает диаграмму последовательности действий, иллюстрирующую, как **`TableModel`** используется в учебном примере EM. На рисунке интерфейс `TableModel` реализован с помощью `PMessageTableModel` (класс «модель таблицы для сообщений» пакета `presentation`). Рисунок показывает, как `PMessageTableModel` используется для хранения исходящего сообщения и выполнения фильтрации исходящих сообщений, отображаемых объектом `JTable`. Операции сортировки и фильтрации размещены в `PMessageTableModel`. Это сделано потому, что `PMessageTableModel` отвечает за хранение данных, которые могут затем быть отсортированы и отфильтрованы. Запросы фильтрации идут от `PWindow` (класс «окно» пакета `presentation`), как показано на рисунке. `PMessageTableModel` фильтрует данные, делегируя запрос к его `PDisplayList` (класс «список отображения» пакета `presentation`). Последующие операции `JTable`, запрашивающие данные от своей модели (в данном случае от `PMessageTableModel`), приводят к возвращению фильтрованных данных.

Рис. 16.12 иллюстрирует, как `PWindow` запрашивает `PMessageTableModel` для выполнения фильтрации в соответствии с некоторыми критериями (показаны только `datefilter` — фильтрация по дате и `contactfilter` — фильтрация по деловому партнеру). Затем `PMessageTableModel` запрашивает свои данные, `PDisplayList`, чтобы устранить те `PDisplayData` (класс «отображаемые данные» пакета `presentation`), которые не удовлетворяют критерию. Как только фильтрация будет закончена, `PWindow` вызывает экран, который бу-

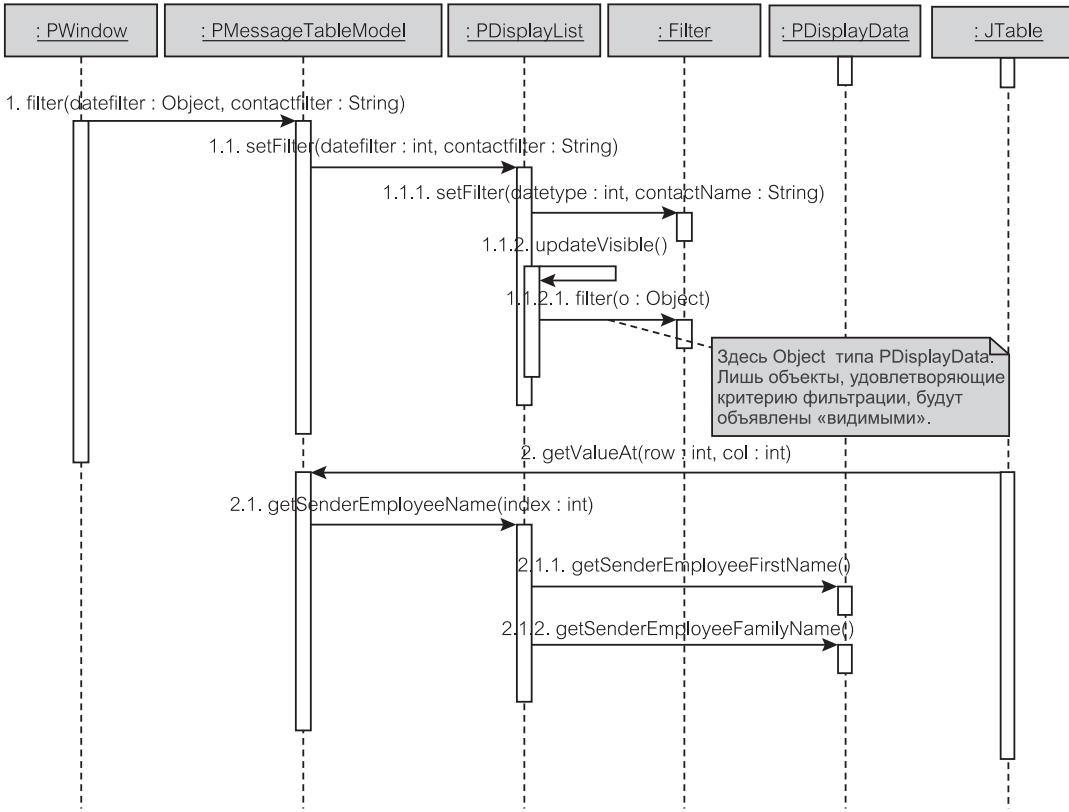


Рис. 16.12. Диаграмма последовательности действий, включающая PMessageTableModel

дет перерисован, и заставляет `JTable` показать измененное подмножество данных. `JTable` запрашивает свою модель, `PMessageTableModel`, для получения *нового* списка сообщений, которые будут отображены на экране.

`JTable` запрашивает данные на основе строк и столбцов. `PMessageTableModel` переводит этот запрос на соответствующие поля, размещенные в модели. Эта обработка выполняется сообщением `getSenderEmployeeName()` (получить имя служащего-отправителя) к `PDisplayList`. `PDisplayList` затем формирует имя служащего-отправителя, объединяя его/ее имя и фамилию.

16.4. Паттерны и пользовательский интерфейс

Графические пользовательские интерфейсы (Graphical user interfaces — GUI) в значительной степени ответственны за переключение от процедурного к объектно-ориентированному ПО. Хотя объектно-ориентированное ПО и языки программирования были известны с 1960-х годов, использование GUI требовало перехода к использующему события программированию, свойствен-

ному объектной ориентации. GUI перемещают центр управления от программы к пользователю. Пользователь отвечает за выполнение программы, решая, какие действия должны быть далее выполнены. Эти действия переводятся во входные события. Каждое входное событие требует объекта, который будет обслуживать его.

Графические пользовательские интерфейсы также в значительной степени ответственны за популярность **паттернов** в проектировании системы. «Вид и поведение» обеспечивают продажу ПО. Пользователи требуют ранних прототипов пользовательского интерфейса, чтобы оценить вид и поведение системы. Интегрированные среды разработки (Integrated development environments — IDE) и технологии быстрой разработки приложения (Rapid Application Development — RAD) позволяют быстро создавать такие прототипы. Все эти требования приводят к широкому повторному использованию компонентов GUI. Такой прогресс зарегистрирован в паттернах, относящихся к GUI, и наоборот, большинство особо важных паттернов **Банды четырех** (Gang of Four — GoF) [32] может быть найдено в проекте компонентов GUI.

Этот раздел охватывает только небольшое подмножество обычно используемых паттернов в разработке пользовательского интерфейса. Рассматриваемые паттерны следующие: Наблюдатель (Observer), Декоратор (Decorator), Цепочка обязанностей (Chain of Responsibility) и Команда (Command). Паттерны Наблюдатель и Цепочка обязанностей были обсуждены ранее в других контекстах.

16.4.1. Наблюдатель

Различные компоненты в пользовательском интерфейсе наблюдают объекты модели (сущности) согласно шаблону Model-View-Controller (раздел 9.2.1). Они изменяют поведение или представление пользовательского интерфейса, основанные на полученных уведомлениях о том, что модель изменилась. Такое отношение наблюдения модели удобно назвать **паттерном Наблюдатель** (раздел 9.3.4).

Рассмотрим пример отношения между компонентом `JTable` (класс «таблица» пакета Swing) и его моделью `JTableModel` (класс «модель таблицы» пакета Swing), как показано на рис. 16.13. Таблица регистрирует события своей модели. После модификации модели таблица уведомляется об этом и поэтому способна соответственно ответить. Рис. 16.13 показывает наблюдателей, реализованных на взаимодействии между `JTable` и его моделью. Рисунок не включает все компоненты, связанные с `JTable`, так как их достаточно много. Он показывает только `JTableModel` и `TableCellEditor` (редактор ячеек таблицы).

Таблица сформирована с экземпляром `JTableModel`. Она регистрирует свой интерес к модели через метод `addTableModelListener()` (добавить приемник модели таблицы). Это гарантирует, что таблица будет уведомлена, когда модель изменится.

Когда пользователь решает изменить ячейку в таблице двойным нажатием мышью по ячейке, таблица строит `CellEditor` (редактор ячейки). В `CellEditor` (или скорее его редактируемом компоненте) устанавливается величина, извлеченная из модели через `getValueAt()` — получить значе-

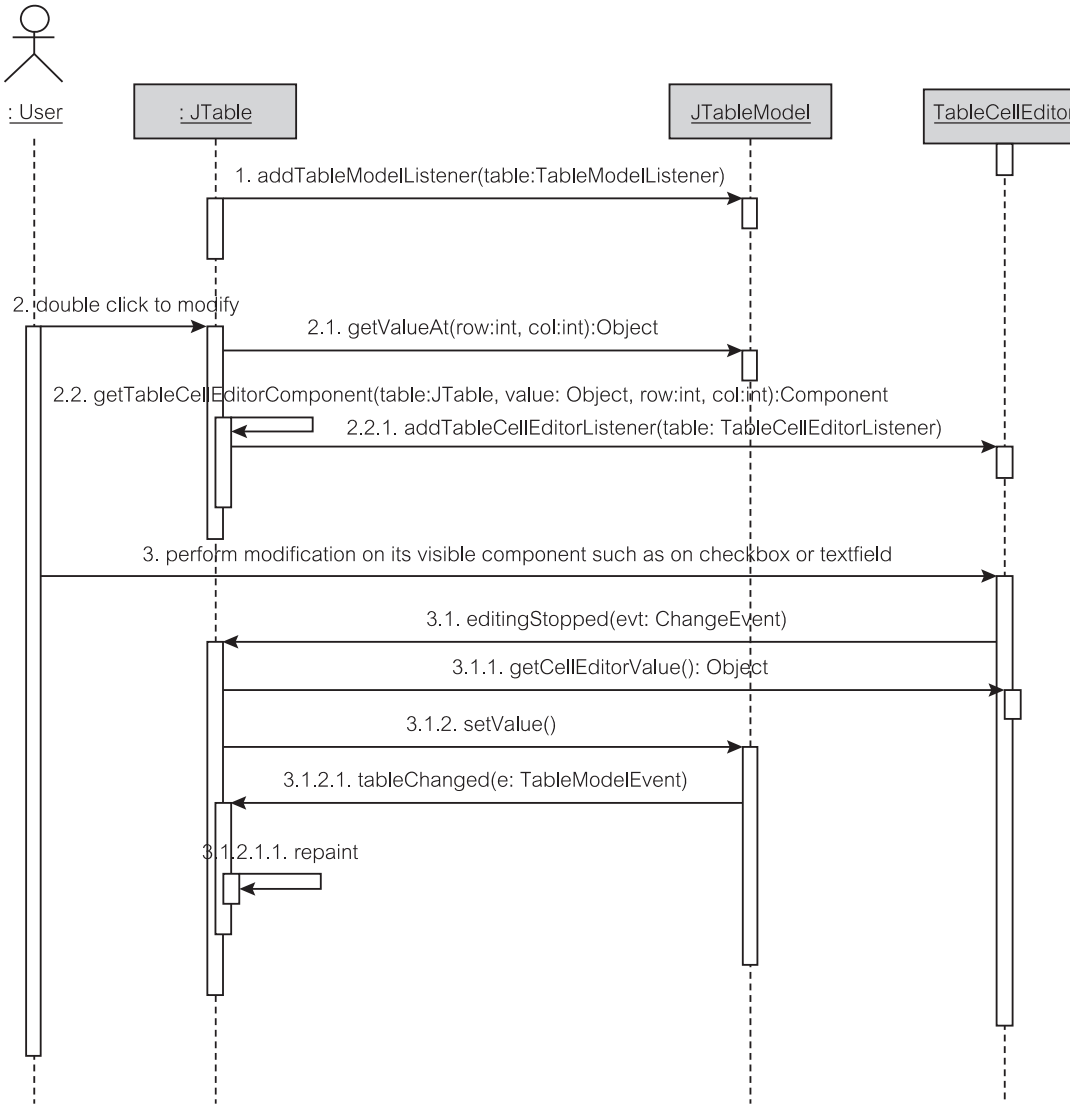


Рис. 16.13. Наблюдатели

ние о... (сообщение 2.1). Что интересно отметить, так это то, что сам `TableCellEditor` фактически является моделью по отношению к `JTable`. `JTable` — подписчик (приемник) к `TableCellEditor` (сообщение 2.2.1).

Рис. 16.13 использует паттерн Наблюдатель для:

- наблюдения компонентом `JTable` модели `JTableModel` (сообщение 1);
- наблюдения компонентом `JTable` редактора `TableCellEditor` (сообщение 2.2.1).

Когда на редактируемом компоненте типа выключателя или текстового поля выполнена модификация (сообщение 3), `TableCellEditor` уведомляет таблицу через ее метод `editingStopped()` — редактирование завершено (реализация Наблюдателя из сообщения 2.2.1). В этом случае таблица может обработать измененную величину, извлеченную из `getCellEditorValue()` (получить значение редактора ячейки), и изменить значение ее модели `JTableModel` (сообщение 3.1.2 `setValue()` — задать значение).

`JTable` действует как посредник между величиной, измененной в компоненте `TableCellEditor`, и величиной, запасенной в `JTableModel`. Модификация в `JTableModel` вызывает уведомление всех заинтересованных сторон (`JTable` и другие классы, которые регистрируются через `addTableModelListener()`, сообщение 1). Уведомление осуществляется вызовом их метода `tableChanged()` (таблица изменена). Наконец наблюдатель `JTable`, перерисовывает себя (сообщение 3.1.2.1.1), запрашивая модель о ее данных, которые будут показаны на экране.

Паттерн Наблюдатель в проекте пользовательского интерфейса используется широко. Это позволяет разрабатывать объединения компонентов пользовательского интерфейса и взаимодействия компонентов, обладающие большими возможностями (сложные уведомления через события). Паттерн Наблюдатель имеет дело с требованиями систем, управляемых событиями, так что модель может быть изменена в любое время, и пользовательский интерфейс должен знать о таких модификациях.

16.4.2. Декоратор

Паттерн Декоратор [32] позволяет использовать ряд классов-вместилищ, чтобы *украсить* внешний вид компонента. Это происходит, например, когда используются полосы прокрутки, чтобы украсить `JList` (класс «список» пакета Swing), формируя прокручиваемый список. Другой пример — `Frame` (структура), украшенная полосами прокрутки, строкой меню, панелью инструментов и границами.

Рис. 16.10 иллюстрирует использование декораторов. Он показывает `Panel` (панель), украшенную границей с названием «`ButtonGroup`» (группа кнопок). Он также показывает `JList`, помещенный внутри `JScrollPane` (класс «прокручиваемая панель» пакета Swing). `JScrollPane` является прокручиваемой панелью.

Паттерн Декоратор используется, чтобы украсить и увеличить привлекательность (дружелюбие к пользователю) пользовательского интерфейса. Единственной целью некоторых компонентов пользовательского интерфейса, типа различных границ, является использование их в качестве декораторов.

16.4.3. Цепочка обязанностей

Когда компонент неспособен обеспечить обслуживание отдельного запроса, он может решить делегировать такую обязанность своим подкомпонентам, пока один из подкомпонентов не сможет обслужить запрос. Связанный с этим паттерн называется **Цепочкой обязанностей** (раздел 9.3.3).

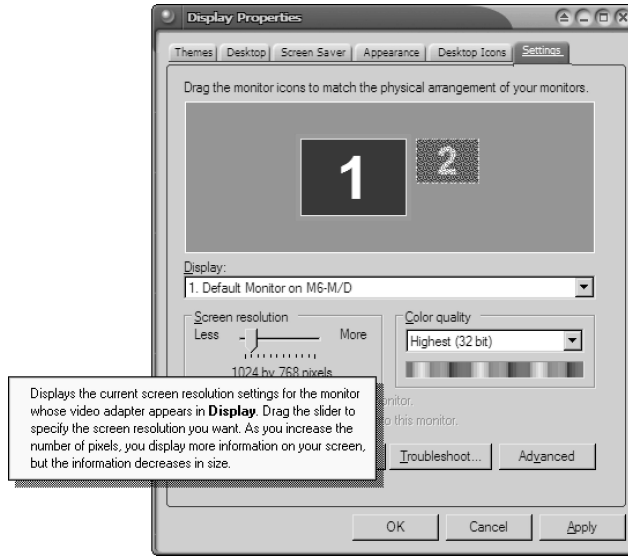


Рис. 16.14. Help (помощь): Цепочка обязанностей

ОС Windows реализует этот паттерн, чтобы поддержать свою систему *Help* (помощь). Всякий раз, когда пользователь Windows не понимает цели некоторого элемента управления в GUI, он/она может нажать правой клавишей по компоненту, чтобы найти разъяснение его цели. Рис. 16.14 показывает, как текущий компонент (ползунок для изменения разрешения экрана) отвечает на нажатие правой клавишей пользователем и выбор элемента меню «What's this?» (Что это?).

Текущий компонент определит, может ли он ответить на запрос. Если нет, он делегирует запрос к следующему объекту в цепочке обязанностей. В конечном счете должен быть найден некий компонент, который может отобразить соответствующую информацию на экране и остановить *цепочку*.

На рис. 16.14 текущий компонент делегирует запрос к своему контейнерному компоненту. Контейнерный компонент — панель с границей, содержащая название «`Screen Resolution`» (разрешение экрана). Если панель имеет ответ, она отобразит информацию на экран и остановит дальнейшую передачу запроса. Цепочка запросов продолжается до самого верхнего контейнерного компонента, который может или представить информацию (это может быть или конкретная, или общая информация) или остановить цепочку без ответа.

Этот паттерн упрощает проектирование обычно требуемых последовательностей делегирования, типа системы *Help*. Однако у этого паттерна имеется недостаток. Он требует, чтобы все вложенные контейнеры реализовали этот паттерн. Контейнер или компонент должны быть способны решать, может ли он ответить на запрос или должен передать его далее. Кроме того, компоненты увеличиваются в размере, если они хотят участвовать в нескольких различных системах контекстной *помощи*, то есть больше чем в одном контексте цепочек обязанностей.

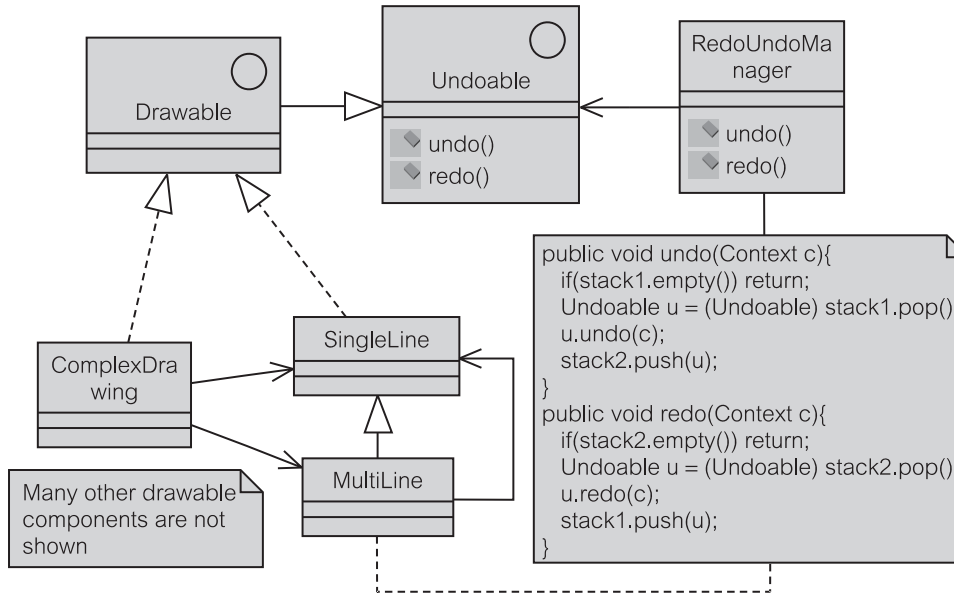


Рис. 16.15. Пример паттерна Команда: Undo (отмена)

16.4.4. Команда

Паттерн Команда [32] подобен паттерну Цепочка обязанностей. Он делегирует запрос к следующему компоненту, пока один из них не сможет обслужить запрос. Небольшое отличие от Цепочки обязанностей, где запрос делегируется родительскому контейнеру и может идти до вершины иерархии, заключается в том, что компоненты в паттерне Команда делегируют запрос известному интерфейсу.

Рассмотрим, например, выполнение приложения. Приложение нужно обеспечить механизмом отмены/восстановления. Каждая операция, выполненная в приложении, должна быть *зарегистрирована*, чтобы позволить приложению возвратиться к своим предыдущим состояниям. Однако операции, выполняемые в приложении, настолько разнообразны, что имеют различные средства *отмены*. Например, перемещение прямоугольника можно рассматривать как размещение четырех строк, каждая в одну линию. Паттерн Команда требует, чтобы независимо от этого разнообразия, вложенные компоненты реализовывали простой интерфейс, и поэтому с ними можно было бы обращаться единообразно.

Рис. 16.15 показывает проект отмены, который следует паттерну Команда. `RedoUndoManager` (менеджер восстановления/отмены) не знает, как каждый из его компонентов, размещенных в стеке, выполняет действия отмены или восстановления. Когда требуется отмена, он восстанавливает прежнюю величину из `stack1` (стек 1) и вызывает свой метод `undo()` (отменить). Сама отмена может быть достаточно сложной операцией, но она оставлена для выполнения компонентом.

Сложный компонент, вроде класса `MultiLine` (несколько строк), может далее использовать свойство `Undoable` (допускает отмену), чтобы выполнить свой метод `undo()`, как показано на рис. 16.15.

16.5. Пользовательский интерфейс для управления электронной почтой

В итерации 2 пользовательские интерфейсы не сильно отличаются от прототипа, показанного в главе 14. Имеются незначительные изменения из-за удаления или добавления некоторых элементов управления окном, чтобы сделать понятным и упростить GUI. Например, на рис. 16.16 был изменен текст на командных кнопках по сравнению с его прототипом на рис. 14.3. Ярлыки кнопок с единственным словом типа *Insert* (добавить) заменяют более длинный текст, предложенный в прототипе, типа *Insert New Message* (добавить новое сообщение). Добавлена очень важная кнопка *Exit* (выход).

Отображение исходящих сообщений на рис. 16.16 сортируется в убывающем порядке параметра `Scheduled Dtp` (намеченный отдел). Нажатие клавишей мыши по столбцу, содержащему имена, выполняет сортировку. При первоначальном отображении исходящих сообщений опции фильтрации по умолчанию запрещены. Нажатие мышью кнопки `Toggle Filter` (переключатель фильтра) делает нижерасположенные опции доступными.

Отображаются дополнительные сообщения, чтобы информировать пользователя относительно неполной информации, представленной в GUI, типа показанной на рис. 16.17. Пример ссылается на подпоток *Create New Message* (создать новое сообщение), представленный в главе 14 на рис. 14.6.

Добавление исходящего сообщения, использующее диалоговое окно, показанное на рис. 14.6, вызывает ряд интересных действий. Прежде чем диа-

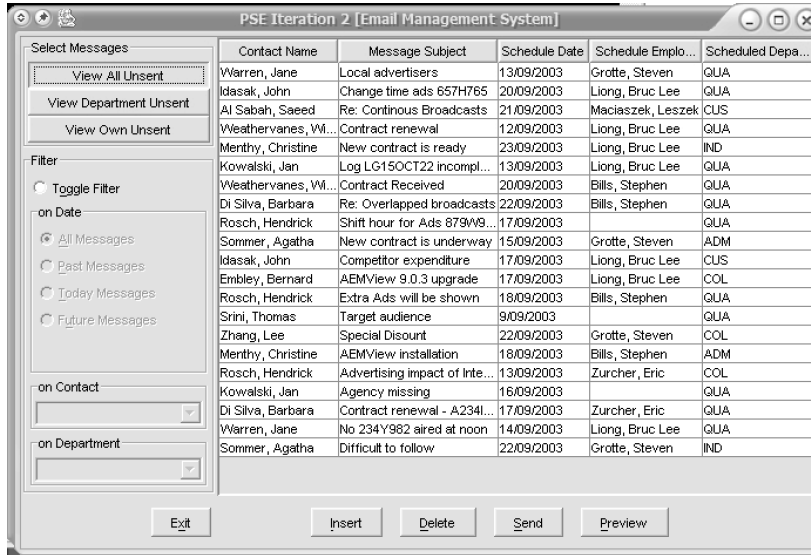


Рис. 16.16. Итерация 2: основной GUI

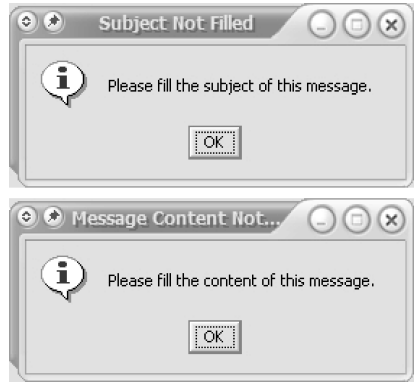


Рис. 16.17. Сообщения об ошибках для Create New Message (создать новое сообщение)

логовое окно можно показать на экране, программа должна построить заместителя исходящего сообщения и гарантировать, что исходящее сообщение зарегистрировано в `EIdentityMap` (класс «коллекция идентичности объектов» пакета `entity`). Рис. 16.18 показывает, что создание OID (идентификатор объекта) для нового исходящего сообщения обработано с помощью `MDataMapper` (класс «преобразователь данных» пакета `mediator`).

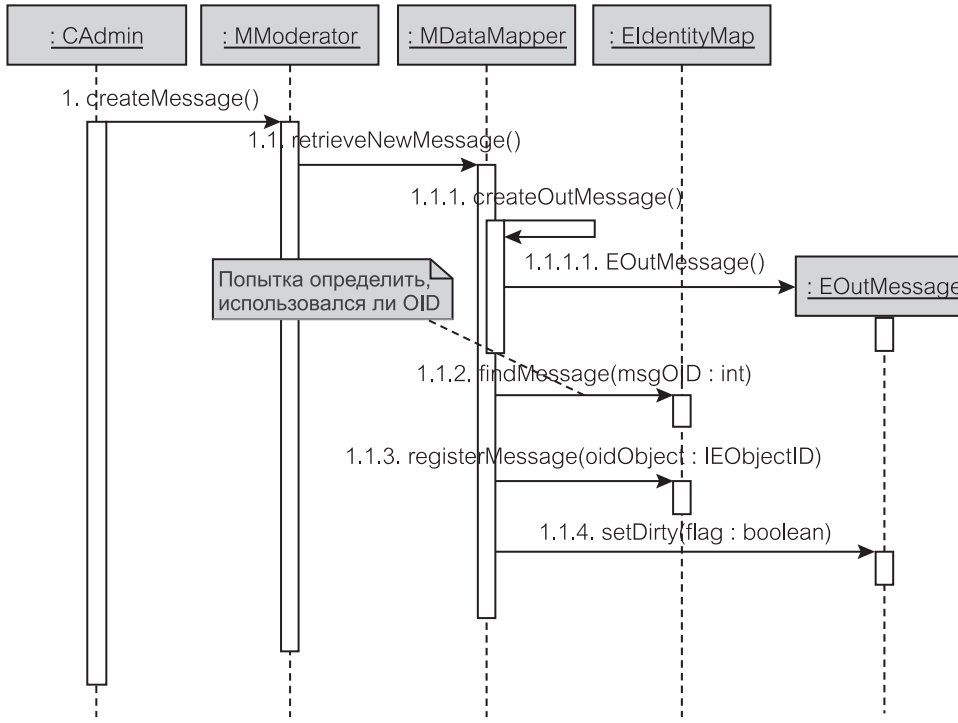


Рис. 16.18. Получение заместителя нового исходящего сообщения

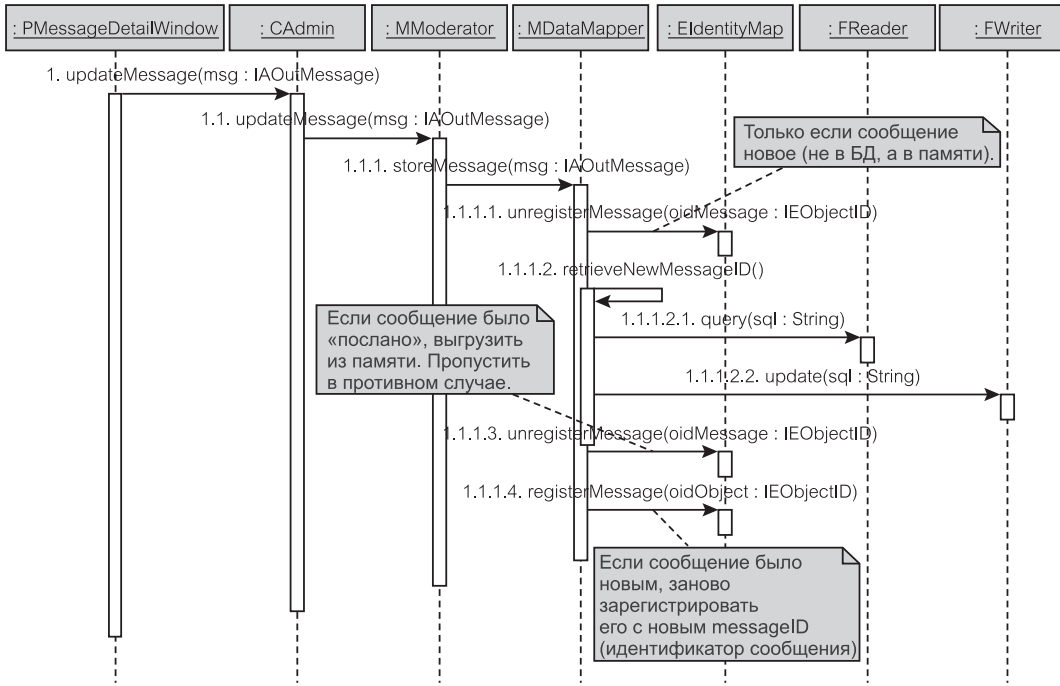


Рис. 16.19. Корректировка модифицированного исходящего сообщения

Далее новое исходящее сообщение автоматически помечается как измененное, чтобы указать, что его требуется включить в следующий цикл корректировки (или перезагрузки), выполняемый с помощью `EIdentityMap`. Как только заместитель исходящего сообщения получен, исходящее сообщение может быть изменено и обновлено с помощью `PMessageDetailWindow` — класс «окно деталей сообщения» пакета `presentation` (рис. 14.6).

Корректировка нового созданного исходящего сообщения происходит в конце выполнения подпотока *S5 — Create New Message* — создать новое сообщение (рис. 14.6). Рис. 16.19 показывает, что механизм сохранения совместим с паттернами *Коллекция идентичности объектов* (Identity Map) и *Преобразователь данных* (Data Mapper), рассмотренными в главе 15.

Рис. 16.19 иллюстрирует сценарий, где пользователь закончил редактировать исходящее сообщение в `PMessageDetailWindow` и хочет сохранить его в БД. Сохранение исходящего сообщения требует, чтобы `MDataMapper` синхронизировал содержание кэша с содержанием БД (поскольку данное исходящее сообщение имеет к этому отношение). В первую очередь, `MDataMapper` не должен регистрировать исходящее сообщение из `EIdentityMap`, если это сообщение только что было создано, то есть оно существует только в памяти, но его еще нет в БД.

Не выполнять эту регистрацию следует потому, что `MDataMapper` первоначально зарегистрировал исходящее сообщение в `EIdentityMap`, исполь-

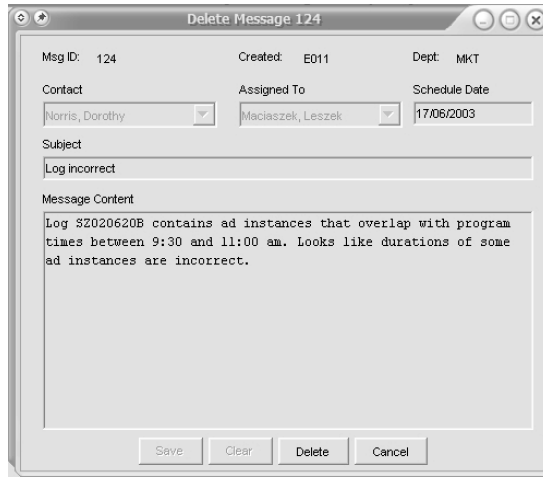


Рис. 16.20. Окно для удаления исходящего сообщения

зуя автоматически формируемый OID, а не определенный на основе свойств реального исходящего сообщения. Если исходящее сообщение не зарегистрировано, его соответствующий `messageID` (идентификатор сообщения) получается из БД до того, как это сообщение будет сохранено в БД. Получение `messageID` для нового исходящего сообщения завершает всю совокупность требуемых данных, необходимых для `EOutMessage`. Корректировка только что созданного сообщения может теперь быть завершена перерегистрацией в `EIdentityMap` со своими OID и `messageID`.

Другие изменения по сравнению с прототипом GUI, представленным в главе 14, касаются использования соответствующего диалога, чтобы подтвердить, что пользователь хочет удалить сообщение. Диалоговое окно, показанное на рис. 16.20, теперь заменяет окно, изображенное на рис. 14.7.

С этим новым окном пользователь информирован, какое исходящее сообщение предполагается удалить, и поэтому можно избежать ошибок. Поля в диалоговом окне не редактируются.

Имеются другие незначительные усовершенствования в проекте пользовательского интерфейса для итерации 2. Например, сообщение об ошибке, чтобы показать, что отдельные исходящие сообщения не были посланы, теперь иллюстрируется на рис. 16.21.

Рис. 16.21. Ошибка «Unable to Send» (невозможно послать)

Резюме

1. Проект пользовательского интерфейса (User interface — UI) должен учитывать две основные категории клиентов: программируемый клиент и клиент-браузер. *Программируемый клиент* предполагает, что программа находится и выполняется на клиенте и имеет доступ к ресурсам, хранящимся на клиенте. *Клиент-браузер* нуждается в сервере, чтобы загрузить требуемые данные и получить инструкции для предоставления данных в пользовательском интерфейсе на основе Web-технологии.
2. Кроме предоставления функциональных возможностей системы пользователю, пользовательский интерфейс должен удовлетворять ряду других требований, известных под термином *применимость*. Основные руководящие принципы проектирования пользовательского интерфейса: 1) пользователь в управлении, 2) непротиворечивость интерфейса, 3) снисходительность интерфейса и 4) адаптируемость интерфейса.
3. *Библиотеки классов* пользовательского интерфейса содержат «оконные» компоненты и определяют впечатление и ощущение от использования приложения. В мире Java библиотека классов называется Java™ Foundation Classes (JFC). Большая часть JFC состоит из набора GUI-компонентов по имени Swing. Компоненты Swing могут быть сгруппированы в контейнеры, меню и элементы управления.
4. *Контейнеры* определяют основные вид и поведение приложения. Они представляют прямоугольные области на рабочем столе пользовательского интерфейса и содержат другие компоненты, включая другие контейнеры, меню и элементы управления. Окна, диалоговые окна, части окон и панели — примеры контейнеров. Swing использует «менеджер расположения», чтобы разместить компоненты внутри контейнера.
5. Окна и некоторые другие компоненты могут содержать *меню*. Компоненты меню используются, чтобы реализовать строки меню и их элементы. Выключатели и кнопки-переключатели могут рассматриваться как виды элементов меню. Панели инструментов могут также рассматриваться как меню.
6. *Элементы управления* представляют модель событий пользовательского интерфейса. Они делятся на командные кнопки и другие элементы управления. Обычные кнопки, кнопки-переключатели, выключатели и элементы меню — все они являются разновидностями командных кнопок. Другие элементы управления включают окна со списками элементов для выбора нескольких или только одного элемента (комбинированная строка ввода), полосы прокрутки и ползунки. Один из наиболее универсальных и полезных элементов управления — представление дерева контейнеров и элементов.
7. *Модель событий* библиотеки Swing получена из MVC-шаблона. Обработка пользовательским интерфейсом событий должна соответствовать паттерну Наблюдатель.
8. Обычно используемые паттерны в разработке пользовательского интерфейса: Наблюдатель, Декоратор, Цепочка обязанностей и Команда.

9. Пользовательский интерфейс итерации 2 учебного примера EM состоит из первичного окна с различными элементами управления, но без строки меню и элементов меню. Пользовательский интерфейс включает также множество вторичных окон (диалоговых окон и окон сообщений).

Ключевые термины

Обзорные вопросы

1. Программируемый клиент или клиент-браузер могут представлять пользовательский интерфейс приложения. Что это за клиенты? Какие опции развертывания должны они предлагать?
2. Понятие интерфейса используется в программной инженерии в разных контекстах и с разным смыслом. Каковы популярные использования слова «интерфейс» в программной инженерии?
3. Который из четырех основных принципов проектирования пользовательского интерфейса должен быть, по вашему мнению, наиболее важным при переходе от процедурного к объектно-ориентированному программированию? Обоснуйте вашу точку зрения.
4. Программирование пользовательского интерфейса с использованием библиотеки Swing требует объединения легких и тяжеловесных компонентов. Как это воздействует на мобильность приложений Java?
5. Каковы различия между первичным и вторичным окном? Может ли приложение иметь несколько первичных окон? Может ли первичное окно быть модальным?
6. Что такое управление расположением и управление выделением уровней? Связаны ли эти две концепции? Объясните.
7. Каково различие между выключателями и кнопками-переключателями? Объясните на примерах.
8. Каково различие между комбинированной строкой ввода и окном со списком? Объясните, как и когда эти два компонента используются?
9. Объясните все «за» и «против» различных вариантов реализации классов подписчика.
10. Объясните различия между паттернами Цепочка обязанностей и Команда. Как должны эти паттерны применяться в проектировании пользовательского интерфейса?
11. Обсудите использование паттернов в проекте окна на рис. 16.10 (дублированного для удобства ниже на рис. 16.22). Рассмотрите следующие паттерны: Наблюдатель, Посредник, Декоратор и Команда.

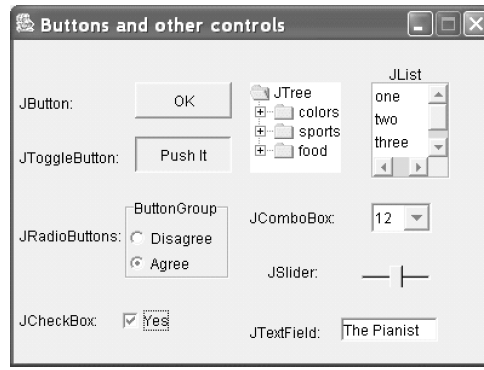


Рис. 16.22. Кнопки и другие элементы управления

Примеры задач

1. Изобразите диаграмму последовательности действий, чтобы показать, как может быть реализована операция `Toggle Filter` — переключатель фильтра (рис. 16.16). Предположите, что вы имеете одну модель таблицы для размещения всех требуемых для показа данных.
2. Изобразите диаграмму последовательности действий для удаления исходящего сообщения в EM-приложении. Примените паттерны `Identity Map` (коллекция идентичности объектов) и `Data Mapper` (преобразователь данных), объясненные в главе 15. Какие интересные проблемы будут обнаружены?
3. Объясните, почему рис. 16.15 «Получение нового заместителя исходящего сообщения» не показывает, как получаются OID — идентификатор объекта и PK — первичный ключ (то есть, `messageID` — идентификатор сообщения) нового сообщения. Как это сделано?
4. Изобразите диаграмму последовательности действий, чтобы показать, как `PMessageTableModel` (класс «модель таблицы для сообщений» пакета `presentation`) может обеспечить функциональные возможности `Filter by Contact` (фильтр по деловому партнеру) для EM.
5. Итерация 2 учебного примера EM определяет, что данные, отображаемые в главном окне, следует отсортировать по некоторым столбцам. Предположим, что эту сортировку можно выполнить с помощью нажатия мышью заголовка таблицы. Как это может быть реализовано? Используйте паттерн Наблюдатель (раздел 9.3.4). Изобразите диаграмму последовательности действий и покажите соответствующий фрагмент кода.
6. Рис. 16.23 представляет окно, реализованное для обеспечения сценария использования «Создание нового сообщения» из подпотока S5, рис. 14.6. Что здесь плохо с точки зрения пользовательского интерфейса? Как это можно улучшить?

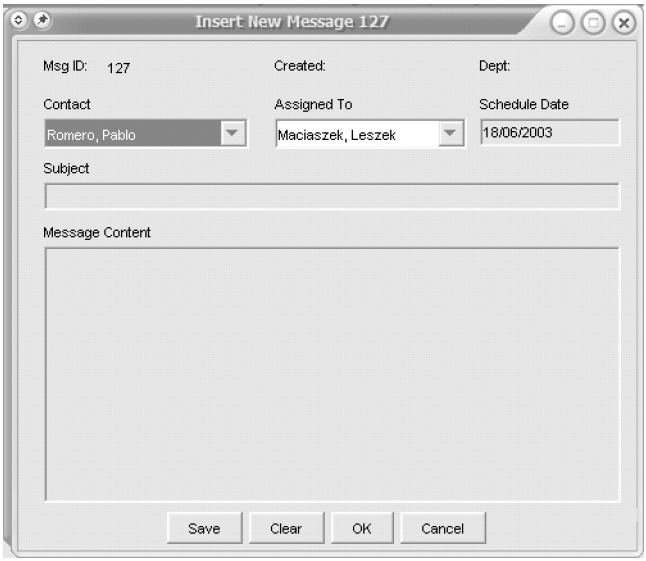


Рис. 16.23. Окно «Создание нового сообщения»

Глава 17

Проектирование и программирование пользовательского интерфейса на основе Web-технологии

Очевидная тенденция в разработке промышленных приложений заключается в создании систем, позволяющих использовать Web-технологию. Web-приложения вездесущи — они могут использоваться в любое время и отовсюду. Ясно, что есть очень немного ситуаций, когда применение Web-приложения не является привлекательным. К сожалению, проблемы при создании таких приложений значительны. Наиболее серьезные проблемы, связанные с ПО сервера, в этой главе не рассматриваются. Данная глава сконцентрирована на ПО клиента.

Web-приложение означает, что браузер Интернета управляет представлением содержания пользовательского интерфейса, но бизнес-логика и БД находятся на сервере. Это прежде всего означает, что используется **язык разметки гипертекстов** (HyperText Markup Language — HTML). **Web-страница** в формате HTML есть соединение содержания представления (например, некоторого текста) и инструкций представления (например, размер шрифта). HTML может использоваться, чтобы применить инструкции предоставления текста в стандартных компонентах пользовательского интерфейса, включая компоненты библиотеки Swing. Это может улучшить представление в приложениях, которые не обязательно допускают Web-технологию.

Компонент `JApplet` в Swing обеспечивает контейнер пользовательского интерфейса, который может быть помещен на Web-страницу. Апплеты Java и JavaScript (язык сценариев скриптов на основе Java) — всего лишь две технологии, которые реализуют в пределах браузера динамический пользовательский интерфейс. Это означает, что пользовательский интерфейс, основанный на Web-технологии, может быть изменен динамически и в смысле расположения, и в смысле содержания. HTML использует признак <object> (объект), чтобы загрузить клиенту объект, который находится по адресу URL (Uniform Resource Locator — унифицированный указатель информационного ресурса). **Апплет** может быть таким объектом в HTML-странице, извлеченным из Web-сервера.

По соображениям безопасности апплет может быть помечен в цифровой форме. Апплет располагается в **«песочнице»** (sandbox) — области памяти, где располагаются апплеты во время выполнения загружаемого кода. Эта область называется *«песочницей»*, поскольку она ограничена и имеет ограниченный доступ к ресурсам системы. Апплет не может иметь доступ к ресурсам системы, если ему не задано явное разрешение. В технических терминах апплет Java состоит из одного или нескольких *JavaBeans* (бинов Java) — ком-

понентов многократного использования, которые хранятся в сжатой форме в **файле архива Java** (Java Archive file — JAR). Это улучшает время загрузки.

Серверные технологии для Web-приложений включают сервлеты и серверные страницы Java (Java Server Pages — JSP). **Сервлет** представляет собой Java-код, который налету создает HTML-страницы и который управляется Web-сервером. Код может быть поддержан серверными страницами Java (JSP). Web-сервер обеспечивает канал связи между HTML-браузерами клиента и сервером БД. Будучи загруженным на Web-сервер, сервлет может соединяться с БД и обслуживать ее соединение больше чем с одним клиентом. Формирование *цепочки сервлетов* разрешает сервлету передавать запрос клиента к другому сервлету.

Если сервлет является кодом Java с вложенными HTML-элементами, то **серверная страница Java** (Java Server Page — JSP) в противоположность этому является HTML-страницей со вложенным Java-кодом (*теги* или *скриплеты*), чтобы управлять динамическим содержанием страницы и снабжать ее данными. JSP динамически добавляется к сервлету до запуска.

Теги фактически являются заместителями для разработанного кода Java, так что они могут быть многократно использованы (раздел 17.4.8). Теги — лучшее предложение, чем скриплеты. **Скриплеты** смешивают Java-код с содержимым Web-страницы и поэтому находятся на границе уровней `presentation` (представление) и `control` (управление). По этой причине они фактически не могут повторно использоваться[1].

Современная Web-технология для пользовательских интерфейсов гарантирует более строгое разделение уровней `presentation` и `control`. Теги и управляемые метаданными шаблоны, основанные на XML (eXtensible Markup Language — расширяемая спецификация языка, предназначенного для создания страниц WWW), обеспечивают это разделение в конце представления (вида). Технология **Struts** проекта Jakarta (раздел 17.4.9), поддержанная XML, усиливает это разделение в конце уровня `control` (использование контроллера), одновременно предлагая ряд тегов многократного использования для уровня `presentation`.

Связанная с этим технология в конце `presentation` (не рассматриваемая здесь) — **JavaServer Faces** (JSF) [46]. Идея состоит в том, чтобы определить в J2EE (раздел 3.3) кое-что вроде библиотеки Swing (глава 16) для Web-технологии пользовательских интерфейсов.

С простыми **тонкими Web-клиентами** вся бизнес-логика для приложения выполнена на сервере. Запрос Web-страницы посылается Web-серверу. Web-сервер выполняет запрос и посылает результат назад клиенту. В этом месте в большинстве случаев соединение между клиентом и сервером завершается.

---

[1] Скриплет или скриптлет представляет собой фрагмент Java-кода, соответственным образом оформленный. Само название — scriptlet — произошло от сочетания двух слов: script и applet. (*Прим. перев.*)

С **толстыми Web-клиентами** можно выполнить некоторую бизнес-логику на клиенте. Хороший, хотя и необязательный, пример выполнения бизнес-логики на клиенте — проверка входящей информации от пользователя.

Компоненты типичной Web-системы делятся, по крайней мере, на три *уровня*: Web-клиент, Web-сервер и сервер БД. В некоторых случаях желателен отдельный сервер приложения. В этой главе рассматриваются возможные технологии для уровня Web-клиента и уровня Web-сервера. Она также объясняет транзакции в системах Интернета без сохранения адресов; другие технологии для сервера БД обсуждаются в других местах книги (главы 10, 20 и 21). Технологии для сервера приложения описаны в главе 22.

Технологии — это изменяемая цель. Принципы и паттерны имеют более длительное применение. Данная глава повторно рассматривает паттерны, объясненные ранее в книге, и обсуждает их с точки зрения Web-приложений. Представлены также некоторые паттерны, которые непосредственно используются в Web-системах. Глава заканчивается описанием варианта сервлета учебного примера EM.

17.1. Допустимые технологии для уровня Web-клиента

С точки зрения пользователя **клиент** — это то, что пользователь видит работающим на автоматизированном рабочем месте. Пользователь может и не знать, происходит ли выполнение фактически внутри автоматизированного рабочего места или отдаленно на некоторой машине сервера. Точка зрения инженера ПО на разделение клиента/сервера, конечно, другая. В любом случае **клиент/сервер** (client/server — C/S) является логическим делением. *Клиент* запрашивает услуги сервера, а **сервер** обеспечивает эти услуги для авторизованного клиента. Расположение компонентов клиента и сервера вторично.

Технологии, рассматриваемые в этом разделе как инструменты реализации для уровня Web-клиента, — это HTML, языки использования сценариев и апплеты. Объяснения других интересных технологий могут быть найдены в книгах, посвященных этой теме (например, [94]).

17.1.1. Основы HTML

Язык разметки гипертекстов (HyperText Markup Language — HTML) — наиболее общий формат, разработанный, чтобы представлять Web-страницы в сети. Страницы, размещенные в формате HTML, используют стандартные *теги форматирования* и, как результат, позволяют организовать надлежащее отображение Web-браузерами.

HTML следует спецификации **World Wide Web Consortium** (W3C) — Консорциум Мировой паутины [110]. Этот консорциум выпускает различные спецификации, чтобы гарантировать стандартизацию Web-технологий и их реализацию. Несмотря на усилия стандартизации, разработчики (продавцы) Web-браузеров могут реализовать свои собственные слегка измененные версии спецификации. Термин «слегка» возможно преуменьшен, поскольку обычно известно, что Web-страница будет иметь различные вид и поведение при использовании различных Web-браузеров.

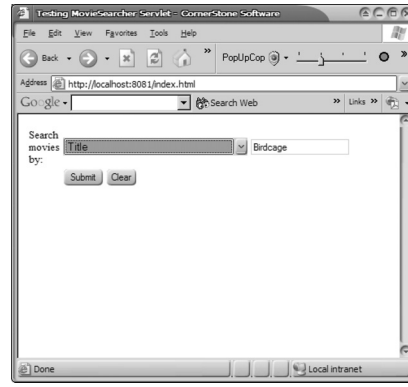


Рис. 17.1. HTML-документ, отображенный браузером

Рис. 17.1 показывает HTML-документ, отображенный Web-браузером. Это — простой пример, который реализован для доступа к БД MovieActor (фильм-актер). Пример показывает, что пользователь может выбрать условие поиска (типа `Title` — заголовок) и напечатать в строке поиска значение (типа `Birdcage` — Клетка для пташек), чтобы найти детали кинофильма.

Листинг 17.1 содержит пример HTML-документа, соответствующего представлению, показанному на рис. 17.1. Он включает основные теги для обработки входной информации.

Листинг 17.1. Образец HTML-документа

Образец HTML-документа

```
1:    <html>
2:    <head>
3:    <!-- комментарий: только пример заголовка -->
4:    <title> Testing MovieSearcher Servlet </title>
5:    <body>
6:      <form method="post" action="/SearchMovie">
7:      <table border=0 width="70%">
8:        <tr><td>Search movies by:</td>
9:          <td><select name="searchBy">
10:           <OPTION value="TitleV">Title
11:           <OPTION value="ActorV">Actor
12:           <OPTION value="DirectorV">Director
13:         </select></td>
14:         <td><input name=searchField type=text></td>
15:       </tr>
16:       <tr><td></td><td><input type="submit" value="Submit">
17:       <input type=reset value="Clear"</td></tr>
18:     </table>
19:     </form>
20:   </body>
21:   </html>
```

HTML-документ должен быть начат с тега <html> и закрыт тегом </html> (строки 1 и 21). HTML-документ может использовать необязательный *раздел заголовка* <head>, чтобы информировать Web-браузер о своих *метаданных* (данных относительно данных, то есть, информационных данных относительно самого документа). Тег <title> может использоваться в разделе заголовка, как указано в строке 4. *Комментарии* могут находиться в любом месте документа, если они помещены внутри тегов <!-- и --> (строка 3).

Большинство **HTML-тегов** должно быть закрыто указанием символа / в имени тега, как, например, в </html>. Некоторые теги, типа <p>, который обозначает начало параграфа, или <option> (строка 10), не требуется явно закрывать. Тег может иметь несколько атрибутов, позволяющих *настройку тега*, например, method (метод) и action (действие) являются атрибутами тега form (форма) (строка 6).

Тело <body> HTML-документа определяет, какие данные должны быть показаны внутри рамки Web-браузера. Большинство тегов HTML применимо только в пределах этого тега.

Обычно тегом, используемым, чтобы представить данные в табличном формате, является <table> (строки 7–18). Данный тег *таблицы* в строке 7 определяет атрибут border (граница) с размером 0, указывающим, что не нужно изображать никакой границы. Это можно видеть на рис. 17.1. Данные в таблице организованы по *строкам* (<tr>, строка 8). Каждая строка может иметь несколько *столбцов* (<td>, строка 8), но обычно для всех строк имеется одно и то же число столбцов.

Эта глава рассматривает только HTML-теги, необходимые для размещения в *форме* входной информации, получаемой от пользователя. Данная ситуация обозначается тегом <form> в строках 6–19. Поля формы будут переданы серверу или внешним источникам, когда пользователь закончит работу с формой.

Тег form имеет два типа атрибута method (метод) — post (отправить), чтобы указать, что данные, собранные в форме, должны быть отправлены серверу, и get (получить), чтобы указать, что данные получаются с сервера. Строка 6 показывает, что данные будут отправлены серверу.

Сервер URL, который форма будет вызывать, указывается в атрибуте action (действие). Это обычно CGI-скрипт, сервлет/JSP или другая программа Web-страницы, способная обрабатывать данные. Строка 6 в листинге 17.1 показывает, что процессор формы управляется *сервлетом* по имени SearchMovie (искать фильм), расположенным в корневой папке сервера (обозначено символом /).

Входные величины могут быть получены с помощью выпадающих списков (тег <select> — строка 9 и тег <input> — строки 14, 16, 17). Каждый из выбранных и заполненных входов передается серверу. Сервер опознает вход по его имени. Например, выпадающий список имеет имя searchBy (искать по...). В случае списка, расположенного в листинге 17.1, серверу будет послана только отобранная пользователем величина, как обозначено тегом <option>. Например, если пользователь выбирает Actor (актер) из выпадающего списка, то searchBy связывается со списком величин Actor.

Имеются и другие типы входных величин. Листинг 17.1 показывает только `text` (текст), `submit` (представить) и `reset` (переустановить). Тип `text` указывает, что входная величина должна быть помещена в текстовое поле. Тип `submit` указывает, что требуется кнопка, которая должна быть показана вместо этого тега `input`. Кнопка `submit` вызовет передачу данных, полученных в форме, по Web-адресу `action` формы. Тип `reset` позволяет повторно установить поля формы в их первоначальное состояние, обычно или пустое, или выбираемое по умолчанию.

Другие типы входа — `button` (кнопка), `radio` (переключатель), `hidden` (скрыть) и т. д. Тип `button` отображает на экране кнопку, которую пользователь может нажать. Тип `radio` представляет кнопку-переключатель с невыбранным состоянием. Тип `hidden` — атрибут, при котором сервер может запомнить некоторые величины, не измененные пользователем. Переустанавливаемая величина просто передается серверу для хранения.

В этой главе не говорится, как элементы на Web-странице могут быть организованы в страницу, приятную для рассмотрения. Это можно было бы сделать, оптимизируя атрибуты различных тегов (типа атрибута `align` — выравнивание), а также форматированием тегов (типа `<center>`). Использование тега `frame` (структура) здесь также не демонстрируется. Тег `frame` позволяет отображать коллекции Web-страниц вместе в виде единственной структуры.

17.1.2. Язык скриптов

Языки скриптов используются в HTML-документах, чтобы расширить возможности взаимодействия пользователя. **Скрипты** типа VBScript и JavaScript используются, чтобы выполнить быструю проверку входной информации, простую мультипликацию и другие операции, чтобы сделать Web-страницу «живой». Скрипты, если они используются, должны быть объявлены в заголовке HTML-документа. Эта глава содержит пример JavaScript, используемый для проверки входной информации.

JavaScript был представлен компанией Netscape. Спецификации JavaScript доступны в сжатом файле компании Netscape [72]. Microsoft прелагает изменения к этому скрипту, известные как **JScript**. Третий язык скриптов, поддерживаемый спецификацией HTML — **VBScript**. Это — вариант Visual Basic (VB) компании Microsoft. Документацию и JScript, и VBScript можно найти на Web-сайте Microsoft [68].

Листинг 17.2 показывает `JavaScript` (строки 3–16) чтобы проверить, является ли входная информация, введенная пользователем (строка 20), в пределах диапазона 0–100 (строка 18). Скрипт помещен в раздел заголовка документа. Он «выстреливает», как только пользователь нажмет мышью представленную кнопку, как показано в обработчике события `onSubmit` (представленный) — строка 18. *События* представляют собой часть атрибутов в пределах различных элементов формы. Они могут быть помещены в любой из входных элементов, которые доступны для изменения пользователем. Типы скриптов объявлены на Web-страницах вместе с атрибутом `script language` — язык скриптов (строка 3).

Листинг 17.2. Пример JavaScript

Пример JavaScript

```
1:      <HEAD>
2:        <TITLE> Form Validation </TITLE>
3:        <SCRIPT LANGUAGE="JavaScript">
4:          <!--
5:            function checkIt(int lower, int upper) {
6:              var strval = document.sampleForm.percent.value;
7:              var intval = parseInt(strval);
8:               if ( lower < intval && intval < upper ) {
9:                 return( true );
10:             } else {
11:              alert("Input " + strval + " is out of "+
                    lower+"-"+upper+" range");
12:              return( false );
13:             }
14:           }
15:         <!--  конец скрипта -->
16:       </SCRIPT>
17:     </HEAD>
18:     <FORM NAME="sampleForm" METHOD="post"
              ACTION="/SearchMovie" onSubmit="checkIt(0,100)">
19:           Percentage given:
20:           <INPUT TYPE="text" NAME="percent" VALUE="1">
21:           <INPUT TYPE="submit">
22:     </FORM>
```

Как и в большинстве языков скриптов, JavaScript *статически анализирует* синтаксические ошибки. Однако JavaScript не компилирует код, как, например, языки Java или C/C++. У компилируемых языков обычно используется **статическое связывание**, чтобы установить все ссылки программы во время компиляции.

В JavaScript ссылки связываются динамически. Они определяются по мере необходимости во время выполнения программы. Пример **динамического связывания** в программе, использующей скрипты, — модификация строки `onSubmit="checkIt(0,100)"` (строка 18) на `onSubmit="checkItWrong (0,100)"`. Web-страница будет отображена точно так же, но на странице будет отображаться ошибка, когда будет нажата кнопка.

Гибкость динамического связывания делает JavaScript легким устройством, которое может избежать излишней проверки кода. Добавление и удаление кода также может быть сделано легко, облегчая, таким образом, пошаговую разработку кода. Отрицательным моментом является то, что некоторые ошибки не могут быть обнаружены до времени выполнения, когда пользователь активизирует операторы скрипта (например, при нажатии на кнопку, как показано в листинге 17.2, строка 21).

Продавцы Web-браузеров предлагают несколько отличающиеся реализации скриптов даже притом, что они все объявляются «соответствующими» стандарту. Это заставляет Web-разработчиков так разрабатывать Web-страницы, чтобы быть уверенными, что они будут работать должным образом на разных браузерах.

Реализация JavaScript также достаточно хитра. Например, более ранние версии Web-браузеров (которые поддерживают более ранние спецификации HTML) не понимают тег `<script>` (скрипт) и поэтому игнорируют его и переходят к следующему доступному тегу. Именно поэтому тело скрипта в листинге 17.2 «защищено» тегом *комментария* (строки 4 и 15). Когда более ранний браузер встречает тег комментария, он игнорирует тело скрипта и поэтому не будет вызывать рассматриваемую ошибку. Более новый браузер выяснит, что тег комментария (строка 4) не имеет соответствующего тега, закрывающего комментарий, и проигнорирует всю строку 4.

Скрипт в листинге 17.2 будет выполнен только тогда, когда пользователь нажмет на кнопку `submit` — представить (действие `onSubmit`). Скрипт выполняется через вызов `checkIt()` (проверить это) в строке 18. Функция `checkIt` (строка 5) проверяет значение входного текста, представленного строкой 20, чтобы выяснить, содержит ли она правильную величину (как требуется параметрами `lower` — нижняя граница и `upper` — верхняя граница функции `checkIt()`).

JavaScript может обращаться к текущей Web-странице через ее объекты. Это показано в строке 6, где скрипт проверяет `value` (значение) текстового поля по имени `percent` — процент (объявленного в строке 20). Значение возвращается в виде строки, которую нужно преобразовать в целое число через вызов встроенной функции `parseInt()` (анализ целого числа). Если входная величина правильная (строки 8–9), то скрипт возвращает `true` — истина (строка 9), иначе он отображает окно сообщения, указывающее на неправильную входную величину, и возвращает `false` — ложь (строка 12).

Когда обработчик события `onSubmit` завершается, вызывая скрипт, он решает, продолжить ли выполнение события, как это запланировано, или остановить выполнение. Это — главная причина, почему скрипт должен возвратить величину `true` или `false`. Величина `true` указывает, что все проверено, и выполнение может продолжаться, посылая в этом случае данные, полученные от формы, к `SearchMovie`, как указано атрибутом `action` — действие (строка 18). Величина `false` отменяет вызов `SearchMovie`.

17.1.3. Апплет: тонкий и толстый

Апплеты могут быть подразделены на тонкие и толстые. Обе категории представляют множества классов, которые составляют приложение. Различие состоит в том, сколько кода находится на клиенте и сколько на сервере (рис. 17.2). И тонкие, и толстые апплеты имеют своих серверных коллег. Однако **тонкий апплет** исполняет большую часть своих операций на сервере, а **толстый апплет** исполняет большинство своих операций на клиенте.

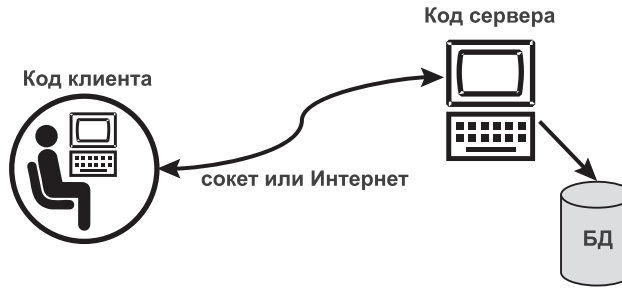


Рис. 17.2. Апплет-серверное соединение

Тонкий апплет подразумевает, что в компьютер клиента будет загружено минимальное количество кода. Это требует мощной поддержки от сервера, чтобы обеспечить полные функциональные возможности. Из-за меньшего размера кода клиента, тонкий апплет загружается намного быстрее и поэтому предпочтителен для медленного соединения Интернета.

Преимущества тонкого апплета обычно являются недостатками толстого апплета, и наоборот. Ниже приведены некоторые характеристики тонкого апплета:

- Он загружает машину клиента быстрее, так как код клиента состоит только из классов уровня `presentation` (представление).
- Он требует мощной обработки на сервере, поскольку классы уровня `presentation` не делают много вычислений/логики (соответственно, на сервере необходимы дополнительные классы).
- Он отделяет классы уровня `presentation` от классов, расположенных на сервере. Изменения с обеих сторон (код клиента или сервера) обычно не воздействуют на другую сторону, если изменения не существенны и не затрагивают функциональные возможности приложения.
- Тонкие апплеты более трудно обслуживать, так как имеются два возможных места модификации: код сервера и клиента.
- В нем более трудно обнаружить ошибки, так как они могут находиться как в коде клиента, так и в коде сервера. Может оказаться трудно определить, имеются ли проблемы в коде сервера или ненадежно соединение сокета/Интернета.

Листинг 17.3 объясняет подмножество *методов апплета*. Метод `init()` (инициализировать) является местом размещения всех методов, имеющих отношение к GUI. Метод `start()` (начать) активизирует программу и отвечает на действия пользователя (например, запуск потоков). Метод `stop()` (остановить) останавливает выполнение операций апплета, типа остановки потоков отображения пользовательского интерфейса. Метод `destroy()` (уничтожить) освобождает все ресурсы и готовит апплет к выгрузке из системы.

Листинг 17.3. Резюме методов апплета

`void`	**`destroy()`**
	Вызывается браузером или средством просмотра апплета, чтобы сообщить этому апплету, что он отработал и что он должен уничтожить все ресурсы, которые в нем размещены.
`void`	**`init()`**
	Вызывается браузером или средством просмотра апплета, чтобы информировать этот апплет, что он должен быть загружен в систему.
`void`	**`start()`**
	Вызывается браузером или средством просмотра апплета, чтобы информировать этот апплет, что он должен начать свое выполнение.
`void`	**`stop()`**
	Вызывается браузером или средством просмотра апплета, чтобы информировать этот апплет, что он должен завершить свое выполнение.

Рис. 17.3 показывает **модель переходов между состояниями апплета**. Апплет осуществляет переходы между состояниями по требованию браузера. Первоначально браузер вызывает метод `init()` апплета, чтобы инициализировать его после того, как будет вызван конструктор апплета. Как только апплет будет инициализирован, он может перейти в состояние `active` (активное). Апплет активизируется через вызов его метода `start()`. Конструирование апплета выполняется перед его инициализацией. Это основная причина, почему конструирование пользовательского интерфейса не может быть помещено в конструктор апплета. В этот момент апплет еще полностью не загружен в систему. Когда апплет находится в состоянии `started` (запущенное), он остается в этом состоянии, пока открыта Web-страница.

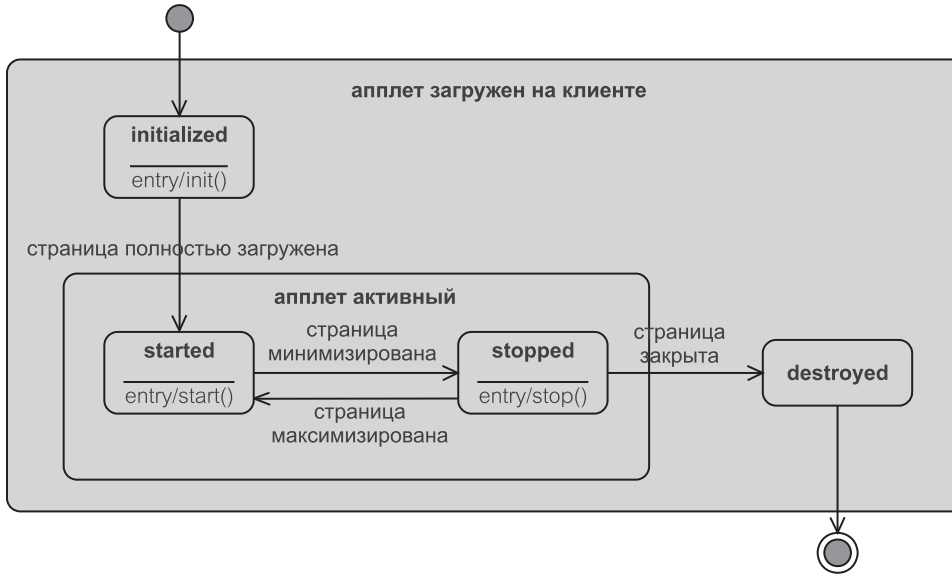


Рис. 17.3. Модель переходов между состояниями апплета

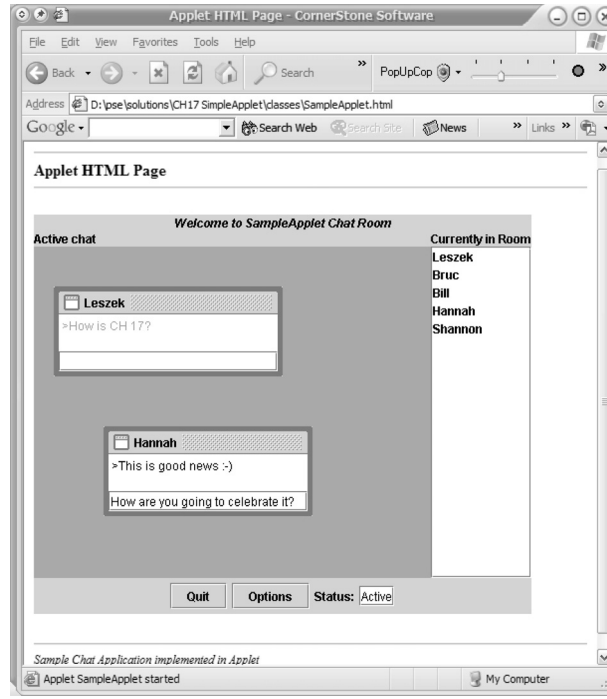


Рис. 17.4. Пример простого апплета

Имеются моменты, когда апплет изменяет свое состояние на `stopped` (остановленное). Они зависят от способа, которым апплет реализован, но обычно данный переход происходит, когда апплет минимизирован или когда он невидим на экране. Состояние `stopped` позволяет приложению экономить ресурсы системы, получаемые бездействующими апплетами. Когда апплет больше не нужен, как, например, в случае, когда Web-страница закрыта, он переходит в состояние `destroy` (уничтоженное). В этот момент апплет освобождает все системные ресурсы, которые он получил.

Рис. 17.4 показывает пример апплета. Этот апплет обеспечивает механизм диалогового взаимодействия (чата), посредством которого пользователи соединяются с центральным сервером и разговаривают со своими друзьями. Более сложные варианты такого апплета реализованы в ICQ (I Seek You — система интерактивного взаимодействия), MSN (Multiservice Network — многофункциональная сеть) и других механизмах диалогового взаимодействия, которые используют Web-технологию. Комбинация апплета и языка скриптов, как представляется, лучшая пара для такого типа приложений. Другие примеры обычных приложений, которые используют апплеты, — банковские приложения, контролирующие акции через Интернет, и подобные приложения, которые требуют обширной обработки на сервере и меньше — на клиенте.

Выполнение апплета на Web-странице помещается в очень ограниченную оболочку, называемую моделью «песочницы». Модель **«песочницы»** — на-

бор правил, ограничивающий доступ апплета к своей централизованной системе. Это предотвращает апплет от обращения к уязвимым ресурсам централизованной системы. Если бы ему позволяли иметь неограниченный доступ к памяти и ресурсам ОС, кто-то со злым умыслом мог бы повредить систему. Все апплеты помещаются в «песочницу», где они создаются и выполняются. Они могут иметь доступ только к ресурсам в ограниченных пределах «песочницы». Так называемые **доверенные апплеты** выполняются вне «песочницы» согласно данным им привилегиям.

Модель «песочницы» Java основана на трехуровневой системе защиты, которая представляет собой следующее:

1. *Устройство проверки байт-кода* — весь код Java компилируется и проверяется во время компиляции; это гарантирует, что только коду, который соответствует спецификации языка Java, будет позволено работать в виртуальной машине Java.
2. *Загрузчик классов апплета* — загрузчик классов апплета загружает код апплета; он предотвращает замену важных Java API-классов другими классами.
3. *Менеджер безопасности* — менеджер безопасности ограничивает операции, выполняемые апплетом; он разрешает некоторые операции и ограничивает опасные.

Короче говоря, апплету не позволяется:

- читать данные из файловой системы централизованной машины (включая файлы, свойства и т. д.);
- записывать или удалять файлы;
- соединяться с сетевым портом на любой машине кроме HTTP-сервера, с которым он работает;
- выполнять или загружать код других программ/библиотек/DLL.

Единственный путь для доступа апплета к ресурсам централизованной системы — лишь если разработчик предоставит разрешение апплету. Когда **подписанные апплеты** (имеющие подпись) загружены, пользователь должен дать апплету разрешения. Апплет не будет выполнен, если пользователь не позволит это. Эта глава не рассматривает полностью задание подписи апплету, поскольку данная процедура зависит от реализации браузера. Шаги, необходимые, чтобы подписать апплет, слегка отличаются для Internet Explorer, Sun и Netscape.

Версия апплета итерации 2 для учебного примера EM — **толстый апплет**. Преобразовать уже существующий код итерации 2 в толстый апплет достаточно просто, благодаря PCMEF-структурному проектированию. Преобразование выполнено на классе `PWindow` (класс «окно» пакета `presentation`), так что он становится наследником класса `JApplet` (класс «апплет» пакета Swing) вместо `JFrame` (класс «структура» пакета Swing). Рис. 17.5 показывает апплет, запущенный на `AppletViewer` (просмотрщик апплетов). Функциональные возможности этого апплета те же самые, что и в итерации 2 Java-приложения, рассмотренные в этой части книги.

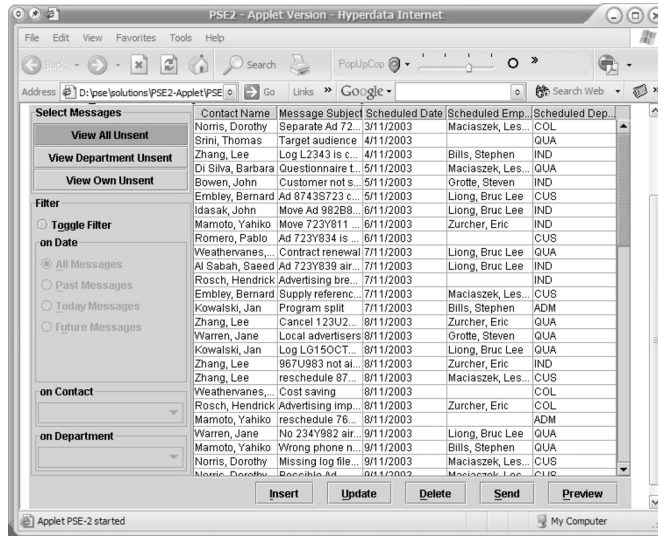


Рис. 17.5. EM-итерация 2 в виде апплета

17.2. Допустимые технологии для уровня Web-сервера

Уровень **Web-сервера** управляет связью между Web-клиентами и бизнес-логикой приложения. Сама *бизнес-логика* может быть реализована внутри Web-сервера или внутри специально предназначенного сервера приложения, например, использующего EJB-технологию. Другие типичные функции уровня Web-сервера включают [94]:

- представление динамического содержания на Web-страницах, включая HTML, графику, изображения, звук и видео;
- управление потоком экрана у Web-клиентов (то есть, последовательностями, в которых могут быть отображены экраны);
- поддержание транзакций клиентов.

Ранней технологией Web-сервера, поддерживающей динамическое содержание клиентов, была *Common Gateway Interface* (CGI — общий шлюзовой интерфейс). Эта технология теперь заменена более эффективными J2EE-технологиями сервлетов и JSP-страницами. Далее рассматриваются эти две технологии.

17.2.1. Сервлет

Различие между апплетом и сервлетом в том, что **сервлет** полностью расположен на сервере. По существу у сервлета нет никакого кода клиента. Клиент — это HTML-страница или другие страницы, типа JSP или ASP. Сервлет — это сервис, предназначенный для запуска на машине сервера, чтобы обслуживать многих клиентов. Он является Java-кодом, скомпилированным и развернутым на Web-сервере.

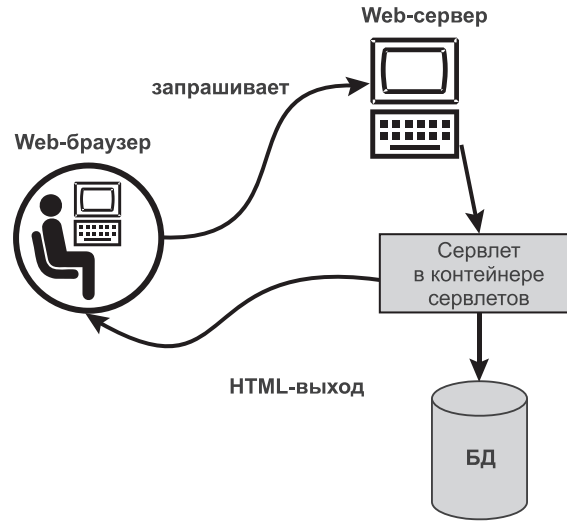


Рис. 17.6. Связь сервлета

Термин «сервлет» (servlet) происходит от термина «серверный апплет» (server applet). Он ожидает запросы от клиентов, будь то HTML-клиент или JSP, и соответственно отвечает. Рис. 17.6 показывает, как Web-сервер отвечает на запрос от компьютера клиента (HTML). Web-сервер имеет **контейнер сервлетов** (или *Web-контейнер*), который отвечает за нахождение, загрузку и выполнение требуемого сервлета.

Сервлет имеет доступ к ресурсам сервера, включая БД. Затем он обычно возвращает результат в формате HTML. Фактически результат мог бы быть и в другой форме, но для целей этой главы принят формат HTML.

Следующий список представляет преимущества сервлета по сравнению с другими технологиями, типа CGI:

- Более строгая проверка типов (сервлет должен быть скомпилирован и развернут, в то время как CGI анализируется только по мере необходимости).
- Позднее связывание любых ссылок (контейнер сервлетов легко реализует стратегии, направленные на улучшение скорости обращения к сервлету с помощью позднего связывания ссылок, то есть связывания во время выполнения, когда это действительно необходимо).
- Более богатое взаимодействие с сервером (технология сервлета допускает несколько параллельных путей выполнения его на своем сервере в противоположность технологии CGI, которая допускает только единственный путь).
- Эффективность (накопитель сервлетов поддерживается Web-контейнером технологии, и эта совокупность обычно полностью оптимизирована, в то время как CGI-процессы создаются, только когда возникает CGI-запрос; эти процессы используют потоки ОС и поэтому более дороги в создании).
- Более мощное управление входом и выходом (технология сервлета обеспечивает обширные функциональные возможности, чтобы обрабатывать и

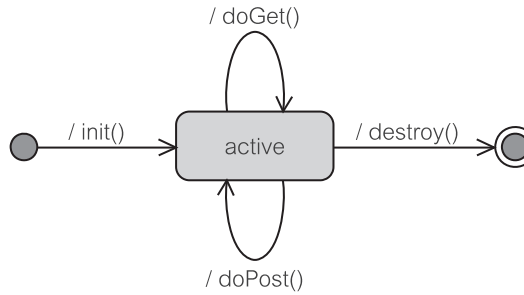


Рис. 17.7. Модель активного состояния сервлета

анализировать входы/выходы; разработчики могут использовать стандартный Java API, чтобы получить соответствующий результат).

- Портативность (сервлет может перемещаться между различными Web-серверами и контейнерами, если только они соответствуют J2EE-спецификации).
- Безопасность от переполнения буфера и других воздействий (Java обеспечивает проверку различных проблем безопасности, типа проверки «за границами массива», которая должна быть сделана вручную CGI-разработчиком).

Модель активного состояния сервлета показана на рис. 17.7. Состояния сервлета полностью управляются его контейнером. Контейнер сервлетов гарантирует, что в нем находится достаточно сервлетов, чтобы обслужить клиентов. Если совокупность сервлетов может переполнить контейнер, из него удаляются сервлеты, не нужные в настоящее время, чтобы сохранить ресурсы. Сервлет переходит в состояние `active` (активное), когда контейнер сервера (Web-сервер) вызовет его метод `init()` (инициализировать). Когда он в состоянии `active`, могут быть обработаны различные запросы (HTTP-запросы), так же, как и передачи HTTP. Когда сервлет перестает использоваться, он обычно удаляется контейнером, вызывая его метод `destroy()` (уничтожить). Это позволяет сервлету освободить все свои ресурсы.

Контейнер сервлетов (Web-сервер) поддерживает различные стратегии обслуживания совокупности активных сервлетов. Один сервлет может обслуживать нескольких клиентов и не поддерживает состояние конкретных клиентов. Рис. 17.8 иллюстрирует, как запрос перехватывается сервером и переадресуется контейнеру сервлетов. Контейнер сервлетов гарантирует, что сервлет загружен (операция `load servlet` — загрузка сервлета) и инициализирован (операция `invoke servlet` — обращение к сервлету). В этот момент сервлет находится в состоянии `running` (выполнение). Обращение к выполняемому сервлету возвращает выходной результат через HTTP-ответ. Как ранее упоминалось, выходной результат может иметь любой тип, обычно это — HTML-результат.

Сервлет может работать с запросами от различных клиентов, типа HTML, апплета, JSP и т. д. Рис. 17.9 — пример выходного результата сервлета для запроса, показанного на рис. 17.1. Он перечисляет детали запрошенного кинофильма в формате таблицы.

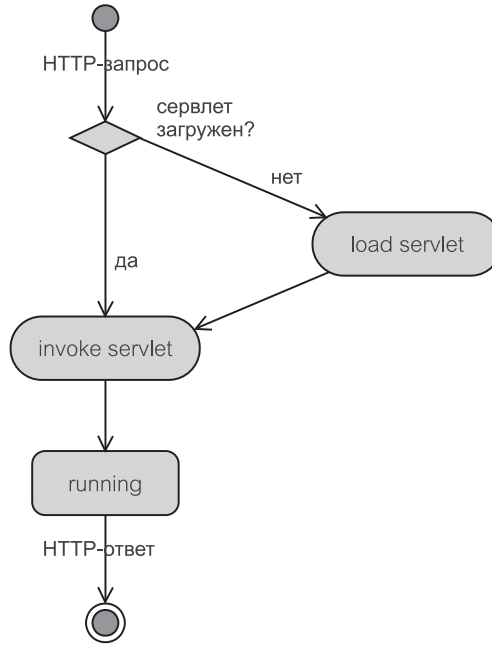


Рис. 17.8. Обращение к сервлету

Рис. 17.9. Сервлет SearchMovie (поиск фильма)

17.2.2. JSP

Вторая обычно используемая J2EE-технология — **Java Server Pages** (JSP — серверные страницы Java). JSP — вариант технологии сервлетов. JSP отделяет бизнес-логику от представления. Бизнес-логика обеспечивается сервлетом, в то время как представление реализуется с помощью JSP. JSP представляет путь связи между клиентом и сервером (рис. 17.10) — здесь не так уж много отличий от того, что показано для сервлета на рис. 17.6.

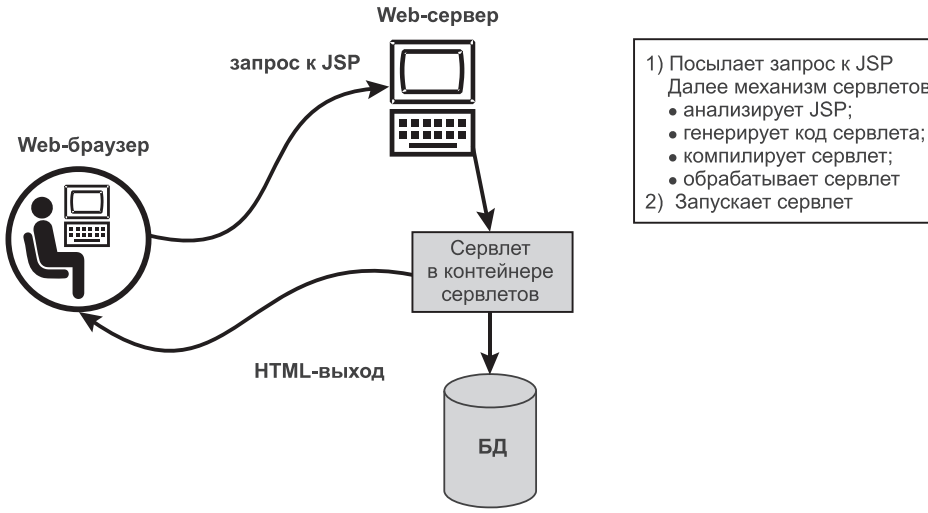


Рис. 17.10. Связь JSP

Главное различие заключается в том, что *запрос JSP* требует пары дополнительных предварительных шагов, прежде чем его можно использовать. JSP-страница должна анализироваться механизмом сервлетов (контейнером сервлетов). Эта операция генерирует код сервлета, который должен быть скомпилирован и инициализирован контейнером. Как только сервлет будет готов, происходит обычное обращение, подобное обращению к сервлету на рис. 17.8.

JSP разделяет большинство преимуществ сервлета и не обеспечивает никаких новых способностей, которые не могли бы быть выполнены с сервлетом. Однако отделение представления от содержания обеспечивает разделение задач. В типичном сценарии динамическое содержание может быть получено сервлетом. Оно затем *форматируется* JSP-страницей, которая будет отображена на клиенте.

Листинг 17.4 показывает пример JSP-страницы. Строки 1–4 содержат *теги-директивы*. JSP-страница может расширять, импортировать или устанавливать конкретные свойства системы, используя теги-директивы. Страница может использовать Java API, как и сервлет или приложение Java. Строки 5–54 показывают типичные HTML-теги, смешанные с JSP-тегами.

JSP-объявления могут быть помещены в пределах *тега объявления* `<%! %>` (строки 9–44). Альтернативно то же самое объявление может быть помещено в *тег-скриплет* `<% %>`.

Листинг 17.4 представляет функции инициализации и уничтожения JSP-страницы (`jspInit()` и `jspDestroy()` соответственно). Необходимое соединение создается в функции `jspInit()` (строки 12–18) и закрывается в `jspDestroy()` (строки 19–22). Функция `displayMoviesByLeadingActors()` (отобразить фильмы по ведущим актерам) извлекает данные из соединения, созданного во время инициализации сервлета, и отображает данные в формате таблицы (строки 32–42) в HTTP-параметре ответа (строка 23).

Листинг 17.4. Пример JSP

Пример JSP

```
 1:  <%@page contentType="text/html"%>
 2:  <%@page import="movie.*" %>
 3:  <%@page import="java.util.*" %>
 4:  <%@page import="java.io.*" %>
 5:  <html>
 6:  <head><title>Movie JSP</title></head>
 7:  <body>
 8:  <%-- declare and put basic function --%>
 9:  <%!
10:     movie.Connection conn;
11:     Exception error = null;
12:     public void jspInit(){
           //переопределение инициализации JSP
13:        try{
14:           conn = new Connection();
15:        }catch(Exception exc){
16:           error = exc;
17:        }
18:     }
19:     public void jspDestroy(){ //обеспечить соответствующую
                                  //версию ресурса
20:        conn.close();
21:         conn = null;
22:     }
23:     public void
        displayMoviesByLeadingActors(ServletResponse response)
24:        throws Exception{
25:        PrintWriter out = response.getWriter();
26:        if(error != null){
27:           out.println("<h1>Error has occurred</h1>");
28:           error.printStackTrace(out);
29:           return;
30:        }
31:        out.println("<h1>The database content</h1>");
32:        out.println("<table border=2>");
33:        out.println("<tr><th>Movie Title</th><th>Main
               actor</th> <th>Director</th></tr>");
34:        Collection listedAs =
               conn.getLeadingActorsByQuery();
35:        Iterator it = listedAs.iterator();
36:        while(it.hasNext()){
37:           ListedAs la = (ListedAs) it.next();
38:           Movie m = la.getMovie();
39:           Actor a = la.getActor();
```

```
40:            out.println("<tr><td>"+m.getMovieTitle()+
                "</td><td>"+a.getActorName()+"</td><td>"+
                m.getDirector()+"</td></tr>");
41:         }
42:         out.println("</table>");
43:      }
44:  %>
45:
46:  <%-- use below to include javabeans --%>
47:  <%--<jsp:useBean id="listedAs" scope="session"
     class="movie.ListedAs" /> --%>
48:  <%-- <jsp:getProperty name="listedAs"
          property="getLeadingActorsByQuery" /> --%>
49:  <%--<jsp:include page="/DisplayMessageContent.jsp"
     flush=true /> -->
50:  <%-- this requested the display of all movies --%>
51:  <% displayMoviesByLeadingActors(response); %>
52:  <p>Click <a href="index.html">here</a> to go to main page.
53:  </body>
54:  </html>
```

JSP может использовать **JavaBeans** и **Enterprise JavaBeans** (EJB), как указано в строках 47 и 48 (при этом тег комментария `<!--` и `-->` удаляется). JavaBeans — множество классов Java, которые подчиняются предопределенным правилам, позволяющим наблюдать и модифицировать их свойства. EJB подобен JavaBeans, но поддерживает возможность обращения к распределенной бизнес-логике и распределенной БД. EJB полностью управляется контейнером сервера (обычно *сервером приложения*).

Атрибут `id` (идентификатор) в строке 47 используется в текущей странице как имя ссылки (имя переменной). Атрибут `class` (класс) задает полное имя класса в JavaBeans. Атрибут `scope` (область действия) задает условия, при которых JavaBeans является активным. В результате все это определяет *состояние* Web-приложения. Область действия может иметь следующие значения:

- page (страница) — Bean-компонент доступен только в текущей странице;
- request (запрос) — Bean-компонент активен только для текущего запроса;
- session (сеанс) — Bean-компонент активен для текущего сеанса;
- application (приложение) — Bean-компонент активен от начала до конца приложения, включая, если необходимо, многократные сеансы.

Атрибут `scope` используется контейнером сервлетов, чтобы эффективно подсчитывать надлежащую стратегию обслуживания ряда сервлетов для запросов клиента. Контейнер сервлетов должен гарантировать, что состояние сервлета будет соответствовать области действий. Если область действия сервлета — сеанс, то контейнер должен гарантировать, что данный сервлет будет существовать во всем сеансе.

Строка 48 листинга 17.4 показывает, что свойство JavaBeans может быть запрошено и изменено через JSP-тег или обычное обращение к функции Java. Строка 48 эквивалентна единственному вызову функции Java: `listedAs.getLeadingActorsByQuery();`.

Строка 49 показывает директиву `include` (включение) в JSP. Директива `include` может использоваться, чтобы включить другую страницу в текущую страницу. Это полезно при разработке сложных Web-страниц. JSP позволяет с помощью такого подхода повторно использовать страницы.

Строка 51 вызывает функцию, объявленную в строке 23. Если функция в строке 23 возвращает строку вместо непосредственного отображения HTTP-ответа, то скриплет (`<% %>`, строка 51) должен быть изменен на тег выражения (`<%= %>`). Тег выражения позволил бы отобразить выходную величину метода как часть его HTML-кода.

К глобальным переменным можно обращаться в JSP-странице. На эти переменные должны быть ссылки из JSP-тегов, как упомянуто выше. Одна такая переменная — `response` (ответ), которая соответствует HTTP-ответу текущей страницы (строка 51). Глобальные переменные позволяют текущей JSP-странице взаимодействовать со средой ее сервера. Другие переменные, типа `exception` (исключение), используются, чтобы обслуживать состояния или события, которые возникают между JSP-страницами.

Рис. 17.11 показывает результат выполнения кода из листинга 17.4. JSP основана на использовании тегов и поэтому ее сила реализуется в виде повторного использования тегов. JSP-спецификация обеспечивает стандартную библиотеку, называемую *Custom Tag Library* (библиотека пользовательских тегов), которая может использоваться в разработке и повторном использовании уже существующих тегов для различных Web-страниц.

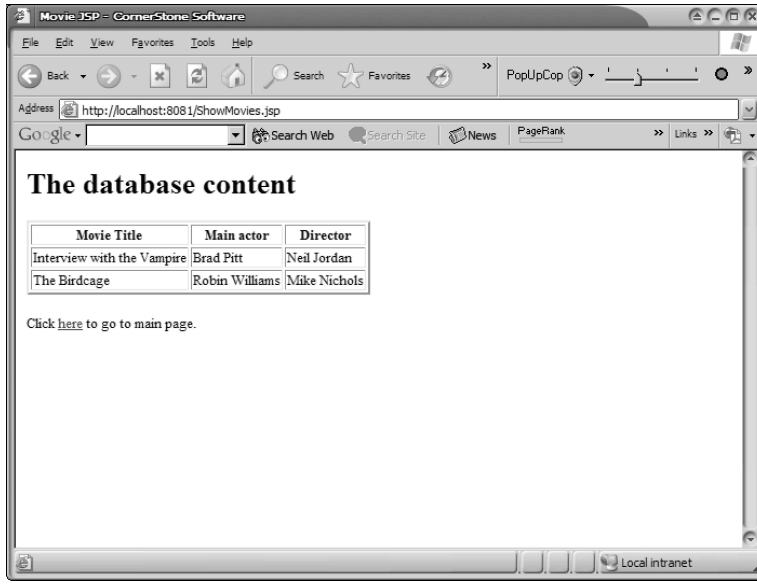


Рис. 17.11. HTML-выход JSP-кода

17.3. Транзакции Интернет-систем, не имеющих состояний

Системы, разработанные для Интернета, ограничены тем, как реализован Интернет. Интернет-системы являются **системами, не имеющими состояний**. Они базируются на серверных приложениях, которые обслуживают запросы без соединения (различающихся) последовательных запросов от одного и того же клиента. Интернет-системы создаются не имеющими состояний, так как они полностью основаны на **pull-технологии** (технологии «вытягивания»). Данные должны быть получены (вытянуты) из сервера. Сервер непрерывно управляет ожидающими запросами клиента. У систем, не имеющих состояний, есть многочисленные преимущества, среди которых следующие:

- более простое программирование приложения сервера — приложению сервера не нужно помнить все возможные состояния взаимодействий, которые он может иметь с каждым клиентом;
- более простое сопровождение и дублирование приложения сервера — это приложение и его данные могут легко быть предметом распределения серверной нагрузки; многие серверы могут обслуживать одних и тех же клиентов приложения без необходимости, чтобы клиенты знали, как происходит идентификация сервера;
- устойчивость — большинство Интернет-систем обычно развернуто на многих серверах, а не на одном; одна и та же услуга может быть получена от различных серверов, что делает клиентов менее зависимыми от конкретного сервера.

Однако имеется много Web-приложений, где важно обеспечить **состояния транзакций** клиентов. Рассмотрим типичное приложение Интернет-бизнеса, где покупатели могут поместить различные товары в корзину для покупок (тележку для покупок) и продолжать рассматривать дополнительные товары перед окончательным размещением заказа (или отказом от размещения). Нужны такие приложения, которые запоминали бы содержание корзины, пока не принято решение сделать заказ. Состояния корзины нужно знать между различными взаимодействиями.

Тогда возникает вопрос, как не имеющая состояний Интернет-система может обслуживать состояния? Какие имеются пути преобразования не имеющей состояний системы в систему, **обладающую состояниями**? Не обязательно, но возможно потребуется, чтобы клиент помнил состояния.

Имеются два основных подхода к запоминанию состояний. Первый подход — *передавать состояние*, полученное в начальном запросе, последующим запросам. Второй подход — хранить состояния в *объекте-сеансе*, который получает клиент, когда он формирует первый запрос к серверу. Также в порядке вещей, если Интернет-система комбинирует оба подхода, чтобы максимизировать количество данных, которые она может запомнить.

HTTP-запрос в листинге 17.5 соответствует выходу сервлета SearchMovie (искать фильм), показанному на рис. 17.9. Листинг иллюстрирует подход, использующий **передачу состояния**. В этом коде предварительно полученные состояния передаются серверу в качестве *параметров*, которые нужно сохра-

нить для следующего запроса. Сервер отвечает на запросы в соответствии со значениями параметров. Поскольку списки параметров изменяются и растут в размере, приложение сервера становится более сложным.

Листинг 17.5. Состояния, передаваемые в качестве параметров

```
Состояния, передаваемые в качестве параметров

http://localhost:8081/SearchMovie?searchBy=ActorV&searchField
                                                    =Birdcage
```

Параметры в листинге 17.5 разделены амперсандом (`&`). Начало параметра обозначено знаком вопроса (`?`). HTTP-запрос передается к `SearchMovie` с двумя параметрами: параметру `searchBy` (искать по...) задается значение `ActorV` (актер V), а `searchField` (поле для поиска) — значение `Birdcage` (клетка для пташек).

Листинг 17.5 имеет ограничения, которые можно переадресовать для хранения состояний на основе сеанса:

- Становится неудобным, когда имеется много состояний, которые следует обслужить. Web-адрес становится слишком длинным и подверженным ошибкам во время программирования.
- Взаимодействия пользователя сложны в том смысле, что из любой отдельной страницы могут исходить несколько путей. Удаление параметра, который не потребуется в списке параметров в следующем запросе, может вызвать трудности позже, когда этот параметр потребуется использовать в последующих запросах. Часто невозможно получить эту информацию заново.
- Нельзя хранить составное или сложное состояние, типа состояний мультимедийных объектов. Такие состояния обычно неконвертируемы к простому текстовому формату, который можно передавать в качестве параметра.
- Некоторые параметры нельзя отображать из-за соображений безопасности, типа налогового номера файла или личного идентификационного номера.

Объект-сеанс может использоваться, чтобы запоминать состояния и устранять недостатки подхода, связанного с передачей состояния. Объект-сеанс создается, когда клиент соединяется с сервером, и поддерживается в течение всего соединения. Сервер может запрашивать и изменять состояния, сохраненные в объекте-сеансе, и поэтому может обеспечить ответы клиенту, учитывающие состояния.

Строка 1 листинга 17.6 показывает, как состояние клиента, определяемое предварительно размещенным `userID` (идентификатор пользователя), может быть извлечено из объекта-сеанса. В этом примере приложение проверяет, получил ли пользователь страницу обычным путем, то есть, через `userID`, известный объекту-сеансу. Приложение не будет продолжаться, если оно определит, что параметр `userID` равен `null` (строки 2–5).

Листинг 17.6. Состояния, размещенные во время сеанса

Состояния, размещенные во время сеанса

```
1: Object userID = request.getSession().getAttribute("userid");
2: if(userID == null){
3:   reportError();
4:    return;
5: }
6: request.getSession().setAttribute("root_page","/SearchMovie");
7:
8: String prevPage = (String) request.getParameter("ref_page");
```

Активный объект-сеанс может быть получен с помощью HTTP-запроса, как показано в строках 1 и 6. Состояния, размещенные в объекте-сеансе, идентифицируются своими названиями. Строка 6 показывает, что атрибут `root_page` (корневая страница) с величиной `/SearchMovie` размещен в сеансе.

Листинг 17.6 также показывает пример поиска параметра (строка 8). Параметр `ref_page` (ссылка на страницу) получается из HTTP-запроса. Сервер может получать только те параметры, которые передаются клиентом. Он не может передавать параметры назад клиенту.

В противоположность тому, как БД управляет транзакциями (глава 21), спецификация J2EE не требует, чтобы состояние сеанса было восстановлено после аварий или отказов [94]. Поэтому приложения, использующие J2EE-реализации, которые обеспечивают дополнительную восстанавливаемость, рискуют быть менее портативными.

17.4. Паттерны и Web-технология

Сегодня имеется огромное количество документации и литературы относительно паттернов ПО. Отделить «зерно от плевел» в этих джунглях паттернов нелегко. Подход к объяснению паттернов, принятый в этой книге, состоит в том, чтобы представить их в контексте обсужденных тем программной инженерии и рассмотреть только наиболее распространенные и важные паттерны.

Паттерны, рассмотренные на данный момент, включали Фасад (Façade), Абстрактную фабрику (Abstract Factory), Цепочку обязанностей (Chain of Responsibility), Наблюдателя (Observer), Декоратора (Decorator), Команду (Command), Посредника (Mediator), Коллекцию идентичности объектов (Identity Map), Преобразователя данных (Data Mapper), Загрузку по требованию (Lazy Load) и несколько других. Это популярные паттерны от **GoF** и **PEAA**-наборов паттернов, то есть, **Gang of Four** (Банды четырех) [32] и **Patterns of Enterprise Application Architecture** (паттерны структуры промышленного приложения) [31]. В контексте Web-технологии наиболее существенным является множество паттернов, известное как *Core J2EE patterns* (паттерны ядра J2EE) [2].

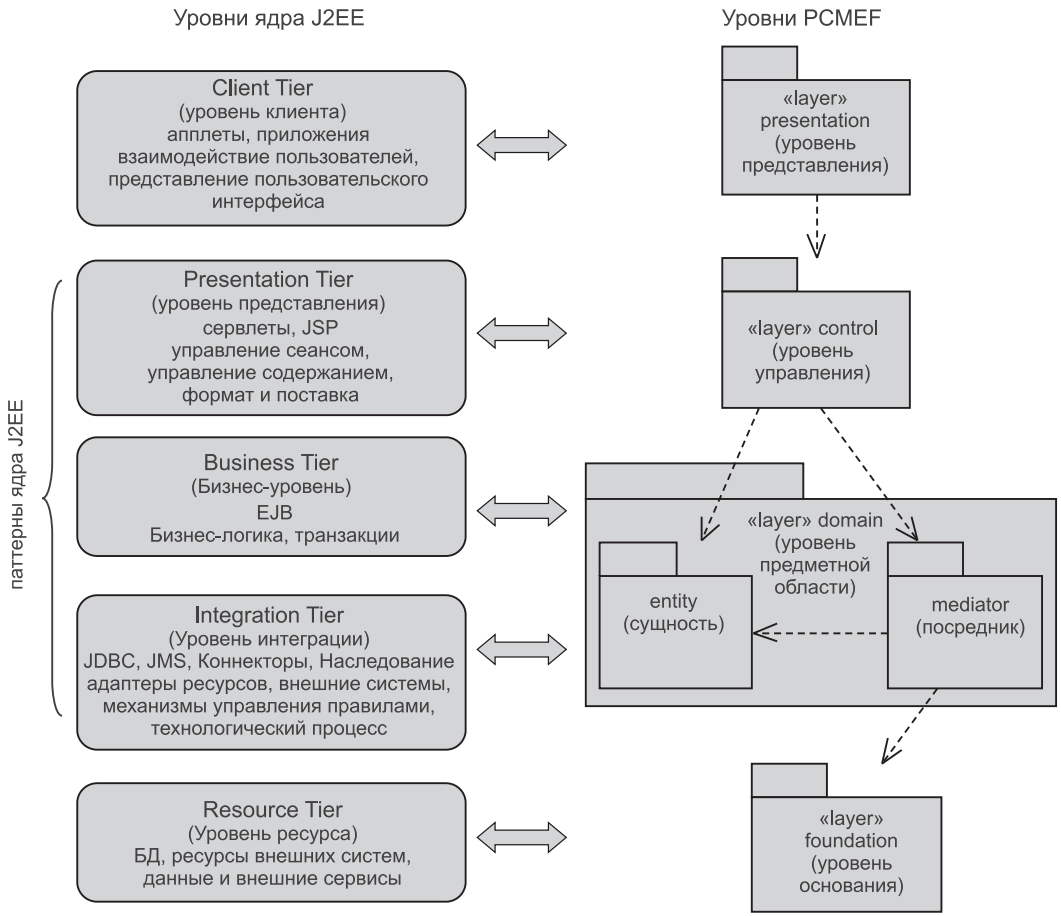


Рис. 17.12. J2EE-уровни и PCMEF-уровни

Паттерны ядра J2EE отражают лучшую практику программной инженерии для разработки с использованием J2EE-технологий, включая сервлеты, JSP и EJB, Web-сервисы, XML. Паттерны ядра J2EE расклассифицированы на уровни. Уровни ядра J2EE [2] хорошо соответствуют PCMEF-уровням, используемым в этой книге. Это иллюстрируется на рис. 17.12.

Вместо того чтобы представить весь новый диапазон паттернов для Web-технологии, этот раздел вводит только один новый J2EE-паттерн (Front Controller — Контроллер запросов) и концентрируется на обсуждении, как некоторые из ранее представленных паттернов могут использоваться с Web-технологиями. Здесь также объясняются новые технологии типа JSP-тегов и Struts (поддержки), чтобы проиллюстрировать использование этих паттернов. Паттерны, рассмотренные в этом разделе, — Наблюдатель (Observer), Компоновщик (Composite), Фабричный метод (Factory Method), Стратегия (Strategy), Декоратор (Decorator), MVC (Model-View-Controller — модель-представление-контроллер) и Контроллер запросов (Front Controller).

17.4.1. Наблюдатель

Паттерн Наблюдатель (Observer), уже рассмотренный в разделах 9.3.4 и 16.4.1, представляет собой множество *наблюдателей*, которые автоматически уведомляются относительно изменений в *субъекте*. Кажется, паттерн удовлетворяет приложениям, основанным на Web-технологии, но это, как ни удивительно, не всегда так. Рассмотрим приложение, основанное на Web-технологии, состоящее из кода сервлетов/JSP (субъекта) и набора страниц клиента (наблюдателей). Можно ожидать, что страницы клиента должны быть уведомлены, когда в сервере имеются новые корректировки. Это, однако, не может быть сделано в Интернет-системах, не имеющих состояний, поскольку сервер не имеет никакого контроля над своими клиентами.

Имеются способы обойти *отсутствие состояний* в Интернет-системах (раздел 17.3), но они все-таки не позволяют паттерну Наблюдатель выполнять свои функции из-за ограничений браузера. Особенность отсутствия состояний Web-приложений приводит к тому, что процесс уведомления об изменении состояний становится трудной задачей. Чтобы получить изменения, Web-браузерам нужно заново запросить состояние субъекта.

Java-приложения, использующие апплеты, обычно хорошо сочетаются с паттерном Наблюдатель. Рассмотрим апплет фондовой биржи, который используется, чтобы контролировать подъемы и падения стоимости конкретной акции. Эта отдельная система будет иметь серверное приложение, которое имеет список идентификаторов акций и ряд апплетов, связанных с сервером. Апплеты являются наблюдателями сервера. Сервер является субъектом. Сервер обслуживает список связанных апплетов и сообщает апплетам относительно изменений в ценах акций.

Общая реализация таких приложений заключается в том, чтобы вынудить апплеты клиента запрашивать сервер через фиксированные интервалы времени. Противоположная реализация могла бы заключаться в том, чтобы заставить сервер непрерывно передавать данные об акциях апплетам. Компромисс ищется между трудностью реализации и эффективностью. Последний подход заставляет сервер непрерывно работать, пока подключен апплет. Данные передаются апплетам независимо от того, контролирует ли пользователь цену конкретной акции или цены других акций. Первый вариант более эффективен, потому что с сервера требуются только данные об акциях, проверяемых в настоящее время пользователем. Сервер не должен хранить состояния апплетов и начинать предназначенные для обслуживания каждого апплета потоки. Апплет делает большую работу по опросу сервера в заданные интервалы.

17.4.2. Компоновщик

Паттерн Компоновщик (Composite) [32] определяет, что объект может быть построен как объединение других объектов. Это формирует иерархическое дерево объектов. Особенность этого паттерна заключается в том, что пользователь обращается как к индивидуальным объектам, так и к составным объектам одинаково. Чтобы обслужить запросы пользователя, может эффективно использоваться передача сообщения с помощью *делегирования* вниз по дере-

ву объединения. Составной объект делегирует полученные запросы к тому объекту-компоненту, который может их обслужить.

Соединение — естественное средство для организации страниц, показываемых с помощью Web-технологии. Web-страница может быть составлена из нескольких страниц благодаря HTML-*тегу структуры* (frame). Однако это несколько отличается от типичного использования Компоновщика при разработке обычных систем ПО. Поскольку клиент знает состав *сложного объекта* (структуры страниц), нет необходимости направлять запрос к нему. Он может послать сообщение непосредственно конкретному объекту, содержащемуся в структуре.

Компоновщик может использоваться для создания отдельных Web-страниц, как это делалось с помощью JSP. Разработчик может использовать JSP-директиву `<jsp:include>`, типа приведенной в листинге 17.4. Повторное использование Web-страниц достигается с помощью задания этой директивы `include` (включить). Однако в отличие от типичного использования Компоновщика, когда сложные объекты представляются на экране, JSP-директива `include` объединяет несколько страниц в единственную Web-страницу. Таким образом, JSP-директива `include` позволяет получить общий *вид и поведение* от всего Web-сайта.

Компоновщик обычно используется с другими паттернами типа Шаблона и Фабричного метода группы GoF. Паттерн *Шаблон* управляет размещением Web-страниц. Он может гарантировать, что общая Web-страница, типа заголовка компании или эмблемы, будет последовательно показана на всех Web-страницах. Паттерн *Фабричный метод* может использоваться, чтобы создавать или включать содержание Web-страниц в сложную Web-страницу, обеспечивая, таким образом, динамическое поведение на этой Web-странице.

17.4.3. Фабричный метод

Паттерн Фабричный метод (Factory Method) [32] определяет, что подклассы базового класса ответственны за создание *изделия*. Клиент знает лишь о существовании базового класса. Он вызывает метод базового класса, чтобы создать некоторые объекты (изделие). Однако базовый класс не знает, как создать изделие, и он делегирует запрос к переопределенному методу, реализованному его подклассом. Рис. 17.13 показывает пример Фабричного метода. Здесь видно, что только подклассы `Sub1` и `Sub2` знают, как создать `ProductA` и `ProductB` соответственно.

Этот паттерн широко используется в Web-приложениях для ситуаций, когда клиент запрашивает Web-страницу, и сервер создает эту страницу динамически. Приложение сервера (класс `Base` — базовый — в рассматриваемом случае) решает, какую Web-страницу следует показать. Решение, какую страницу показать, зависит от логики, заложенной в подклассы Фабричного метода (`Sub1` или `Sub2` на рис. 17.13). Это позволяет серверу создавать Web-страницы динамически. Клиент знает только одну точку входа к этим Web-страницам через класс `Base`.

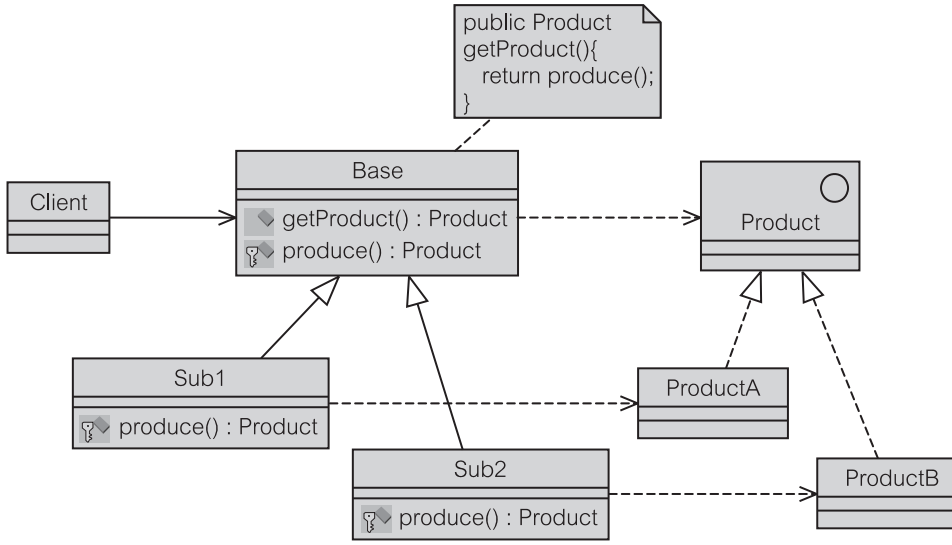


Рис. 17.13. Паттерн Фабричный метод

17.4.4. Стратегия

Паттерн Стратегия (Strategy) [32] является небольшой модификацией Фабричного метода. Стратегия может использоваться, чтобы определить Web-страницу (`View`), которая делегирует свою работу объекту `Controller` (контроллер) — рис. 17.14. Объект `Controller` знает *стратегии* для списка действий, которые могут быть выполнены в `Model` (модель). Объекты `View` (представление) и `Event` (событие) обеспечивают действия объекта `Controller`.

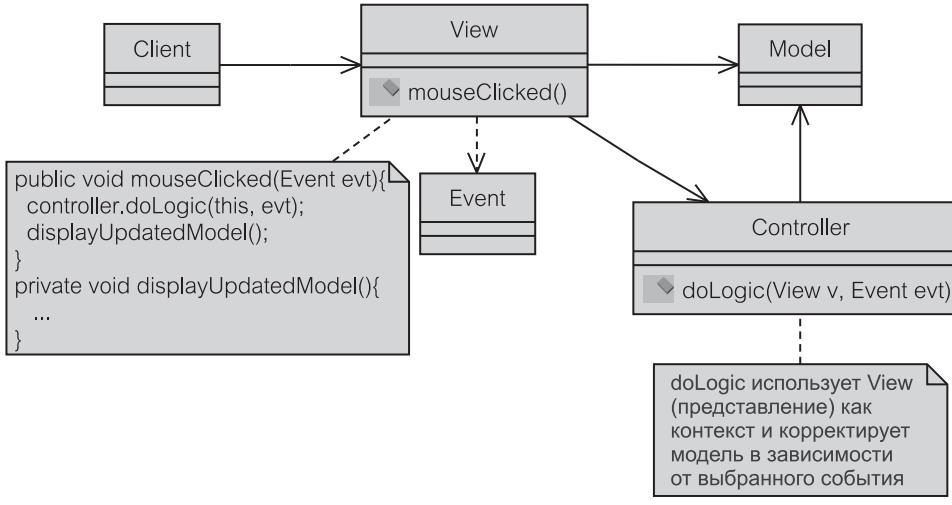


Рис. 17.14. Паттерн Стратегия

В Web-приложениях представление может быть Web-страницей или апплетом. Этот паттерн используется сервлетом SearchMovie (поиск фильма) — рис. 17.9.

17.4.5. Декоратор

Другая разновидность паттерна Компоновщик известна как **паттерн Декоратор** (Decorator) [32]. Этот паттерн определяет, что основной объект декорируют другие объекты. В типичном GUI главное окно *декорируется* полосами прокрутки, меню, панелями инструментов и другими элементами управления. Наличие элементов художественного оформления в классе `Window` (окно) можно выбирать.

Главное отличие паттерна Декоратор от паттерна Компоновщик заключается в том, что главный объект (`Window`) и его элементы художественного оформления знают друг друга, в то время как такая информированность в паттерне Компоновщик однонаправлена. Сложный объект знает составляющие его объекты (и делегирует к ним), но не наоборот.

Использование Декоратора в Web-системах включает применение внутри HTML языка скриптов. Спецификация стандарта HTML не позволяет использовать элементы художественного оформления. С помощью языка скриптов возможно украсить Web-страницу полосами прокрутки и другими элементами управления, которых нет в HTML.

17.4.6. Model-View-Controller (MVC)

Паттерн MVC (раздел 9.2.1) — Model-View-Controller (модель-представление-контроллер) имеет несколько отличное использование в Web-приложениях. Интернет-системы не имеют состояний и по этой причине они не допускают традиционного использования MVC. Традиционный паттерн MVC требует использования View (представления), чтобы переадресовать все запросы к Control (управление), которое затем изменяет Model (модель). View и Model находятся в отношении, определяемом паттерном Наблюдатель, так что View наблюдает Model и уведомляется относительно изменений Model.

Однако в Web-приложениях MVC не может обеспечить преимуществ уведомления Model. В HTML невозможно реализовать паттерн Наблюдатель (если только приложение не является апплетом) — раздел 17.4.1. Это вынуждает сделать некоторую модификацию использования паттерна MVC, как показано на рис. 17.15. Объединение JSP и EJB может обеспечить этот тип реализации MVC.

Класс `Control` на рис. 17.15 является сервлетом HTTP (поэтому он обозначается как HS, что означает HTTP servlet). `Model` представляется с помощью JavaBeans или Enterprise JavaBeans (EJB). `Views` — представления (обозначаемые как client pages — страницы клиента) являются множеством JSP-файлов, которые отображают данные, извлеченные из `Model`. События пользователя или команды представлены в форме методов `post` (отправить) или `get` (получить) HTTP, управляемых с помощью `Control`. `Control` является сервлетом, который обслуживает многие представления.

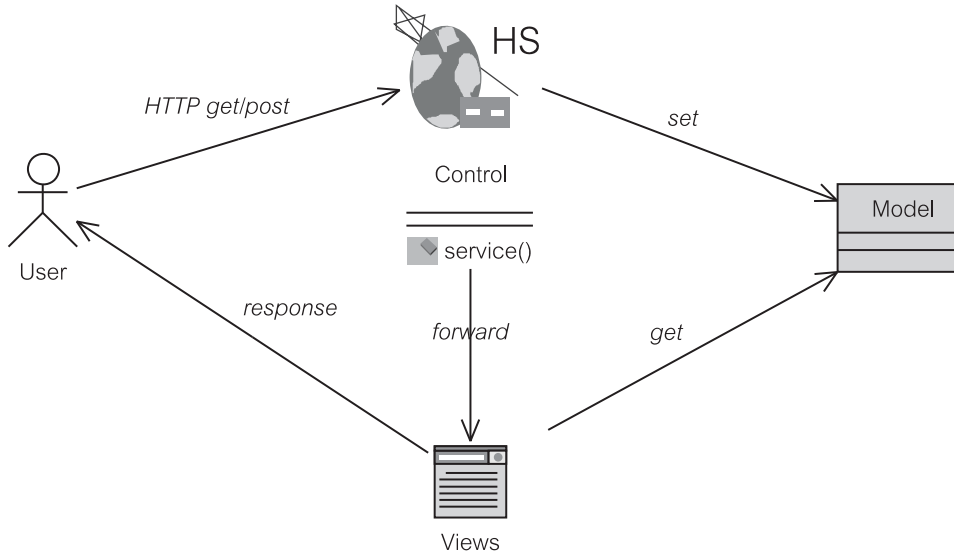


Рис. 17.15. Web-паттерн MVC

В этом модифицированном паттерне MVC не используется классическое отношение «один к одному» между `Control` и `View`. `Control` реализован как паттерн `Front Controller` (Контроллер запросов), рассмотренный в следующем разделе. Сам класс `Control` может реализовать либо паттерн `Strategy` (Стратегия), либо паттерн `Factory Method` (Фабричный метод), чтобы решить, что делать с запросами или какое `View` отобразить.

17.4.7. Контроллер запросов

Типичное Web-приложение имеет единственную точку входа, обслуживаемую сервлетом. Пользователь должен знать только один-единственный Web-адрес, а не несколько. Разработчики свободны по своему усмотрению реализовывать навигацию от этой единственной точки входа. Паттерн, поддерживающий этот стиль работы, именуется **паттерном Контроллер запросов** (Front Controller) [2]. Принципы Контроллера запросов проиллюстрированы на рис. 17.16.

Сервлет, который действует как Контроллер запросов, решает, какие представления отобразить, учитывая состояние сеанса пользователя. Именно здесь используются паттерны Стратегия или Фабричный метод. Паттерн Контроллер запросов позволяет обеспечить централизацию кода, связанного с сервисами системы, службами безопасности, поиском содержимого, управлением представлениями и навигацией. Паттерн способствует повторному использованию кода приложения при различных запросах (вместо помещения одного и того же кода в многочисленные представления).

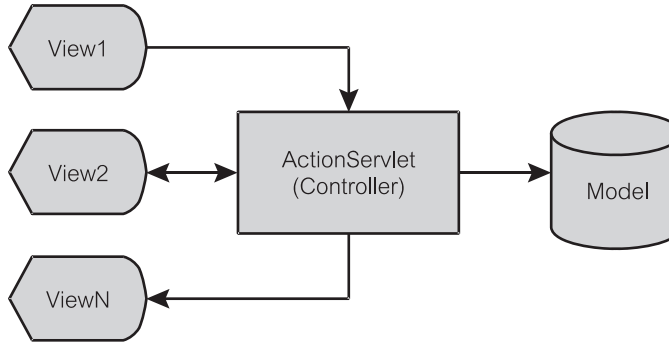


Рис. 17.16. Контроллер запросов

17.4.8. Повторное использование тегов в JSP

Учитывая возможности JSP-технологии, разработчики хотели бы повторно использовать компоненты из предварительно реализованных Web-страниц. JSP обеспечивает такое повторное использование через **Custom Tag Library** (библиотека пользовательских тегов). Разработчики страниц, типа JSP-разработчиков, сосредотачиваются на разработке Web-страниц с помощью пользовательских тегов. **Пользовательские теги** обеспечивают простое представление существенно более сложных функциональных возможностей низкого уровня, типа доступа к БД, доступа к сети или интернационализации. Разработчики ядра ПО реализуют эти функциональные возможности низкого уровня. Пользовательские теги, таким образом, позволяют «передавать знания» от разработчиков ядра ПО разработчикам страниц.

Ниже приведена последовательность операций использования библиотек пользовательских тегов (кратко называемых `taglib`):

1. Добавить директиву `taglib` к JSP-файлам.
2. Создать описатель библиотеки тегов (с расширением `.tld`) с описанием тегов и объявить его в файле `web.xml`.
3. Реализовать обработчики тегов, чтобы поддерживать теги.

Листинг 17.7 показывает, как библиотека тегов используется в JSP. Библиотека тегов, известная как `SampleTags` (пользовательские теги) упомянута в JSP как `mytag` — мой тег (строка 6). Эта ссылка находится в разделе заголовка HTML. По этой причине тег `mytag` доступен в теле HTML. Строка 15 показывает, как используется операция `openConnection` (открыть соединение) тега `mytag`, чтобы открыть соединение с БД, называемой `myDatabaseName` (имя моей БД). Как только БД будет открыта, формируется запрос для определения всех необходимых параметров служащего (строка 16). Результат запроса отображается в формате таблицы (строки 18–22).

Листинг 17.7. Директива taglib в JSP

Добавление директивы taglib к JSP-файлам (sample.jsp)

```
 2:  <head>
 6:     <%@ taglib uri='SampleTags' prefix='mytag' %>
 7:  </head>
11:  <body>
15:     <mytag:openConnection db="myDatabaseName">
16:        <mytag:query> select * from employee </mytag:query>
17:        <table>
18:        <mytag:rows>
19:           <tr><mytag:columns columnVal= 'cVal' >
20:              <td><%= cVal %> </td>
21:           </mytag:columns></tr>
22:        </mytag:rows>
23:     </mytag:openConnection>
```

Пользовательские теги, единожды разработанные, должны быть развернуты на Web-сервере, который поддерживает JSP типа Tomcat [105]. Объявления тегов должны быть помещены в описатели развертывания или они могут быть упомянуты в файле по имени `web.xml`. Листинг 17.8 показывает, как объявляется `taglib` из листинга 17.7.

Листинг 17.8. Объявление taglib в файле web.xml

Объявление директивы taglib в файле развертывания (web.xml)

```
10:  <taglib>
11:     <taglib-uri>SampleTags</taglib-uri>
12:     <taglib-location>/WEB-INF/tlds/SampleTags.tld</
                                              taglib-location>
13:  </taglib>
```

Листинг 17.8 показывает, как листинг 17.7, строка 6, может определить существование тега. Это возможно, потому что тег объявлен в листинге 17.8, строка 11. Реальное местоположение описателя тега задается в строке 12.

XML — обычно используемый в настоящее время стандарт для Web-разработки. Он включает разработку taglibs (библиотек тегов). Описатель taglib объявляется в XML, чтобы следовать формату, определенному в JSP-стандарте. Листинг 17.9 показывает избранные теги, объявленные в файле `SampleTags.tld` (примеры тегов). Строки 1–2 определяют стандарт, реализованный пользовательским тегом. Тип документа — `taglib` (строка 3), который соответствует XML-формату (строка 1). `SampleTags` имеет версию 1.0 (строка 4) и реализован согласно JSP-спецификации версии 1.2 (строка 5). Строки 6–9 обслуживают цели документации `taglib`.

Листинг 17.9. SampleTags.tld

SampleTags.tld

```
 1:  <?xml version="1.0" encoding="ISO-8859-1" ?>
 2:  <!DOCTYPE taglib PUBLIC
       "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
        "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
 3:  <taglib>
 4:     <tlibversion>1.0</tlibversion>
 5:     <jsp-version>1.2</jsp-version>
 6:     <short-name>My Sample Tags</short-name>
 7:     <uri>http://somewhere.com/tlds/SampleTags</uri>
 8:     <display-name>SampleTags</display-name>
 9:        <description>Sample Tags To Illustrate TagLib
          </description>
15:      <tag>
16:         <name>openConnection</name>
17:         <tag-class>sample.OpenConnectionTag</tag-class>
18:         <body-content>JSP</body-content>
19:         <description>
20:           Capable of nesting database connection from which
              queries can be executed
21:         </description>
22:         <attribute>
23:            <name>db</name>
24:            <required>true</required>
25:            <rtexprvalue>false</rtexprvalue>
26:         </attribute>
27:      </tag>
28:      <tag>
29:         <name>query</name>
30:         <tag-class>sample.QueryTag</tag-class>
31:         <body-content>JSP</body-content>
32:         <description> Execute query defined in its body
                                                  </description>
33:         <attribute>
34:         <name>var</name>
35:         <required>true</required>
36:         <rtexprvalue>false</rtexprvalue>
37:         </attribute>
38:      </tag>
```

Наиболее интересные части листинга 17.9 — строки 15–27 и 28–38, где объявлены два тега. Строки 15–27 определяют тег `openConnection` (открыть связь) — строка 16, который был реализован классом `openConnectionTag` (открыть тег связи) в пакете `sample` — пример

(строка 17). Этот тег имеет один атрибут, называемый db, который должен быть заполнен и который не возвращает никаких величин (строки 22–26). Точно так же тег query (запрос) должен быть объявлен в строке 29, чтобы ссылаться на реализацию класса sample.QueryTag.

На разработчика ПО возлагается реализация тегов, описанных выше. *Обработчики тегов* могут быть реализованы наследованием от javax.servlet.jsp.tagext.Tag. Из этого интерфейса должны быть реализованы только три метода: doStartTag() (начать тег), doEndTag() (завершить тег) и release() (отключить). Они показаны в листинге 17.10. Когда JSP-механизм встречает начало тега, он вызывает метод doStartTag(). Как только тег будет полностью обработан, выполняется его метод doEndTag(), чтобы позволить тегу выполнять другие необходимые операции. Наконец, когда тег больше не нужен и попадает в категорию «мусора», вызывается его метод release(), чтобы обеспечить для тега выход с *очисткой*.

Листинг 17.10. Сводка методов java.servlet.js.tagext.Tag

```
int    doEndTag()
           Обработка конца тега для данного экземпляра.
int    doStartTag()
           Обработка начала тега для данного экземпляра.
Tag    getParent()
           Получение предка (ближайший обработчик тега) для данного обработчика
           тега.
void   release()
           Обращение к обработчику тега, чтобы освободить состояние.
void   setPageContext(PageContext pc)
           Задание контекста текущей страницы.
void   setParent(Tag t)
           Задание предка (ближайший обработчик тега) для данного обработчика тега.
```

Документация фирмы Sun [101] иллюстрирует жизненный цикл тега в модели состояний, показанной на рис. 17.17. Задание его предка и значений контекста страницы инициализирует тег. После инициализации тег может перейти в состояние Process tag content (обработка содержимого тега) после обращения к doStartTag(). Как только тег будет закончен с помощью метода doEndTag(), он может быть освобожден с помощью вызова метода release(). Лишь затем тег может повторно использоваться, если все его параметры заново установлены в известные значения, используемые по умолчанию.

Обратите, однако, внимание, что цикл между состояниями Property initialized (свойства инициализированы) и Process tag content может быть выполнен, если обращение метода doStartTag() к тегу не вызовет никаких исключений. Ошибки могут быть сформированы как экземпляры JspException (исключение JSP) любым из методов doStartTag(), doEndTag() или release().

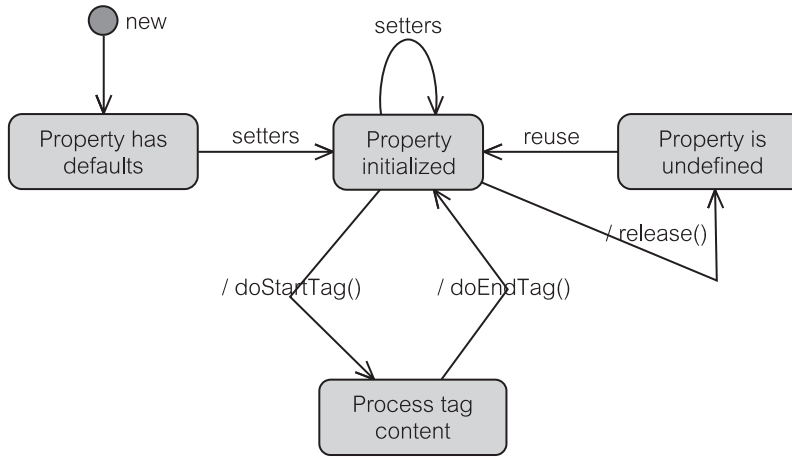


Рис. 17.17. Жизненный цикл тега

Быстрый набросок `OpenConnectionTag` (тег открытого соединения) показан в листинге 17.11. Класс расширяет `TagSupport` (поддержка тега), который содержит весь скелет для создания стандартного тега, включая пустую реализацию интерфейса `Tag` (тег). Атрибут `db`, объявленный в листинге 17.9, строка 23, пользуется возможностями метода `setDb()` — задать БД (строки 6 и 20–22).

Листинг 17.11. Реализация OpenConnectionTag

OpenConnectionTag.java

```
 1:  package sample;
 2:  import javax.servlet.jsp.*;
 3:  import javax.servlet.jsp.tagext.*;
 4:  import javax.servlet.http.*;
 5:  public class OpenConnectionTag extend TagSupport{
 6:     String dbName;
20:     public void setDb(String db){
21:        dbName = db;
22:     }
23:     public int doStartTag() throws JspException{
24:        performOpenConnection();
25:        return SKIP_BODY;   //не оценивать тело с этим тегом
26:     }
27:     public int doEndTag() throws JspException{
28:        if(conn != null) conn.close();
29:     }
```

Метод `doStartTag()` открывает новое соединение к названной БД через вызов `performOpenConnection()` — выполнить открытие соединения (строки 23–24). Предполагается, что метод `performOpenConnection()` является конфиденциально реализованным методом, и он здесь не показан. `return SKIP_BODY` (строка 25) указывает, что если имеются еще данные, вложенные в использование тега в JSP, их необходимо перескочить (не обрабатывать текущим тегом). Метод `doEndTag()` просто закрывает открытое соединение.

17.4.9. Несвязное управление: Struts

Технология **Struts** — поддержки [35] обеспечивает дополнительный уровень разъединения между представлением и бизнес-логикой. Struts — открытый проект фирмы Apache [3]. Эта технология была разработана из паттерна MVC-проекта и затем преобразована для включения в паттерн Контроллер запросов. Этот раздел посвящен MVC-аспектам Struts.

Большинство реализаций Struts следует диаграмме, показанной на рис. 17.18. Контроллер запросов представлен сервлетом действий `ActionServlet` (`Controller`) и действиями `ActionA-ActionZ`. `ActionServlet` единственный. Взаимодействие между `Controller` и его `Action` использует преимущества паттернов Стратегия, Фабричный метод, Сервис обеспечения работы, Диспетчер и Навигатор.

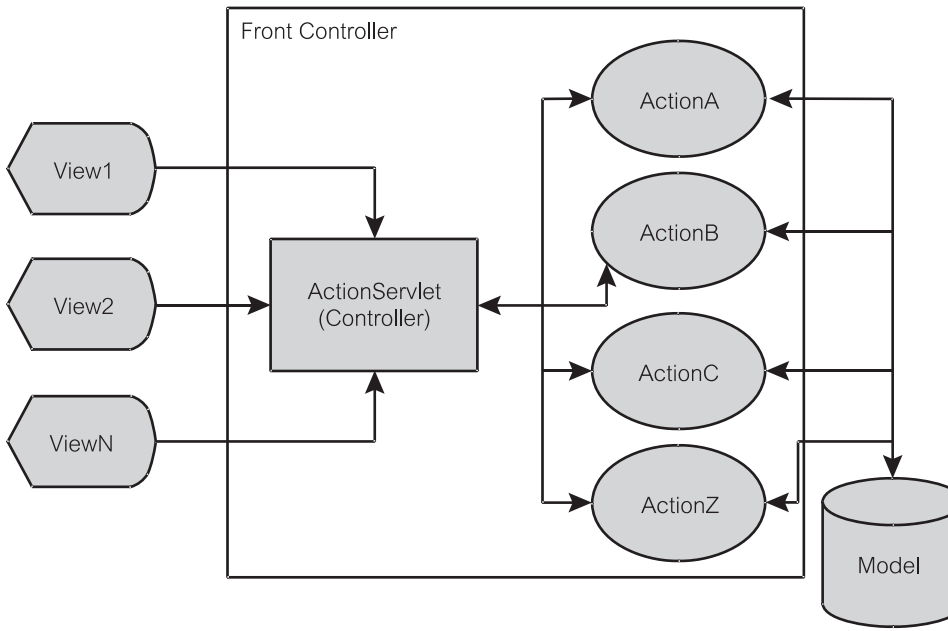


Рис. 17.18. Общее представление о Struts

Источник: материал, преобразованный из *Mastering Jakarta Struts* Джеймсом Гудвиллом. Copyright © 2002 by Ryan Publishing Group. Перепечатано здесь по разрешению Wiley Publishing, Inc. All rights reserved

ActionServlet решает, какой класс Action должен быть выполнен (Стратегия и Навигатор). Он делегирует выполнение сервиса (Сервис обеспечения работы) и распределяет события между классами Action (Диспетчер). Классы Action должны сообщить контроллеру ActionServlet с помощью паттернов Фабричный метод или Навигатор, какое представление отображать как результат выполнения.

Последовательность выполнения начинается с представлений View1-ViewN, требующих модификации данных, которая должна быть выполнена с помощью ActionServlet. ActionServlet динамически находит для выполнения подходящие действия (ActionA-ActionZ). Выбранный экземпляр действия выполняет операции на модели и возвращает результат к ActionServlet. ActionServlet представляет этот результат одним из View1-ViewN.

Как и в случае библиотеки пользовательских тегов в JSP, Struts требует, чтобы разработчик реализовал сервлет и обеспечил необходимое соответствие между реализованным классом и его опубликованным именем. Шаги здесь следующие:

1. Создать представления (обычно JSP-страницы — листинги 17.7, 17.9, 17.11).
2. Создать контроллер и классы действий.
3. Определить отношения между представлениями и контроллером (struts-config.xml).
4. Описать все компоненты Struts в файле web.xml (листинг 17.8).

Struts обеспечивает более высокий уровень абстракции, чем JSP. С уровнем presentation (представление) в Struts обращаются как с отдельной самостоятельной единицей. Struts позволяет Web-разработчикам отделить представление от действий, выполненных на основе бизнес-логики и данных.

17.5. Реализация сервлета, обеспечивающего управление электронной почтой

Рис. 17.19 показывает полную модель переходов состояний для реализации сервлета итерации 2 учебного примера EM. Сценарии использования регистрационного имени, отображения исходящего сообщения и передачи исходящего сообщения управляются сервлетом EMS (сервлет EM). Сервлет EMSEdit (редактор EMS) управляет просмотром и редактированием исходящих сообщений. Сервлеты фактически называются PEMS (класс «сервлет EM» пакета presentation) и PEMSEdit (класс «редактор сервлета EM» пакета presentation). Однако сервлеты могут иметь псевдонимы, которые отличаются от их первоначальных имен. Это позволяет Web-администраторам настраивать сервлеты по своим потребностям. Эта глава и глава 18 попеременно ссылаются на имена EMS и PEMS. Точно так же эквивалентны EMSEdit и PEMSEdit.

Версия сервлета итерации 2 EM является полностью основанным на Web-технологии учебным примером. После успешного задания регистрационного имени пользователю предлагается главная страница, где отображают-

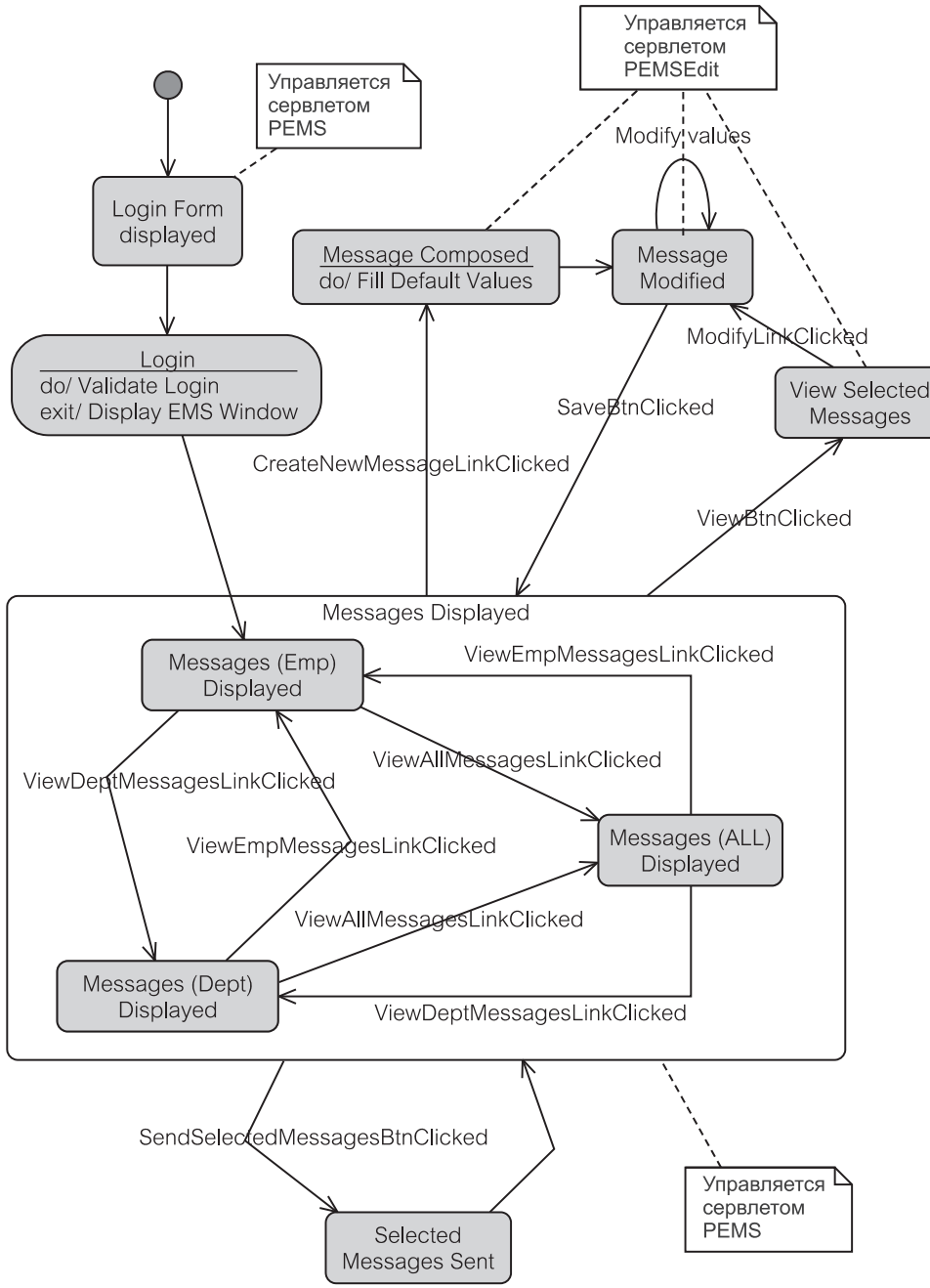


Рис. 17.19. Версия сервлета EM итерации 2 — модель состояний

Рис. 17.20. Экран задания регистрационного имени

ся непосланные исходящие сообщения, предназначенные этому пользователю. На этой странице пользователь может нажать мышью любую из следующих связей:

- связь со страницей «Исходящие сообщения отдела»;
- связь со страницей «Все непосланные исходящие сообщения»;
- кнопка просмотра/редактирования деталей исходящих сообщений, показанных на странице «Выбранные исходящие сообщения»;
- кнопка, чтобы послать исходящее сообщение со страницы «Выбранные исходящие сообщения».

Когда пользователь находится на любой из вышеупомянутых страниц, он/она может перейти к другим страницам через связи. Обычный экран ввода регистрационного имени для итерации 2 был преобразован в стандартный HTML-формат, как показано на рис. 17.20.

Сервлет загружается с локального Web-сервера. Окно задания регистрационного имени формируется сервлетом `EMS`, как указано его Web-адресом. Процесс проверки регистрационного имени показан на рис. 17.21. `CAdmin` (класс «администратор» пакета `control`) ответствен за выполнение последовательности проверки регистрационного имени согласно нормальной GUI-реализации итерации 2.

Метод `doGet()` (выполнить получение) требуется, когда сервлет вызывается первый раз, чтобы инициализировать процесс обработки регистрационного имени (рис. 17.20). После нажатия кнопки `Submit` (проверить) выполняется метод `doPost()` (выполнить передачу), и проверка регистрационного имени продолжается.

Как только регистрационное имя будет полностью проверено, показывается следующая страница, которая перечисляет все исходящие сообщения, назначенные текущему служащему (рис. 17.22). Пользователь может просмотреть и послать исходящие сообщения через это окно.

Обратите внимание, что рис. 17.22 показывает точно тот же самый Web-адрес, что и на странице задания регистрационного имени. Это указывает, что данные две страницы обслуживает один и тот же сервлет. В этом слу-

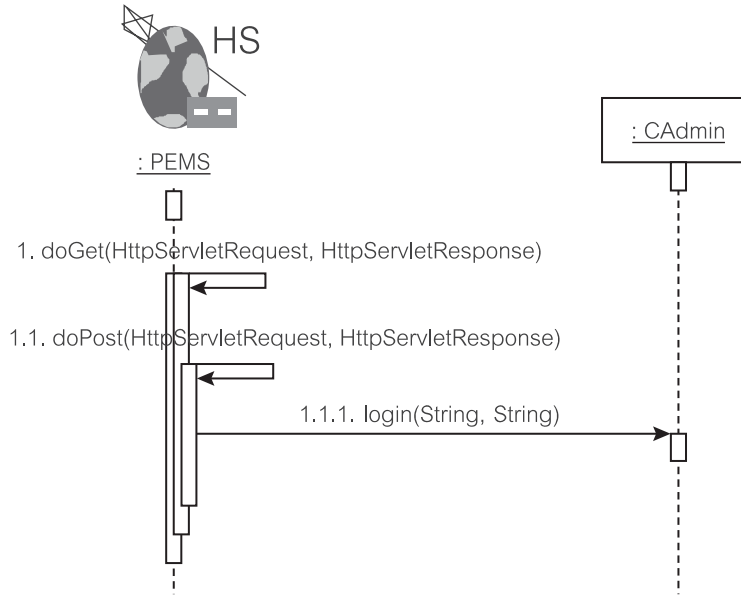


Рис. 17.21. Проверка регистрационного имени

чае именно сервлет `EMS` обслуживает данные страницы. Такой подход требует, чтобы сервлет `EMS` помнил состояния соединения пользователя на посессионной основе.

Диаграмма последовательности действий для внесения в список исходящих сообщений, назначенных текущему служащему, проиллюстрирована на рис. 17.23. Запрос на поиск исходящего сообщения перехватывается сервлетом `EMS`, который далее переадресовывает его к `CMsgSeeker` (класс «поиск сообщения» пакета `control`). Извлеченные исходящие сообщения затем

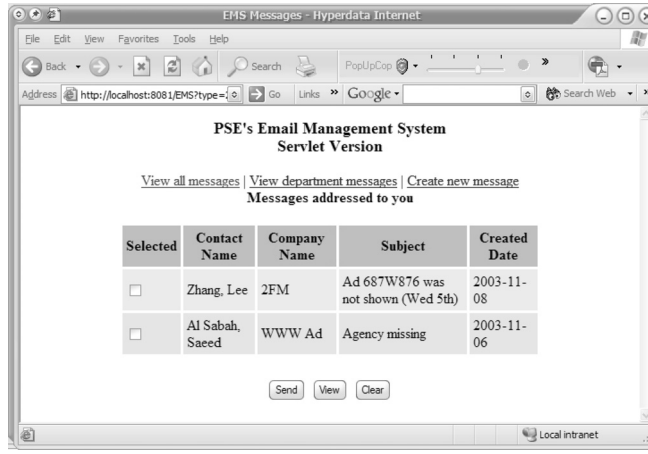


Рис. 17.22. Представление собственных исходящих сообщений

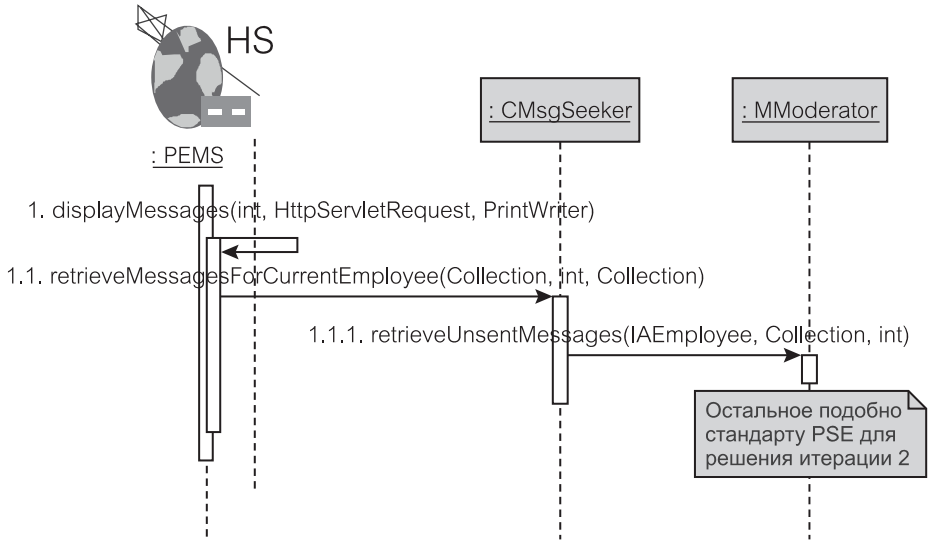


Рис. 17.23. Диаграмма последовательности отображения собственных исходящих сообщений

отображаются на экране, чтобы пользователь мог их просматривать и манипулировать с ними.

Когда пользователь решает просмотреть некоторые отобранные исходящие сообщения, содержимое этих исходящих сообщений отображается на экране, показанном на рис. 17.24.

Рис. 17.24 показывает, как отображается полученное в итоге содержимое отобранных исходящих сообщений. Обратите внимание, что нижняя часть

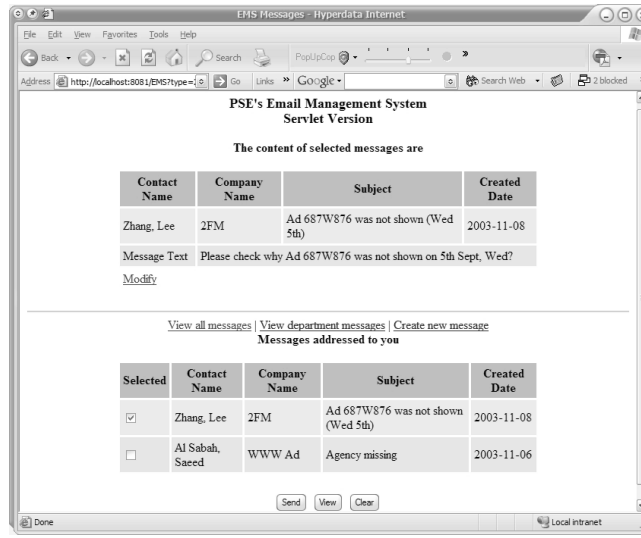


Рис. 17.24. Представление выбранных исходящих сообщений

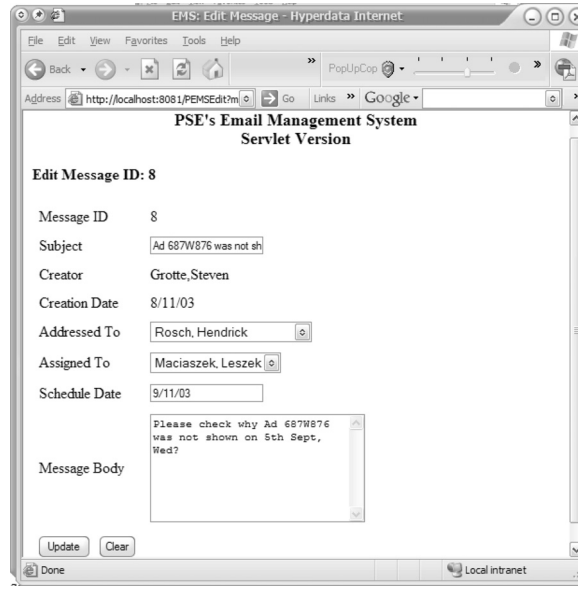


Рис. 17.25. Редактирование выбранных исходящих сообщений

окна содержит список исходящих сообщений, как и на рис. 17.22. Обратите также внимание, что таблица обеспечивает связь со страницей `Modify` (модифицировать) для просматриваемого исходящего сообщения. Редактирование сообщения снабжено сервлетом `EMSEdit`. Рис. 17.25 указывает в строке Address (адрес), что сервлет `EMSEdit` вызывается с параметром `msg=1`[2]. Этот параметр не показан на рис. 17.24, потому что он вложен в связь `Modify`.

Каждое из полей, показанных на рис. 17.25, можно редактировать. Выпадающие списки являются выбираемыми и содержат перечни всех доступных служащих и деловых партнеров. Последовательность выполнения редактирования сообщения использует два сервлета, показанные в диаграмме последовательности действий на рис. 17.26. Диаграммы последовательности действий подчеркивают взаимодействия между сервлетами `EMS` и `EMSEdit`, необходимые для редактирования исходящего сообщения.

Сервлет `EMS` обрабатывает запросы, чтобы просмотреть текст отобранного исходящего сообщения. Затем он вызывает `EMSEdit` с надлежащим параметром, чтобы показать исходящее сообщение на экране. Это завершается окном, показанным на рис. 17.25. Как только пользователь нажмет мышью кнопку `Update` (корректировать), изображенную на рис. 17.25, исходящее сообщение обновляется на сервере через запрос `updateMessage()` (корректировать сообщение) объекта `CAdmin` (класс «администратор» пакета `control`).

[2] К сожалению, на рисунке в этой строке, находящейся в левой верхней части страницы, изображен только первый симол — m, а остальные символы скрыты. (*Прим. перев.*)

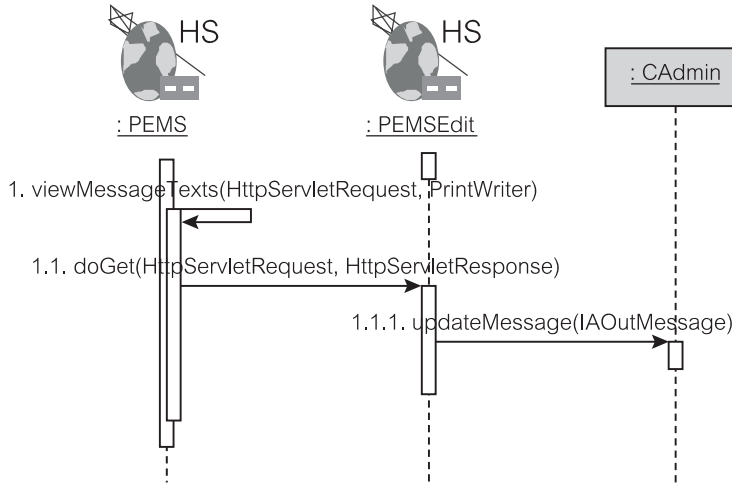


Рис. 17.26. Взаимодействия для редактирования исходящих сообщений

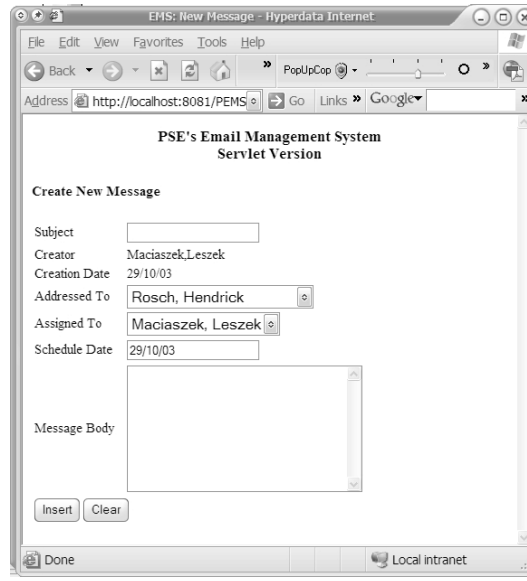


Рис. 17.27. Помещение нового исходящего сообщения

Рис. 17.27 представляет другое использование сервлета `EMSEdit`. Запрос показать страницу создания нового исходящего сообщения завершается окном, изображенным на рис. 17.27. Выпадающие списки и другие поля инициализированы значениями по умолчанию. Обратите внимание, что поля `Creator` (создатель) и `Creation Date` (дата создания) неизменяемые. Это предотвращает искажение данных и незаконную манипуляцию с исходящими сообщениями.

Несколько отличается от рис. 17.22 рис. 17.28, показывающий список всех непосланных исходящих сообщений, доступных в БД.

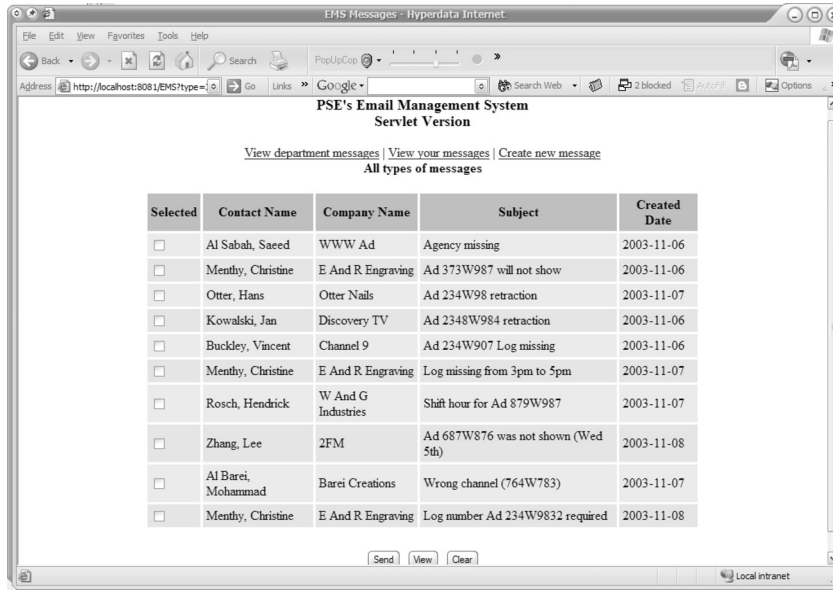


Рис. 17.28. Просмотр всех исходящих сообщений

Резюме

1. *Приложение, основанное на Web-технологии*, означает, что браузер Интернета управляет представлением содержимого пользовательского интерфейса, но бизнес-логика и состояние БД существуют на сервере. Технологии клиента для приложений, основанных на Web-технологии, включают *апплеты*. Технологии сервера включают сервлеты и *Java Server Pages* (страницы сервера Java). Другие связанные технологии — *теги*, *скриптлеты*, *Struts* и *JavaServer Faces*.
2. Компоненты типичной Web-системы развернуты, по крайней мере, на трех *уровнях*: Web-клиент, Web-сервер и сервер БД. В некоторых случаях желателен отдельный сервер приложения.
3. *Клиент/сервер* (client/server — C/S) — логическое разделение. *Клиент* запрашивает услуги сервера, а *сервер* обеспечивает эти услуги для авторизованных клиентов.
4. *HyperText Markup Language* (HTML) — язык разметки гипертекстов — наиболее общий формат, предназначенный для представления Web-страниц в сети. Страницы, размещенные в HTML-формате, используют стандартные *теги форматирования* и, как результат, обеспечивают их надлежащий показ Web-браузерами.
5. *Языки скриптов* используются в HTML-документах, чтобы обогатить возможности взаимодействия пользователя. *Скрипты* могут выполнять быструю проверку входных величин, осуществлять простую мультипликацию и другие операции, чтобы сделать Web-страницу «живой».

6. *Апплеты* можно разделить на тонкие и толстые. Как тонкие, так и толстые апплеты имеют своих напарников на сервере. *Тонкий апплет* исполняет большинство своих операций на сервере, а *толстый апплет* исполняет большинство своих операций на клиенте.
7. *Сервлет* — специализированный сервис, запускаемый на машине сервера, чтобы обслуживать многих клиентов. Он представляет собой код Java, который должен быть скомпилирован и размещен на Web-сервере. На клиенте нет никакого кода сервлета. Клиент — HTML-страница или другие страницы, типа JSP или ASP.
8. *Java Server Pages* (JSP) — серверные страницы на Java — разновидность технологии сервлета. JSP отделяет бизнес-логику от представления. Бизнес-логика представлена сервлетом, в то время как представление реализуется с помощью JSP. JSP способна использовать *JavaBeans* и *Enterprise JavaBeans* (EJB).
9. Системы Интернета являются *системами, не имеющими состояний*. Имеются два подхода к реализации в Интернете *системы, обладающей состояниями*, — используя *передачу состояния* и используя *объект-сеанс*.
10. *Паттерны ядра J2EE* обеспечивают лучшую практику программной инженерии для разработки с помощью J2EE-технологий, включая сервлеты, JSP и EJB, Web-сервисы, XML. Другие хорошо известные паттерны, типа *GoF* и наборов паттернов *PEAA*, также используются в Web-системах. Применяемые паттерны включают Наблюдателя, Компоновщика, Фабричный метод, Стратегию, Декоратора, MVC и Контроллер запросов.
11. JSP поддерживает повторное использование компонентов предварительно реализованных Web-страниц через *Custom Tag Library* (библиотека пользовательских тегов). Пользовательские теги обеспечивают простое представление более сложных функциональных возможностей низкого уровня, типа доступа к БД, доступа к сети или интернационализации.
12. *Struts* — открытый проект фирмы Apache — обеспечивает дополнительный уровень разъединения между представлением и бизнес-логикой.
13. Версия сервлета итерации 2 учебного примера EM — полностью основанная на Web-технологии реализация EM-учебного примера. Реализация с толстым апплетом итерации 2 учебного примера EM также доступна (и работоспособна) на Web-сайте книги.

Ключевые термины

Обзорные вопросы

1. Некоторые HTML-теги не требуют закрывающего тега, например `<p>`. Какова причина для этой «непоследовательности»? Что произойдет, если вы поместите закрывающий тег для такого HTML-тега?
2. Каково значение тега `<br/>`? Имейте в виду, что тег `<br>` заставляет браузер поместить разрыв строки.
3. Что произойдет, если тег не будет закрыт или когда соответствующие теги открытия/закрытия не в нужном порядке?
4. Что означает атрибут `width` (ширина), когда он определен в теге таблицы (как показано в листинге 17.1, строка 7)? Что произойдет, когда его величина будет слишком большой или слишком малой?
5. Строка 20 листинга 17.2 имеет следующий вход:

   ```
   < INPUT TYPE="text" NAME="percent" VALUE="1" >
   ```

   Какова цель и значение задания атрибута `value`?

6. Обсудите все «за» и «против» тонких и толстых апплетов. Приведите примеры приложений, подходящих для тонкого и для толстого апплетов.
7. Объясните связь между моделью «песочницы» и подписанными апплетами.
8. Каковы типичные обязанности уровня Web-сервера в многоуровневой Web-системе?
9. Каковы типичные обязанности уровня сервера приложения в многоуровневой Web-системе?
10. Объясните различные пути объединенного использования сервлета и JSP-технологий для создания Web-систем.
11. Объясните различные пути получения реализации, обладающей состояниями, в Интернете, не имеющем состояний.
12. Объясните соответствие между уровнями ядра J2EE и PCMEF-шаблоном.
13. Объясните, как паттерны Наблюдатель и MVC могут быть использованы для проектирования различных вариантов Web-систем. Варианты, которые следует рассмотреть, включают тонкий апплет, толстый апплет и сервлет/JSP. Какие другие паттерны могут использоваться, чтобы компенсировать любые недостатки паттернов Наблюдатель и MVC?
14. Оцените критически рисунки 17.25 и 17.27 с точки зрения проектирования пользовательского интерфейса.

Примеры задач

1. Обратимся к рис. 17.20. Реализация окна регистрационного имени использует технологию сервлета. Реализуйте то же самое окно регистрационного имени, используя HTML. Предположите, что EMS-сервлет может обращаться с post-запросом (запросом на передачу) HTTP этого HTML.
2. Предположим, что вы должны разработать Web-страницу, как на рис. 17.24. Как бы вы могли отобразить содержимое отобранного исходящего сообщения, которое следует за списком исходящих сообщений, адресованных текущему пользователю? Какие JSP-теги могут использоваться, чтобы получить это? Используйте в решении директиву включения `<jsp:include>`. Не нужно демонстрировать JSP-файлы, которые поставляют Web-страницу.
3. Рис. 17.24 показывает таблицу, которая не имеет одинаковое число столбцов для строк. Как бы вы запрограммировали эти функциональные возможности в HTML-теге таблицы?
4. Перечислите изменения, необходимые для преобразования версии сервлета итерации 2 учебного примера EM к приложению на основе Struts.

Глава

18 Итерация 2. Аннотированный код

Учебный пример EM, рассматриваемый в данной главе, не охватывает код, который был представлен в части 2 в контексте итерации 1. Эта глава продолжает стиль представления, принятый в главе 13 *«Итерация 1. Аннотированный код»*. Однако здесь используется дополнительная технология, чтобы представить API EM. Документация, произведенная с помощью **Javadoc**, дополнена снимками диаграмм классов, созданными инструментальным средством ПО, называемым **yDoc**, фирмы yWorks — компании, специализирующейся на диаграммах [112].

Глава концентрируется на объяснении изменений кода между итерацией 1 и итерацией 2. Любой код, не представленный в этой главе по причинам недостатка места, доступен на Web-сайте книги. Глава начинается с обзора кода для итерации 2. Обзор кода объясняет иерархии пакетов, интерфейсов и классов. Соответствующая диаграмма классов дает визуальный обзор кода. Перечислено и аннотировано большинство полей и методов. Наиболее важные методы подробно обсуждены, а менее важные или тривиальные опущены.

Web-версия итерации 2 учебного примера EM обсуждается после представления настольной версии итерации 2. Web-версия представляет итерацию 2 в форме апплета, а также в виде реализации сервлетов. Понятно, что большая часть обсуждения Web-реализации итерации 2 касается пакета `presentation` (представление).

18.1. Обзор кода

Происходя из итерации 1, код итерации 2 учебного примера EM строго следует **структурному шаблону PCMEF+**, рассмотренному в разделе 9.2.2. Иерархия пакетов и классы внутри каждого пакета показаны на рис. 18.1.

Имеются значительные различия между моделью классов итерации 2 (рис. 18.2) и моделью итерации 1, представленной в главе 13. Многие из этих изменений уже были обсуждены в главе, посвященной рефакторингу (глава 15). Некоторые важные изменения следующие:

- пакет `presentation` (представление) перестроен так, чтобы поддержать графические и основанные на Web-технологии пользовательские интерфейсы;

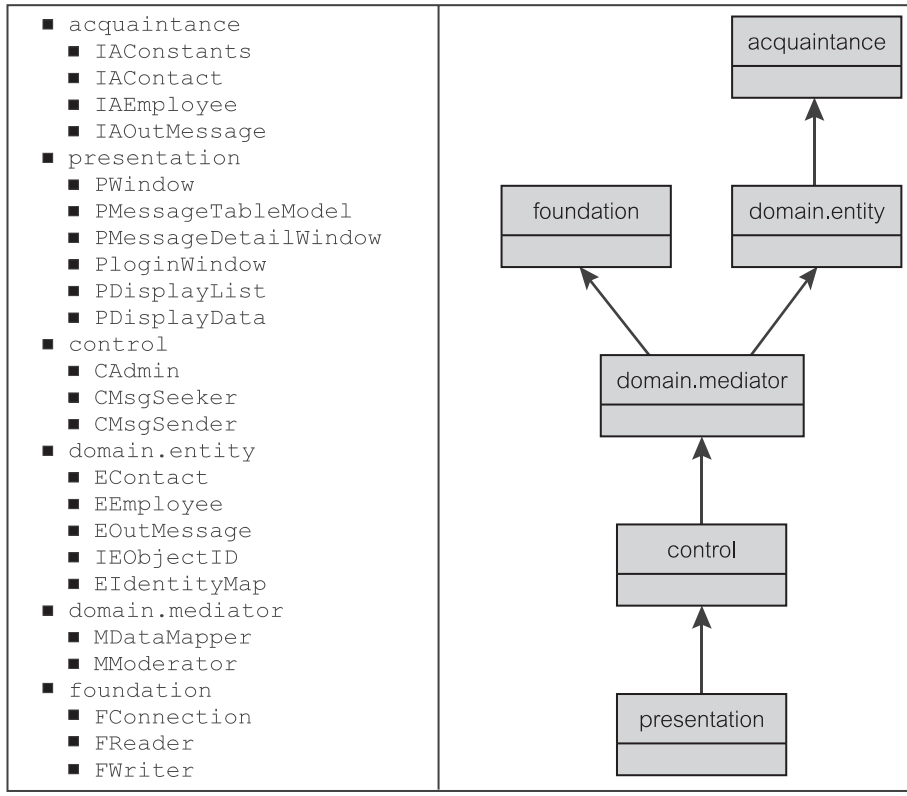


Рис. 18.1. Пакеты итерации 2 учебного примера EM

- `CActioner` — класс «исполнитель» пакета `control` был разбит на `CAdmin` — класс «администратор» пакета `control`, `CMsgSeeker` — класс «поиск сообщения» пакета `control` и `CMsgSender` — класс «передача сообщения» пакета `control` (и соответствующие изменения были сделаны в пакете `mediator` — посредник);
- интерфейс `domain.entity.IEDataSupplier` (интерфейс «источник данных» пакета `entity` уровня `domain`) больше не используется;
- добавлены классы `EIdentityMap` (класс «коллекция идентичности объектов» пакета `entity`) и `MDataMapper` (класс «преобразователь данных» пакета `mediator`), чтобы управлять кэшированием объектов пакета `entity`.

Обсуждение в этой главе ограничено из-за недостатка места. Читатели, заинтересованные в более детальном изучении, приглашаются посетить ресурсы на Web-сайте книги, которые включают исходный код для всех итераций учебного примера EM.

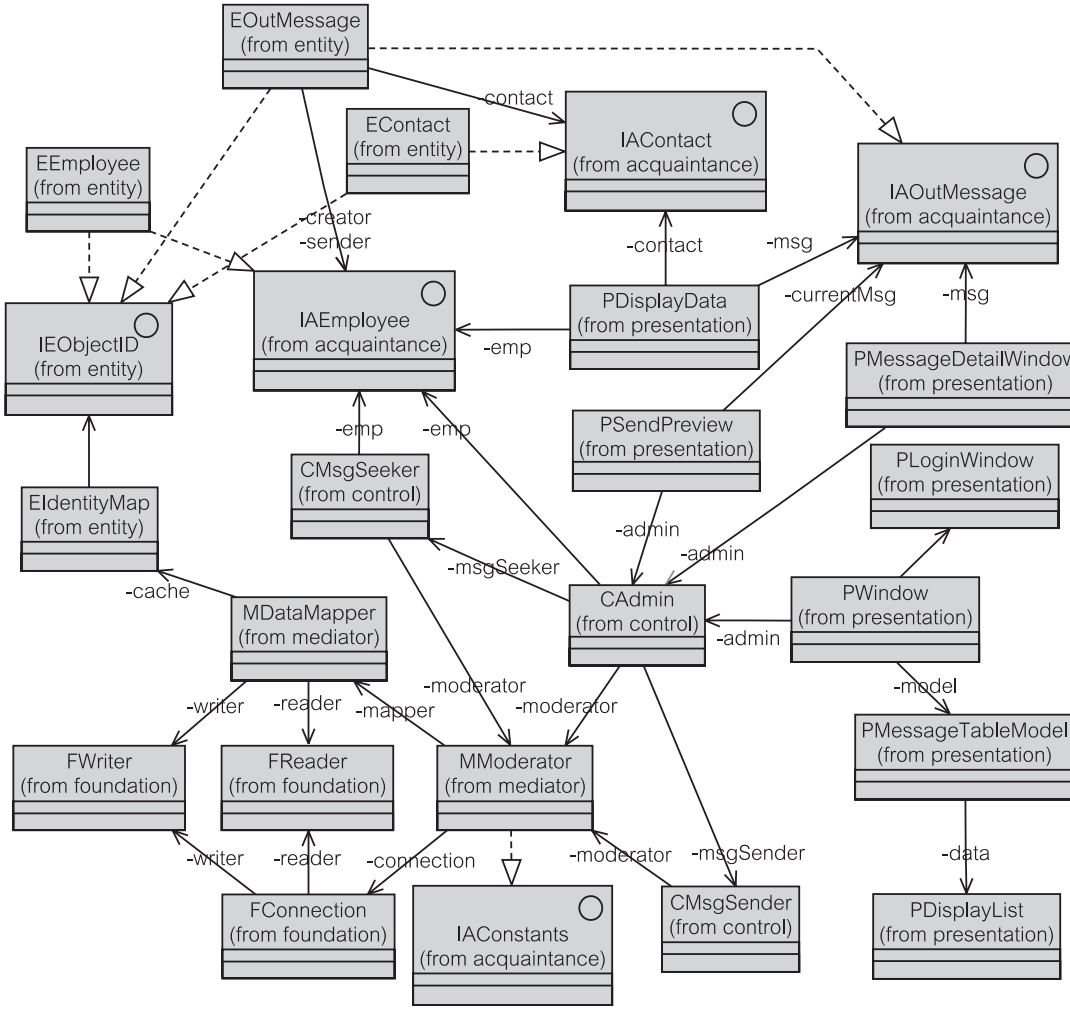


Рис. 18.2. Диаграмма классов итерации 2 учебного примера EM

18.2. Пакет Acquaintance

Пакет `acquaintance` (знакомство) состоит из четырех интерфейсов (рис. 18.3). `IAConstants` (интерфейс «константы» пакета `acquaintance`) в итерации 2 не изменен и поэтому в данной главе не рассматривается.

Хотя пакет `acquaintance` состоит из того же числа интерфейсов, что и в предыдущей итерации, каждый интерфейс представляет новые свойства, чтобы обеспечить сопровождение объектов менеджером памяти. Интерфейсы позволяют *Коллекции идентичности объектов* (раздел 15.3.1) размещать объекты, представляемые этими интерфейсами.

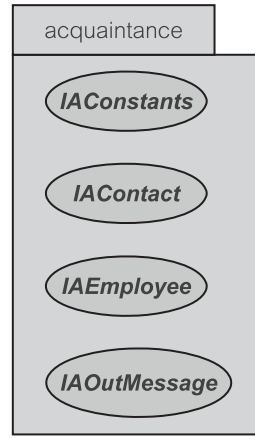


Рис. 18.3. Пакет Acquaintance и его классы в EM 2

18.2.1. Интерфейс IAEmployee

Заметной модификацией `IAEmployee` (интерфейс «служащий» пакета `acquaintance`) является добавление методов `isDirty` (измененный), `get/setDepartmentCode` (получить/задать код отдела) и `add/removeOutMessage` (добавить/удалить исходящее сообщение) — рис. 18.4. Они позволяют обеспечить надлежащее сопровождение связей между служащим и его/ее исходящими сообщениями.

Итераторы `java.util.iterator` используются, чтобы просматривать исходящие сообщения. Это — удобный способ позволить приложению искать исходящие сообщения служащего. Различают два различных типа исходящих сообщений: исходящие сообщения, назначенные служащему как отправителю и исходящие сообщения, созданные служащим. `EEmployee` (класс «служащий» пакета `entity`) — единственный класс, который реализует интерфейс `IAEmployee`. Два других интерфейса, связанные с объектами пакета `entity`, а именно, `IAOutMessage` (интерфейс «исходящее сообщение» пакета `acquaintance`) и `IAContact` (интерфейс «деловой партнер» пакета `acquaintance`) следуют той же самой тенденции, что и `IAEmployee`.

18.3. Пакет Presentation

Пакет `presentation` (представление) содержит семь классов и один внутренний класс (рис. 18.5). Согласно объектно-ориентированному принципу «равномерного распределения сведений», поведение, находящееся в итерации 1 пакета `presentation`, было разделено среди ряда новых классов, введенных в итерации 2.

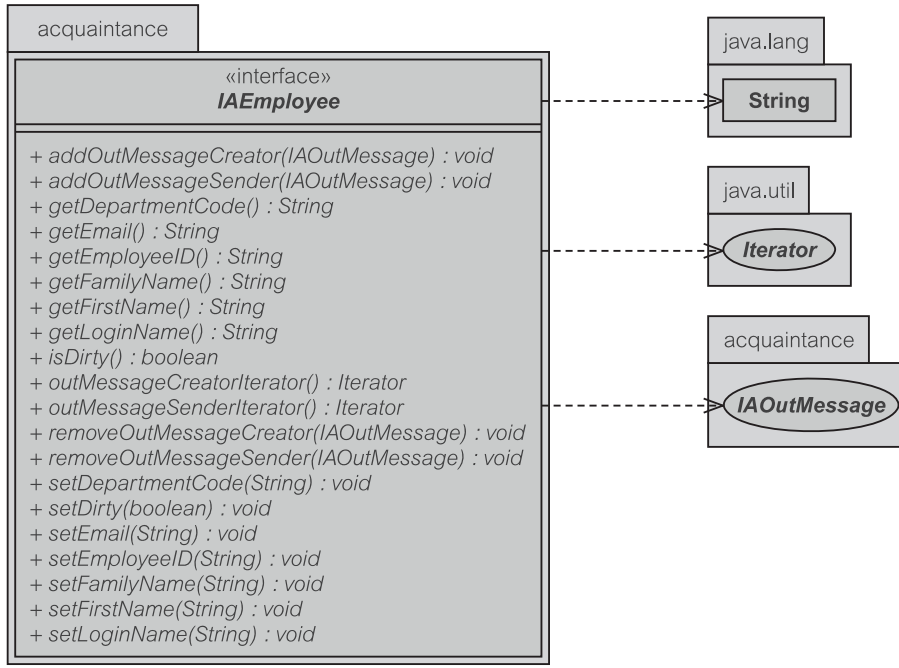


Рис. 18.4. Интерфейс IAEmployee в EM 2

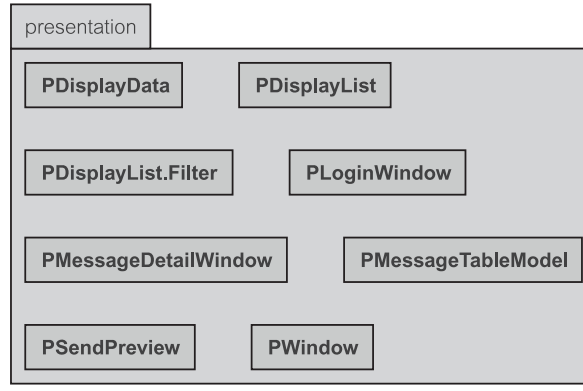


Рис. 18.5. Пакет Presentation и его классы в EM 2

18.3.1. Класс PWindow

Класс `PWindow` — класс «окно» пакета `presentation` (рис. 18.6) является заменой для `PConsole` — класс «консоль» пакета `presentation`. Класс также содержит метод `main()` для начала работы приложения, который ранее был размещен в `PMain` (класс «основной» пакета `presentation`).

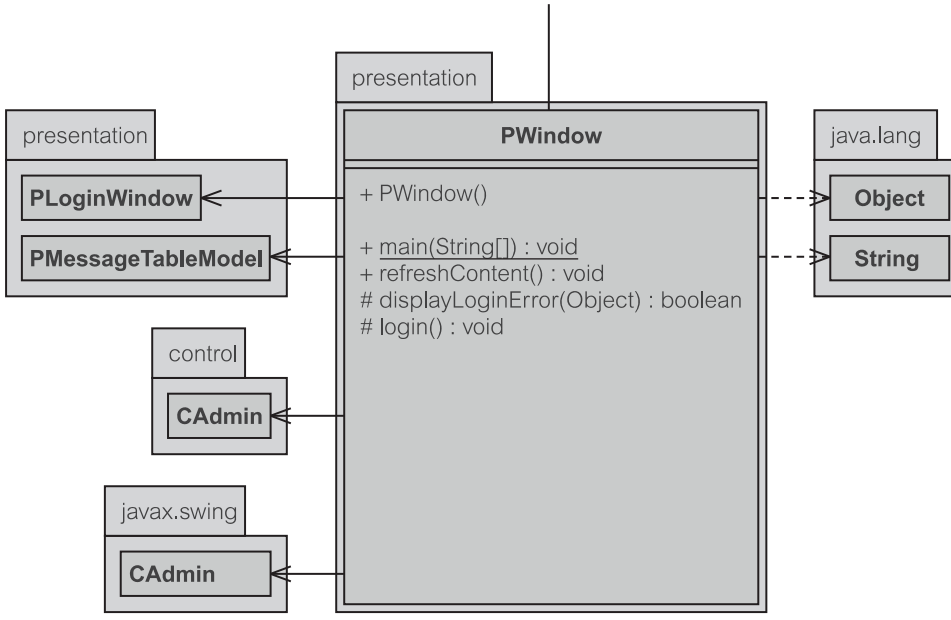


Рис. 18.6. PWindow в EM 2

Класс `PWindow` содержит ссылки на `CAdmin` (класс «администратор» пакета `control`), который является **доминирующим классом** уровня `control` (управление), и на `PMessageTableModel` (класс «модель таблицы для сообщений» пакета `presentation`), который является единственной ссылкой на модель `JTable` (класс «таблица» пакета Swing), используемую в GUI. Упрощенная конструкция `JTable` показана на рис. 16.3. `PLoginWindow` (класс «окно для регистрационного имени» пакета `presentation`) используется, чтобы получить информацию о регистрационном имени для соединения с БД. Библиотека `javax.swing` используется широко, но детали на рис. 18.6 исключены.

Конструирование и запуск PWindow

Java инициализирует различные события в процессе конструирования элементов управления. В режиме проектирования GUI программист может пожелать игнорировать эти события Java и сконцентрироваться на событиях, инициализируемых пользователем. Для этой цели используется `designMode` (режим проектирования), чтобы указать, что конструируется GUI. Когда конструирование будет закончено, параметру `designMode` будет задано значение `false` — ложь (строка 35 в листинге 18.1).

Листинг 18.1. Конструирование PWindow

Конструкторы PWindow()

```
23:     public PWindow() throws Exception{
24:        designMode = true;
25:        admin = new CAdmin();
28:        login();
29:        model = new PMessageTableModel(retrieveData(1));
30:        initComponents();
31:        model.attachSortHeading(viewTable);
                                  //активизация сортировки
32:        populateContact();
33:        populateDepartment();
34:        setButtonsEnable(false);
35:        designMode = false;
36:     }
```

Итерация 2 следует тому же требованию итерации 1, что пользователь должен использовать `login()` — регистрационное имя (строка 28) для работы с БД прежде, чем он/она сможет использовать систему. Как только пользователь будет проверен, данные, находящиеся в `PMessageTableModel`, извлекаются с помощью вызова функции `retrieveData()` — извлечь данные (строка 29). Модель также поддерживает сортировку своих столбцов. Это достигается с помощью вызова метода `attachSortHeading()` — подключить сортировку заголовков (строка 31). В GUI должны быть размещены различные другие элементы типа комбинированной строки ввода делового партнера или комбинированной строки ввода отдела (строки 32–33). Эти комбинированные строки ввода позволяют пользователю выбирать значение из выпадающего списка. Они используются, например, для фильтрации исходящих сообщений (будет объяснено позже).

Приложение начинается с метода `main()` класса `PWindow` (листинг 18.2). Строка 569 содержит запрос к `UIManager` (менеджер пользовательского интерфейса), чтобы представить Java-приложение с видом и поведением, характерными для текущей ОС. По умолчанию Java будет использовать вид и поведение, известные как *Metal*. Затем создается и выводится на экран новый экземпляр `PWindow` (строки 572–573).

Листинг 18.2. Запуск PWindow — main()

Первоначальный запуск PWindow — main()

```
567:    public static void main(String args[]){
568:       try{
569:          UIManager.setLookAndFeel(
                 UIManager.getSystemLookAndFeelClassName());
570:       }catch(Exception exc){}
```

```
571:        try{
572:           PWindow w = new PWindow();
573:           w.show();
574:        }catch(Exception exc){
575:           exc.printStackTrace();
576:        }
577:     }
```

Извлечение данных в PWindow

Имеются три различных типа извлечения данных, поддерживаемых классом `PWindow` (листинг 18.3). Первый — извлечение всех непосланных исходящих сообщений (подпоток S1.1, описанный в разделе 14.2.3). Второй — извлечение всех непосланных исходящих сообщений, намеченных только отделу конкретного служащего (подпоток S1.2). Последний тип — извлечение всех непосланных исходящих сообщений, намеченных текущему служащему (подпоток S1.3).

Листинг 18.3. Метод retrieveData() в PWindow

Метод retrieveData() в PWindow

```
72:     private Collection retrieveData(int type){
73:        Collection data = new ArrayList();
75:        int remainder = 0;
76:        switch(type){
77:           case 1: remainder = admin.getMsgSeeker()
                  .retrieveAllMessages(data,
                  IAConstants.MAX_MESSAGE,null);
                  break;
78:           case 2: remainder = admin.getMsgSeeker()
                  .retrieveMessagesForDepartment(
                  admin.getEmployee().getDepartmentCode(),
                  data,IAConstants.MAX_MESSAGE,null);
                  break;
79:           case 3: remainder = admin.getMsgSeeker()
                  .retrieveMessagesForCurrentEmployee(
                  data,IAConstants.MAX_MESSAGE,null);
                  break;
80:           default: throw new IllegalArgumentException(
                  "Unknown type:"+type);
81:        }
85:        return data;
86:     }
```

Всякий раз, когда пользователь нажимает мышью любую из трех командных кнопок, названных «View department messages» (просмотр сообщений от-

дела), «View own messages» (просмотр своих сообщений) и «View all messages» (просмотр всех сообщений), необходимо обновить исходящие сообщения, отображаемые на экране. Метод, ответственный за изменение списка исходящих сообщений на экране, — `changeModel()` — изменить модель (листинг 18.4). Метод извлекает данные через вызов `retrieveData()` — извлечь данные (строка 424) и затем далее фильтрует данные согласно предварительно выбранным опциям фильтра. Первоначально исходящие сообщения фильтруются по датам, как показано в строках 426–428. Исходящие сообщения могут быть далее отфильтрованы по информации об их деловых партнерах, как показано в строках 430–431. Наконец, если пользователь затребовал фильтрацию по отделу, исходящие сообщения будут отфильтрованы так, как показано в строках 433–434. Поскольку исходящие сообщения отличаются от предыдущих, необходимо повторно заполнить комбинированные строки ввода для делового партнера и отдела (строки 436–437).

Листинг 18.4. Метод changeModel() в PWindow

Метод changeModel() в PWindow

```
423:  private void changeModel(int type){
424:     model.setModel(retrieveData(type));
425:     if (filterOnOffBtn.isSelected()){
426:        model.filter(todayChk.isSelected()?
              PMessageTableModel.TODAY_FILTER:
427:           (futureChk.isSelected()?
                 PMessageTableModel.FUTURE_DAYS_FILTER:
428:              (pastChk.isSelected()?
                 PMessageTableModel.LAST_DAYS_FILTER:
429:                 PMessageTableModel.ALL_DAYS_FILTER)));
430:        String cont = contactCmb.getSelectedItem().
                                          toString().trim();
431:        model.filter(cont.length() == 0?
              PMessageTableModel.NO_CONTACT_FILTER:cont);
433:        String dep=departmentCmb.getSelectedItem().
                                          toString().trim();
434:      model.filterDepartment(dep.length() == 0? null: dep);
435:     }
436:     populateContact();
437:     populateDepartment();
438:  }
```

Листинг 18.5 показывает реализацию подпотоков S1.1, S1.2 и S1.3. Все эти методы используют метод `changeModel()`, передавая ему различные параметры. Снова, если GUI находится в `designMode` (режим проектирования), событие кнопки игнорируется.

Листинг 18.5. Методы ownBtnActionPerformed(), deptBtnActionPerformed() и allBtnActionPerformed() в PWindow

Методы ownBtnActionPerformed(), deptBtnActionPerformed() и allBtnActionPerformed() в PWindow

```
416:    private void ownBtnActionPerformed(ActionEvent evt) {
418:          if(designMode) return;
419:          changeModel(3);
420:    }
439:    private void deptBtnActionPerformed(ActionEvent evt) {
441:          if(designMode) return;
442:          changeModel(2);
443:    }
445:    private void allBtnActionPerformed(ActionEvent evt) {
447:          if(designMode) return;
448:          changeModel(1);
449:    }
```

Задание в комбинированных строках ввода делового партнера и отдела — сходные операции. Листинг 18.6 показывает, как это сделано для комбинированной строки ввода отдела. Названия отделов должны читаться из модели (строка 98) и помещаться в комбинированную строку ввода (строка 102). Однако с отделом обращаются несколько по-другому в том смысле, что нужно проверить, есть ли хотя бы одно исходящее сообщение, назначенное отделу. Если исходящее сообщение не было назначено на конкретный отдел, то нет никакой необходимости добавлять такой отдел к комбинированной строке ввода, как показано в строке 101.

Листинг 18.6. Метод populateDepartment() в PWindow

Метод populateDepartment() в PWindow

```
95:     private void populateDepartment(){
96:        departmentCmb.removeAllItems();
97:        departmentCmb.addItem("");
98:        Iterator it=model.getDepartmentNames().iterator();
99:        while(it.hasNext()){
100:         Object d = it.next();
101:         if(d != null && d.toString().trim().length() != 0)
102:             departmentCmb.addItem(d);
103:          }
104:    }
```

Когда пользователь решает предварительно просмотреть содержимое выделенного исходящего сообщения, отображается новое окно `PSendPreview` (класс «предварительный просмотр отправителем» пакета `presentation`)

с содержимым исходящего сообщения (листинг 18.7). В этом окне пользователь может увидеть исходящее сообщение, как оно будет отображено у получателя электронного письма. Как можно заметить, многие из окон на уровне `presentation` (представление) реализуют соответствующим образом функцию `setTarget()` (задать цель).

Листинг 18.7. Метод previewBtnActionPerformed() в PWindow

Метод previewBtnActionPerformed() в PWindow

```
389:  private void previewBtnActionPerformed(ActionEvent evt) {
390:     int []rows=viewTable.getSelectedRows();
391:     if(rows.length == 0) return;
392:     for(int i=0;i<rows.length;i++) {
393:        PSendPreview msgWin = new PSendPreview(this,true,
                                                        admin);
394:        msgWin.setTarget((IAOutMessage)model.
           getRawObject(rows[i]));
395:        msgWin.show();
396:     }
397:     refreshContent();
398:  }
```

Активизация фильтра

Возможность фильтрации может быть включена или выключена в зависимости от состояния кнопки-переключателя (`filterOnOffBtn` — кнопка включения/выключения фильтрации). Однако реализация кнопки-переключателя в Java не может снять включение. Это означает, что как только кнопка-переключатель выбрана, нет никакого способа отменить это. Чтобы получить отключенное состояние кнопки-переключателя, код в листинге 18.8 снабжается другой кнопкой-переключателем (`invisibleBtn` — невидимая кнопка). И `invisibleBtn`, и `filterOnOffBtn` сгруппированы в `ButtonGroup` (группа кнопок). `ButtonGroup` позволяет обеспечить только единственный выбор какого-либо из компонентов в группе (поэтому другие кнопки отключены). Таким образом, когда требуется отключенное состояние, должна быть выбрана `invisibleBtn`, как показано в строках 462–464. Метод `setButtonsEnable` (задать разрешение кнопок) используется, чтобы задать все другие области фильтрования, а разрешающие и запрещающие кнопки зависят от состояния `filterOnOffBtn`.

Всякий раз, когда пользователю нужно фильтровать исходящие сообщения на основе информации об его/ее деловом партнере или отделе, он/она может сделать это, нажимая мышью комбинированную строку ввода отдела или делового партнера. Фильтрация выполняется в методе `filterDepartment()` (фильтрация по отделу) модели (строка 404 в листинге 18.9) или в соответствующем методе для фильтрации по деловому партнеру. Прежде чем фильтрацию можно будет выполнить, необходимо проверить, что GUI не находится в

режиме designMode и что кнопке filterOnOffBtn позволяется выполнять фильтрацию. Фильтрация не будет выполняться, если GUI находится в режиме проектирования или кнопка фильтра выключена.

Листинг 18.8. Метод filterOnOffBtnActionPerformed() в PWindow

Метод filterOnOffBtnActionPerformed() в PWindow

```
450:    boolean prevSelected = false;
451:    private void filterOnOffBtnActionPerformed
                                        (ActionEvent evt) {
453:       if(designMode){
454:          setButtonsEnable(false);
455:          return;
456:       }
458:       if(!prevSelected){
459:          prevSelected = true;
460:          setButtonsEnable(true);
461:       }else{
462:          setButtonsEnable(false);
463:          invisibleBtn.setSelected(true);
464:          prevSelected = false;
465:       }
466:    }
```

Листинг 18.9. Метод departmentCmbActionPerformed() в PWindow

Метод departmentCmbActionPerformed() в PWindow

```
400:   private void departmentCmbActionPerformed(ActionEvent
                                                        evt) {
402:       if(designMode || !filterOnOffBtn.isSelected())
                                                        return;
404:       model.filterDepartment(departmentCmb.
                  getSelectedItem().toString());
405:    }
```

Листинг 18.10 показывает один пример фильтрации по дате. Метод фильтрации объекта PMessageTableModel вызывается с соответствующим параметром, как показано в строке 543. Подобный подход используется для futureChkActionPerformed() (проверка действий, выполняемых в будущем), pastChkActionPerformed() (проверка действий, выполненных в прошлом) и allChkActionPerformed() (проверка всех выполняемых действий).

Листинг 18.10. DateFiltering в PWindow

DateFiltering в PWindow: метод todayChkActionPerformed()

```
539:    private void todayChkActionPerformed(ActionEvent evt) {
541:       if(designMode) return;
543:       model.filter(PMessageTableModel.TODAY_FILTER);
544:    }
```

18.3.2. Класс PMessageDetailWindow

Объект `PMessageDetailWindow` (класс «окно деталей сообщения» пакета `presentation`) используется, чтобы показать детали исходящего сообщения (листинг 18.11). Он может также отобразить данные пустого исходящего сообщения, как показано в методе `showNewMessage()` (показать новое сообщение). Показанное исходящее сообщение может быть изменено с помощью вызова метода `setEditable()` (задать режим редактирования).

В случае если пользователь захочет изменить исходящее сообщение, потребуется коллекция служащих и деловых партнеров (листинг 18.12). Если исходящее сообщение редактируемо, то пользователь имеет возможность перепоручить другому служащему работу с ним и/или изменить делового партнера, которому предназначено это исходящее сообщение. Эти две коллекции используются также и при создании нового исходящего сообщения.

Метод `SetTarget()` (задать цель) заставляет `PMessageDetailWindow` показать информацию `IAOutMessage` (интерфейс «исходящее сообщение» пакета `acquaintance`), которую передает этот метод (листинг 18.13). Название окна зависит от того, является ли оно редактируемым. Когда оно редактируемо и исходящее сообщение содержит ссылку на делового партнера, название изменяется на *Update Message* (скорректировать сообщение); если этой ссылки нет, окно называется *Insert New Message* (добавить новое сообщение). Когда окно не редактируемо, название меняется на *Delete Message* (строки 85–87). Различные другие свойства типа поля делового партнера, поля служащего и т. д. заполняются информацией, извлеченной из исходящего сообщения (строки 89–94).

Листинг 18.11. Сводка методов класса PMessageDetailWindow

```
static void  main(java.lang.String[]  args)
                Тестирование.
void         setEditable(boolean editable)
                Сделать поля редактируемыми/нередактируемыми.
void         setTarget(IAOutMessage msg)
                Изменить содержимое экрана.
void         showNewMessage()
                Очистить все поля и показать пустое сообщение.
```

Листинг 18.12. Поля в PMessageDetailWindow

Поля в PMessageDetailWindow.java

```
19:     private control.CAdmin admin;
20:     private boolean editable;
21:     private acquaintance.IAOutMessage msg;
23:     private java.util.Collection employees;
25:     private java.util.Collection contacts;
```

Листинг 18.13. Метод setTarget() в PMessageDetailWindow

Метод setTarget() в PMessageDetailWindow

```
82:   public void setTarget(acquaintance.IAOutMessage msg){
83:    this.msg = msg;
84:    if(editable){
85:     if(msg.getContact() != null)
                setTitle("Update Message "+msg.getMessageID());
86:     else setTitle("Insert New Message"+msg.getMessageID());
87:   }else setTitle("Delete Message"+msg.getMessageID());
89:        assignContact(msg.getContact());
90:        assignTo(msg.getSenderEmployee());
91:        msgIDLbl.setText(""+msg.getMessageID());
92:        creatorLbl.setText(msg.getCreatorEmployeeID());
93:        deptCodeLbl.setText(msg.getDepartmentCode());
94:        fillMessageDetails(msg);
95:     }
```

Метод `saveMessage()` (сохранить сообщение) сохраняет информацию, содержащуюся в полях `PMessageDetailWindow` после модификации пользователем (листинг 18.14). Первоначально поля проверяются перед их вводом (строки 339–344 и вырезанные строки). Как только значения всех полей будут проверены, соответственно изменяется исходящее сообщение (строки 397–399). Попытка корректировать исходящее сообщение выполняется в строке 404. Если эта попытка не удается, выводится сообщение об ошибке и метод возвращает `false` (ложь), указывая на ошибку.

Листинг 18.14. Метод saveMessage() в PMessageDetailWindow

Метод saveMessage() в PMessageDetailWindow

```
338:    private boolean saveMessage(){
339:       IAContact contact =
             findContact(contacts,contactCmb.getSelectedItem().
                                                 toString());
340:       if(contact == null){
```

```
341:          JOptionPane.showMessageDialog(this,
                 "Please select contact information.",
                 "Contact Not Selected",JOptionPane.
                                        INFORMATION_MESSAGE);
342:          contactCmb.requestFocus();
343:          return false;
344:       }
           ... [вырезано для сохранения места]
397:       msg.setSenderEmployee(employee);
398:       msg.setSubject(subject);
403:       try{
404:          admin.updateMessage(msg);
405:       }catch(Exception exc){
406:          exc.printStackTrace();
407:          JOptionPane.showMessageDialog(this,"Fail to save
                 message, please check that all fields
                 are filled.","Fail to Save",JOptionPane.
                                             ERROR_MESSAGE);
408:          return false;
409:       }
410:       return true;
411:    }
```

Листинг 18.15. Метод okBtnActionPerformed() в PMessageDetailWindow

Метод okBtnActionPerformed() в PMessageDetailWindow

```
413:    private void okBtnActionPerformed(ActionEvent evt) {
414:          if("Delete".equals(okBtn.getText())) admin.
                                                 deleteMessage(msg);
415:          closeDialog(null);
416:    }
```

Кнопка OK служит двум целям (листинг 18.15). Первая — как признак того, что пользователь закончил свои операции с исходящим сообщением наподобие того, как это было уже рассмотрено. Вторая — признак того, что пользователь хотел бы удалить текущее сообщение, если PMessageDetailWindow находится в нередактируемом состоянии (строка 414).

Листинг 18.16. Метод populateContact() в PMessageDetailWindow

Метод populateContact() в PMessageDetailWindow

```
434:    private void populateContact(){
435:       contactCmb.removeAllItems();
438:       contacts = admin.listContacts();
439:       java.util.Iterator it = contacts.iterator();
```

```
440:        while(it.hasNext()){
441:           IAContact c = (IAContact) it.next();
442:           contactCmb.addItem(c.getFamilyName()+
                                        ","+c.getFirstName())
443:        }
444:     }
```

Размещение содержимого различных элементов управления типа комбинированных строк ввода делового партнера или служащего показано в листинге 18.16. Список деловых партнеров извлекается из `CAdmin` (класс «администратор» пакета `control`) в строке 438. Затем он повторяется и добавляется к комбинированной строке ввода `contactCmb` — комбинированная строка ввода делового партнера (строка 442).

18.3.3. Класс PMessageTableModel

Класс `PMessageTableModel` (класс «модель таблицы для сообщений» пакета `presentation`) представляет модель для `JTable` (класс «таблица» пакета Swing) — рис. 18.7. Модель используется, чтобы хранить исходящие сообщения, фильтровать исходящие сообщения, обеспечивать сервисы классу `JTable` для данных, которые будут отображены, и, наконец, выполнять сортировку.

Методы типа `getColumnCount()` (получить число столбцов), `getColumnName()` (получить имя столбца), `getColumnClass()` (получить класс столбца), `isCellEditable()` (является ли ячейка редактируемой), `getRowCount()` (получить количество строк), `getRow()` (получить строку), `getValueAt()` (получить значение...), `clear()` (очистить) являются переопределенными методами, обычно обеспечивающими сервисы для `JTable`. Пожалуйста, читайте документацию по Java, содержащую их описания.

Листинг 18.17. Поля в PMessageTableModel.java

```
Поля в PMessageTableModel.java

16:     public static final Object LAST_DAYS_FILTER =
                                          new Integer(0);
17:     public static final Object TODAY_FILTER =
                                          new Integer(1);
18:     public static final Object FUTURE_DAYS_FILTER =
                                          new Integer(2);
19:     public static final Object ALL_DAYS_FILTER =
                                         new Integer(3);
20:     public static final Object NO_CONTACT_FILTER =
                                       "No contact filter";
22:     private PDisplayList data;
23:     private int columnCount = 5;
24:     private boolean[] ascending = {true,true,true,
                                             true,true};
```

```
25:     private String[] columnName ={"Contact Name",
           "Message Subject",
           "Schedule Date",
           "Schedule Employee", "Scheduled Department"};
26:     private Class[] types = new Class [] {
27:        java.lang.String.class, java.lang.String.class,
           java.util.Date.class, java.lang.String.class,
           java.lang.String.class
28:     };
29:     boolean[] canEdit = {false,false,false,false,false};
```

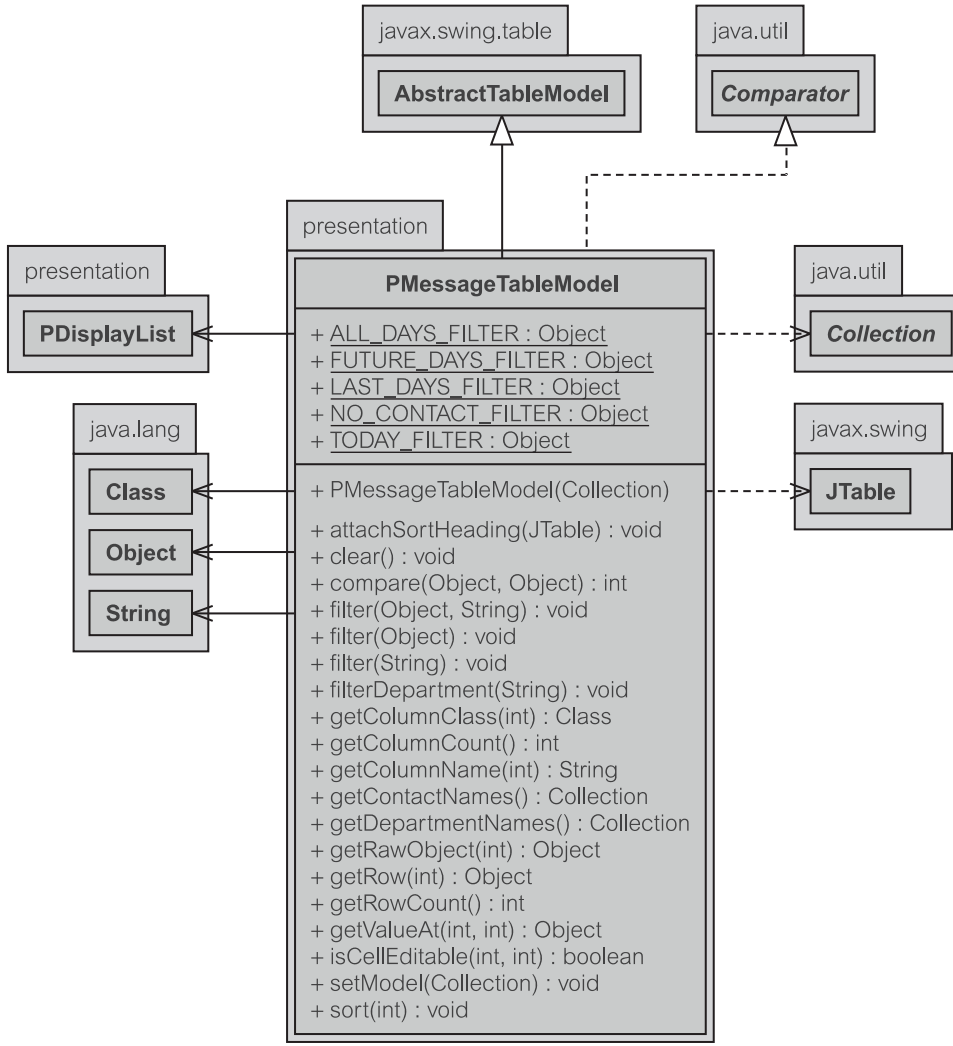


Рис. 18.7. PMessageTableModel.java

Некоторые статические переменные определены в классе PMessageTableModel (класс «модель таблицы для сообщений» пакета presentation), чтобы обеспечить возможность фильтрации клиентам класса (листинг 18.17). Класс хранит свои данные в PDisplayList — класс «список отображения» пакета presentation (строка 22). В настоящее время имеются только пять столбцов (строка 23) с определенными именами (строка 25) и сортировкой (строка 24). Типы столбцов, необходимые для getColumnClass(), определены в строках 26–27. Все столбцы нередактируемы (строка 29).

Параметры метода filter() (фильтрация) обеспечивают информацию о том, выполнена ли фильтрация по деловым партнерам или по датам (листинг 18.18). В зависимости от типа требуемой фильтрации метод setFilter() (задать фильтр) вызывается из объекта data (данные), который является экземпляром PDisplayList. Как только фильтрация будет выполнена, таблица должна быть обновлена через вызов fireTableDataChanged() (инициализация измененных данных таблицы).

Однако если фильтрацию нужно выполнить только по деловым партнерам, есть существенно более простой метод setFilterContact() (задать фильтрацию по деловым партнерам) — листинг 18.19, строка 63. Данные фильтруются только по заданному деловому партнеру, и таблица соответственно корректируется. Подобный подход применяется и к фильтрации по отделам (в этой главе не рассматривается).

Сортировка сообщений в PMessageTableModel — весьма интересная задача (листинг 18.20). Прежде всего, модель способна обрабатывать много столбцов сортировки, как указано ее переменной ascending — возрастание (строка 153). Она (модель) использует класс java.util.Collection (коллекция), чтобы выполнить сортировку (строка 154), и запускает свой собственный метод compare() — сравнить (реализация Comparable — сравниваемый). Метод compare() получает соответствующие значения из входных параметров (которые являются экземплярами PDisplayData — класс «отображаемые данные» пакета presentation, что будет объяснено позже). Статические методы класса PDisplayList, типа getContactName() — получить имя делового партнера (строки 166–167), используются, чтобы получить соответствующие величины из v1 и v2. Как только величины будут получены, они сравниваются, чтобы определить, является ли o1 больше, чем o2 или наоборот (строка 186). В случае, когда возникает исключение, сравнение основывается исключительно на том, является ли текущий столбец возрастающим или убывающим (строки 187–189).

Листинг 18.18. Метод filter() в PMessageTableModel — дата и деловой партнер

Метод filter() в PMessageTableModel — дата и деловой партнер

```
44:     public void filter(Object datefilter,
                                String contactfilter) {
45:       if(contactfilter == NO_CONTACT_FILTER)
                                     contactfilter = null;
```

```
48:       if(ALL_DAYS_FILTER.equals(datefilter)){
49:         data.setFilter(3,contactfilter);
50:       }else if(LAST_DAYS_FILTER.equals(datefilter)){
51:         data.setFilter(0,contactfilter);
52:       }else if(TODAY_FILTER.equals(datefilter)){
53:         data.setFilter(1,contactfilter);
54:       }else if(FUTURE_DAYS_FILTER.equals(datefilter)){
55:         data.setFilter(2,contactfilter);
56:       }
58:       fireTableDataChanged();
59:     }
```

Листинг 18.19. Метод filter() в PMessageTableModel — только деловой партнер

Метод filter() в PMessageTableModel — только деловой партнер

```
60:    public void filter(String contactfilter){
61:       if(contactfilter == NO_CONTACT_FILTER)
                                        contactfilter = null;
63:       data.setFilterContact(contactfilter);
64:       fireTableDataChanged();
65:    }
```

Листинг 18.20. Метод sort() в PMessageTableModel

Метод sort() в PMessageTableModel

```
149:    int sortColumn;
150:    public void sort(int column){
151:       if(data == null) return;
152:       sortColumn = column;
153:       ascending[column] = !ascending[column];
154:       Collections.sort(data.getData(), this);
155:       fireTableDataChanged();
156:    }
161:    public int compare(Object v1, Object v2) {
162:       Comparable o1 = null, o2 = null;
163:       try{
164:          switch(sortColumn){
165:             case 0:
166:                o1 = PDisplayList.getContactName(v1);
167:                o2 = PDisplayList.getContactName(v2);
168:                break;
                  ...[вырезано для сохранения места]
185:          }
```

```
186:           return ascending[sortColumn]?
                  o1.compareTo(o2):o2.compareTo(o1);
187:        }catch(Exception exc){
188:           return o1 == null?
                  (ascending[sortColumn]?-1:1):(ascending
                                                 [sortColumn]?1:-1)
189:        }
190:     }
```

18.3.4. Класс PDisplayList

`PDisplayList` — класс «список отображения» пакета `presentation` (рис. 18.8) представляет собой класс, используемый для представления списка исходящих сообщений. Он имеет некоторые общие утилиты для извлечения информации относительно исходящих сообщений. Данные, размещенные в `PDisplayList`, находятся в формате `PDisplayData` (класс «отображаемые данные» пакета `presentation`). Методы вроде `getContactName()` (получить имя делового партнера), `getCreatorEmployeeName()` (получить создателя имени служащего), `getScheduledDate()` (получить намеченную дату), `getScheduledDepartment()` (получить намеченный отдел), `getSenderEmployeeName()` (получить имя отправляющего служащего), `getSubject()` (получить тему) являются представителями полезных статических методов. Возвращаемые значения вышеупомянутых методов являются экземплярами `Comparable` (сравнимый), что означает возможность сравнения их друг с другом.

Чтобы полностью поддержать механизм фильтрации, `PDisplayList` должен хранить данные в двух различных контейнерах (листинг 18.21). Первый называется `data` — данные (`ArrayList` — список-массив, строка 16). Второй называется `invisibleData` — невидимые данные (`Collection` — коллекция, строка 17). Использование `ArrayList` по отношению к `Collection` произвольно. `ArrayList` позволяет обеспечить прямой индексированный доступ, в то время как `Collection` это не позволяет. Класс «отображаемые данные» пакета `presentation`, размещенный в `data`, содержит *видимые* исходящие сообщения, отображаемые на экране, в то время как те, что находятся в `invisibleData`, *невидимы* на экране.

Листинг 18.21. Поля в PDisplayList.java

```
Поля в PDisplayList.java

16:     private ArrayList data;
17:     private Collection invisibleData;
18:     private int visibleSize = 0;
19:     private Filter filter;
```

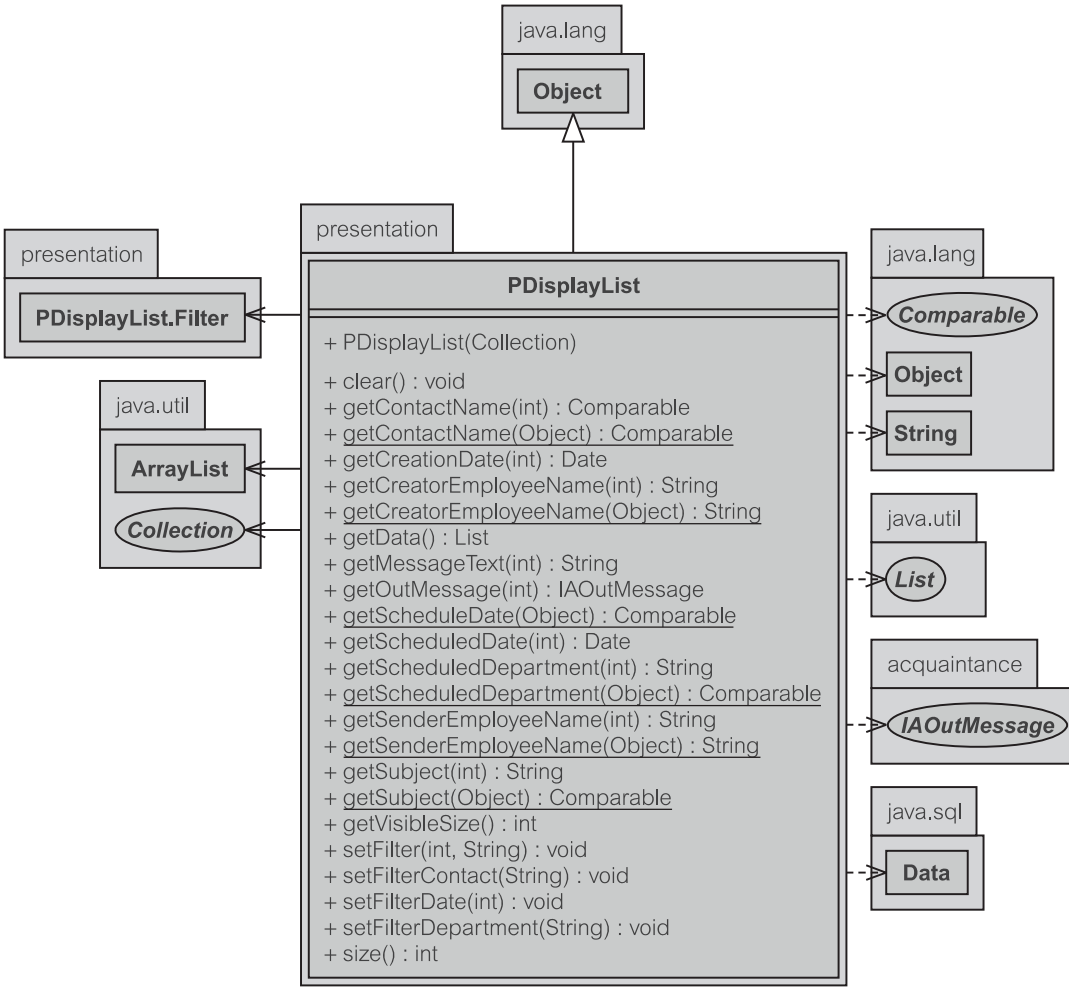


Рис. 18.8. PDisplayList.java

Объект `PDisplayList` построен для обеспечения создания списка исходящих сообщений (листинг 18.22). Он просматривает исходящие сообщения одно за другим (строка 29) и преобразует их в `PDisplayData`, чтобы добавить в список массива `data` — данные (строка 34). Объект `filter` (фильтр) создается в строке 41. Это обеспечивает общие функциональные возможности фильтра, рассмотренные ниже в контексте класса `PDisplayList.Filter`.

Методы, необходимые для пользователя как «получателя», типа `getOutMessage()` (получение исходящего сообщения), передаются к `PDisplayData` из коллекции данных (листинг 8.23, строка 71). Многие другие методы «получателя» работают подобным образом.

Листинг 18.22. Конструктор в PDisplayList

Конструктор в PDisplayList

```
22:     public PDisplayList(Collection msgs) {
24:        data = new ArrayList(msgs.size());
25:        IAOutMessage msg = null;
26:        IAContact contact;
27:        IAEmployee emp;
28:        Iterator it=msgs.iterator();
29:        while(it.hasNext()){
30:           try{
31:              msg = (IAOutMessage) it.next();
32:              emp = msg.getSenderEmployee();
33:              contact = msg.getContact();
34:              data.add(new PDisplayData(msg,emp,contact);
35:           }catch(Exception exc){
36:              exc.printStackTrace();
37:           }
38:        }
39:        visibleSize = data.size();
41:        filter = new Filter();
42:        invisibleData = new ArrayList();
43:     }
```

Листинг 18.23. Метод getOutMessage() в PDisplayList

Метод getOutMessage() в PDisplayList

```
70:     public synchronized IAOutMessage getOutMessage(int
                                                        index){
71:        PDisplayData d = (PDisplayData) data.get(index);
72:        return d.getOutMessage();
73:     }
```

Фильтрация по отделу или деловому партнеру сделана так, как показано в листинге 18.24 на строке 148. Как только фильтрация будет выполнена, отображение должно быть обновлено с помощью `updateVisible()` (скорректировать видимые данные), чтобы вывести новые данные.

Листинг 18.24. Метод setFilterDepartment() в PDisplayList

Метод setFilterDepartment() в PDisplayList

```
147:    public synchronized void setFilterDepartment(String
                                                     deptCode){
148:       filter.setFilterDept(deptCode);
149:       updateVisible();
150:    }
```

Для выполнения фильтрации применяется следующее правило. Все data помещаются в коллекцию invisibleData и удаляются из видимой коллекции (листинг 18.25, строки 158–159). Затем данные просматриваются, строка за строкой (строка 163), и передаются методу filter() класса PDisplayList (строка 165). Если метод filter() возвратит значение true (истина), соответствующее исходящее сообщение добавляется к коллекции data и удаляется из invisibleData.

Листинг 18.25. Метод updateVisible() в PDisplayList

Метод updateVisible() в PDisplayList

```
156:    private void updateVisible(){
158:       invisibleData.addAll(data);
159:       data.clear();
160:       visibleSize = 0;
162:       Iterator it = invisibleData.iterator();
163:       while(it.hasNext()){
164:          Object o = it.next();
165:          if(filter.filter(o)){
166:             data.add(o); //помещение в видимую коллекцию
167:             it.remove(); //удаление из невидимой коллекции
168:             visibleSize++;
169:          }
170:       }
171:    }
```

18.3.5. Класс PDisplayList.Filter

Filter (фильтр) является внутренним классом PDisplayList (класс «список отображения» пакета presentation). Он обеспечивает фильтрацию исходящих сообщений (листинг 18.26). Сигнатуры методов показывают, что каждый метод setFilter() (задать фильтр) обеспечивает свою фильтрацию, типа фильтра по дате, фильтра по деловому партнеру, а также фильтра по отделу.

Листинг 18.26. Сводка методов класса PDisplayList.Filter

```
boolean  filter(java.lang.Object o)
void     setFilter(int datetype)
void     setFilter(int datetype, java.lang.String contactName)
void     setFilter(java.lang.String contactName)
void     setFilterDept(java.lang.String deptCode)
```

В зависимости от типа фильтрации метод filter() выбирает различные стратегии сравнения (листинг 18.27). Если установлен параметр datetype (тип даты), то сообщения первоначально фильтруются по дате (строка 213).

Дата исходящего сообщения (строка 211 в `getScheduledDate()` — получить намеченную дату) — сравнивается с текущей датой, чтобы определить, является ли исходящее сообщение *прошедшим*, *текущим* или *будущим* (строки 215–219, но здесь не все показано). Как только исходящее сообщение будет отфильтровано по дате, оно далее проверяется с помощью фильтра кода отдела (строка 239). В конце обработки метод объявляет, пропускает ли фильтр исходящее сообщение (`selected` — отобрано) или нет.

Листинг 18.27. Метод filter() в Filter

Метод filter() в Filter

```
207:  public boolean filter(Object o){
208:     PDisplayData d = (PDisplayData) o;
209:     Calendar today = new GregorianCalendar();
210:     Calendar oCal = new GregorianCalendar();
211:     oCal.setTime(d.getScheduledDate());
212:     boolean selected = false;
213:     switch(datetype){
214:        case 0: //прошедшее
215:           selected = oCal.before(today) &&
                  !isToday(today,oCal);
216:           break;
           ...[вырезано для сохранения места]
223:        case 3: //все
224:           selected = true;
225:           break;
226:        default:
227:           selected = true;
228:           break;
229:     }
232:     if(selected && contactName != null){
233:        try{
234:         selected=contactName.indexOf(
              d.getContactFamilyName().toLowerCase()) != -1 ||
235:         contactName.indexOf(
              d.getContactFirstName().toLowerCase()) != -1;
237:        }catch(Exception exc){}
238:     }
239:     if(selected && deptCode != null){
240:        try{
241:           selected = d.getScheduledDepartmentCode()
                  .toLowerCase().equals(deptCode);
242:        }catch(Exception exc){}
243:     }
244:     return selected;
245:  }
```

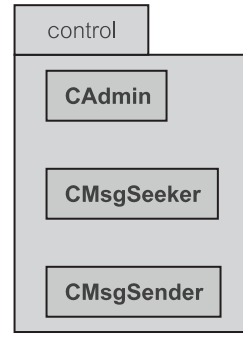


Рис. 18.9. Классы в пакете Control

18.4. Пакет Control

Пакет `control` состоит из трех классов (рис. 18.9). Многое из поведения `CAdmin` (класс «администратор» пакета `control`), `CMsgSeeker` (класс «поиск сообщения» пакета `control`) и `CMsgSender` (класс «передача сообщения» пакета `control`) было рассмотрено ранее в главах 13, 14 и 15.

18.4.1. Класс CAdmin

Как уже упоминалось, класс `CAdmin` — класс «администратор» пакета `control` (рис. 18.10) не имеет никаких существенных изменений по сравнению с итерацией 1 (глава 13). Дополнительные методы — `newMessage()` (новое сообщение), `getMsgSeeker()` (получение поиска сообщений), `getMsgSender()` (получение передачи сообщения), `listContacts()` (список деловых партнеров) и `listEmployees()` (список служащих). Списки служащих и деловых партнеров нужны, чтобы заполнить различные комбинированные строки ввода в пакете `presentation` (представление). Метод `newMessage()` (новое сообщение) используется, чтобы создать новое исходящее сообщение, которое будет размещено в БД.

18.4.2. Класс CMsgSeeker

Сервисы класса `CMsgSeeker` — класс «поиск сообщения» пакета `control` (рис. 18.11) были проанализированы в главе 15, где было принято решение использовать рефакторинг, чтобы разделить `CActioner` (класс «исполнитель» пакета `control`) на `CMsgSeeker` и `CMsgSender` (класс «передача сообщения» пакета `control`). Имеются три различных метода для поиска исходящих сообщений: исходящие сообщения для текущего служащего, для отдела текущего служащего или для всех исходящих сообщений независимо от служащего или отдела.

Листинг 18.28 представляет фрагменты кода, связанные с тремя методами извлечения. Обратите внимание, что исходящие сообщения, уже содержащиеся в коллекции `msgs` (сообщения), заново не будут извлекаться (строки 64–68).

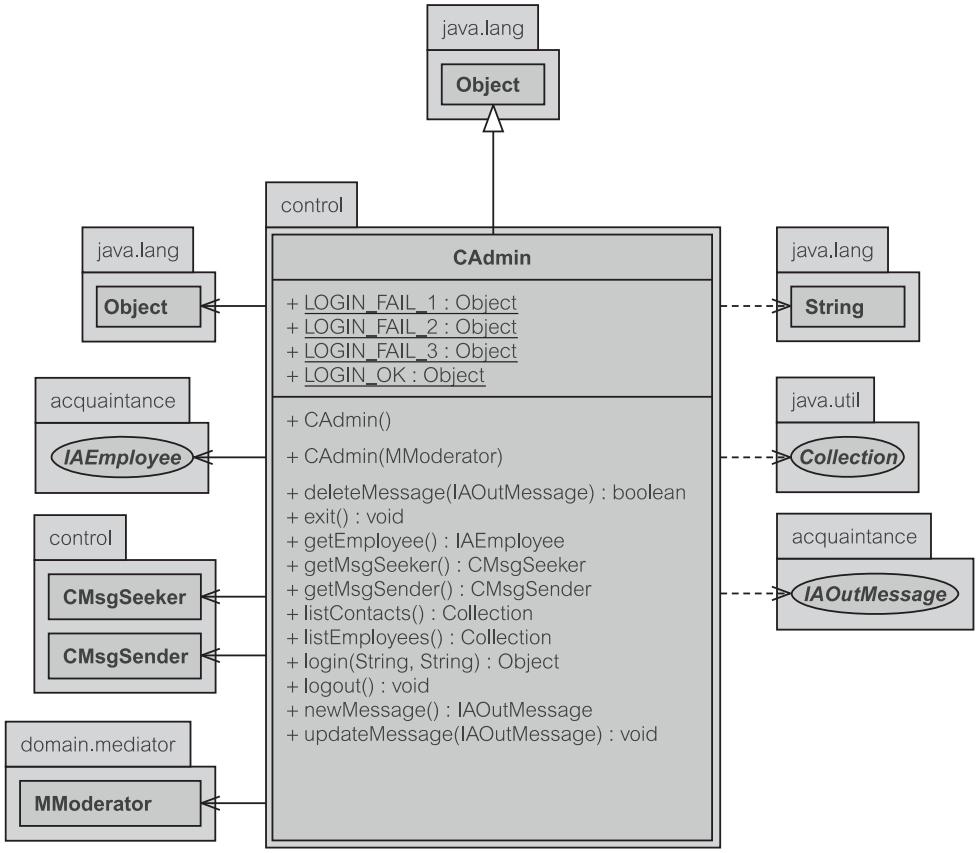


Рис. 18.10. Класс CAdmin в EM 2

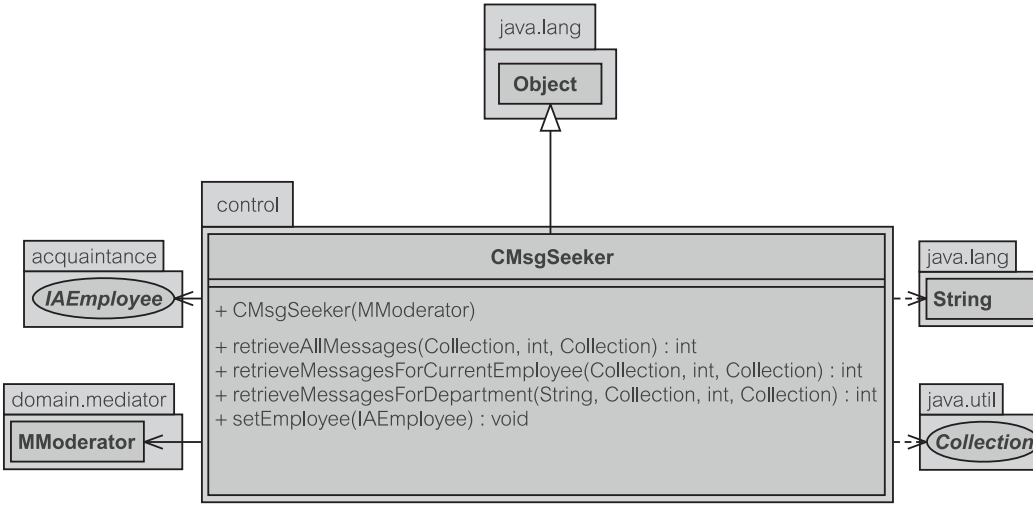


Рис. 18.11. Класс CMsgSeeker в EM 2

Листинг 18.28. Методы извлечения сообщений в CMsgSeeker

Методы retrieveMessagesForCurrentEmployee(), retrieveMessagesForDepartment() и retrieveAllMessages() в CMsgSeeker

```
48:     public int retrieveMessagesForCurrentEmployee(
49:        Collection msgs,
50:        int numMsgs,
51:        Collection msgIDs) {
53:        int remainder = moderator.retrieveUnsentMessages(
              emp, msgs, numMsgs);
56:        if (msgIDs == null || msgIDs.isEmpty())
57:           return remainder;
60:        Iterator it = msgs.iterator();
61:        while (it.hasNext()) {
62:           IAOutMessage msg = (IAOutMessage) it.next();
63:           Integer id = new Integer(msg.getMessageID());
64:           if (!msgIDs.contains(id)) {
65:              it.remove();
66:              remainder++;
68:           }
69:        }
70:        return remainder;
77:     }
91:     public int retrieveMessagesForDepartment(String
                                        departmentCode,
92:        Collection msgs,
93:        int numMsgs,
94:        Collection msgIDs) {
96:        int remainder = moderator.retrieveUnsentMessages(
                  departmentCode, msgs, numMsgs);
               ...[то же самое, что и строки 56-77]
134:     public int retrieveAllMessages(
135:        Collection msgs,
136:        int numMsgs,
137:        Collection msgIDs) {
139:        int remainder = moderator.
                  retrieveUnsentMessages(msgs,
                  numMsgs);
              ...[ то же самое, что и строки 56-77]
```

18.5. Пакет Entity

Наиболее заметные изменения в пакете `entity` (сущность) — это добавление интерфейса `IEObjectID` (интерфейс «идентификатор объекта» пакета `presentation`) и класса `EIdentityMap` (класс «коллекция идентичности

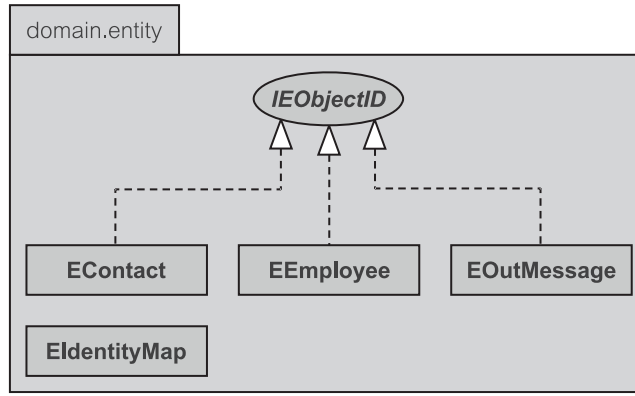


Рис. 18.12. Классы в пакете Entity

объектов» пакета `entity`) — рис. 18.12. Изменения в других объектах пакета `entity` минимальны. Они касаются включения необходимого кода, чтобы помочь классу `EIdentityMap` управлять **кэшем** (свойством измененности объектов) и задать надлежащие ссылки между объектами пакета `entity`, то есть, использовать **навигацию по классам пакета entity** в противоположность навигации по коллекции идентичности объектов (раздел 15.3.4). Имеется ассоциация от `EIdentityMap` к `IEObjectID`, но ассоциация поддерживается в коллекциях и поэтому соответственно не показана на рис. 18.12. Обратитесь к разделу 18.2 «Пакет Acquaintance», чтобы посмотреть описания методов, реализованных в этом пакете.

`IEObjectID` обеспечивает упрощенный интерфейс, чтобы определить свойство OID (идентификатор объекта). `IEObjectiD` выполняет два метода: `getOID()` (получить OID) и `setOID(int)` (задать OID). Другие объекты пакета `entity` поддерживают представление OID, гарантируя, что связи ассоциации проходят через последовательность OID в каждом объекте пакета `entity`. Например, это возможно для объекта `EOutMessage` (класс «исходящее сообщение» пакета `entity`), который может быть запрошен (через `getContactOID()` — получить идентификатор делового партнера) его объектом `EContact` (класс «деловой партнер» пакета `entity`), так как объект `EOutMessage` хранит в качестве ссылки `contactOID` (идентификатор объекта «деловой партнер»). Однако реальный `IAContact` (интерфейс «деловой партнер» пакета `acquaintance`) может быть, а может и не быть загружен в память. Это — реализация паттерна **Заместитель идентификатора объекта**, рассмотренного в разделе 15.3.4. Использование OID объясняется в разделе 13.5.1 «Идентификаторы объектов и паттерн Поле идентификации».

Кроме вышеупомянутых дополнительных методов класс `EEmployee` (класс «служащий» пакета `entity`) содержит новое свойство `department` (отдел). Это свойство используется в спецификации итерации 2, чтобы идентифицировать исходящие сообщения, которые назначены для отдела конкретного служащего (подпоток *S1.2 — View Department Unsent* — представление непосланных отделом сообщений, раздел 14.2.3).

18.5.1. Класс Коллекция идентичности объектов

Класс `EIdentityMap` — класс «коллекция идентичности объектов» пакета `entity` (рис. 18.13) рассматривается в разделе 15.3.1. Его цель состоит в том, чтобы хранить кэш объектов пакета `entity` (сущность), которые должно использовать приложение.

`EIdentityMap` хранит объекты пакета `entity` через различные соответствия между PK (первичный ключ) каждой сущности, его OID (идентификатор объекта) и непосредственно объектом пакета `entity`. Строки 20–22 в листинге 18.29 используются только внутренне классом `EIdentityMap`, чтобы гарантировать, что объекты пакета `entity` не будут загружены больше чем один раз. Главная коллекция — коллекция `OIDToObj` (OID к объекту). Она используется, чтобы получить объект, идентифицированный значением OID.

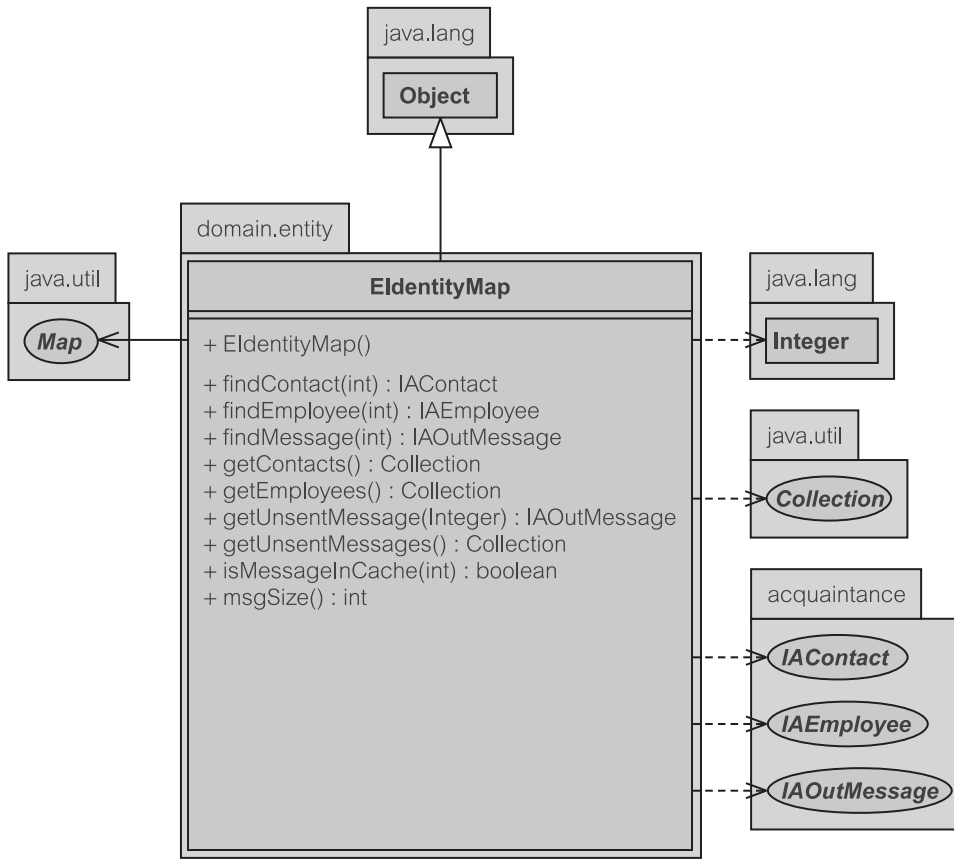


Рис. 18.13. Класс EIdentityMap в EM 2

Листинг 18.29. Поля в EIdentityMap

Поля в EIdentityMap

```
19:     private Map OIDToObj; //OID -> Obj
20:     private Map msgPKToOID; //msgPK -> OID
21:     private Map empPKToOID; //empPK -> OID
22:     private Map ctcPKToOID; //ctcPK -> OID
```

Метод `registerContact()` — регистрация делового партнера (листинг 18.30) получает OID данного `oidObject` (объект с OID) и помещает его сначала в коллекцию деловых партнеров (строка 38) и, наконец, в `OIDToObj` (строка 39). Чтобы выполнить регистрацию для объектов-сообщений и служащего, следует заменить `ctcPKToOID` (первичный ключ в OID для делового партнера) на `empPKToOID` (первичный ключ в OID для служащего) или `msgPKToOID` (первичный ключ в OID для сообщения). Требуется единственная модификация, чтобы снять регистрацию объектов (служащего, делового партнера или исходящего сообщения) — изменить вызовы из коллекций метода *put* (поместить) на *remove* (удалить).

Листинг 18.30. Метод registerContact() в EIdentityMap

Метод registerContact() в EIdentityMap

```
36:     public void registerContact(IEObjectID oidObject) {
37:        Integer oid = new Integer(oidObject.getOID());
38:        ctcPKToOID.put(((IAContact)oidObject).
                        getContactID(),oid);
39:        OIDToObj.put(oid, oidObject);
40:     }
```

Листинг 18.31 показывает, как объект `EOutMessage` извлекается из кэша. Подобная технология используется для `findContact()` (найти делового партнера), `findEmployee()` (найти служащего), `getContact()` (получить делового партнера) и `getEmployee()` (получить информацию о служащем). Методы `findContact()`, `findEmployee()` и `findMessage()` — примеры использования коллекции **`OIDToObj`**.

Листинг 18.31. Методы getUnsentMessage() и findMessage() в EIdentityMap

Методы getUnsentMessage() и findMessage() в EIdentityMap

```
67:     public IAOutMessage getUnsentMessage(Integer msgID) {
68:        Integer oid = (Integer) msgPKToOID.get(msgID);
70:        if (oid == null)
71:           return null;
72:        return findMessage(oid.intValue());
73:     }
```

```
62:     public IAOutMessage findMessage(int msgOID) {
63:        return (IAOutMessage) OIDToObj.get(new
                                              Integer(msgOID));
64:     }
```

Листинг 18.32 показывает, как метод `getUnsentMessages()` (получить непосланные сообщения) просматривает в кэше список PK (первичные ключи) исходящих сообщений и запрашивает `findMessage()`, чтобы тот получил объект пакета `entity`. Другие методы получения служащих и деловых партнеров (`getEmployees()` — получить информацию о служащих и `getContacts()` — получить информацию о деловых партнерах) аналогичны. В то время как `getUnsentMessages()` извлекает информацию из коллекции `msgPKToOID`, `getContacts()` извлекает информацию из коллекции `ctcPKToOID`, а `getEmployees()` — из коллекции `empPKToOID`.

Листинг 18.32. Метод getUnsentMessages() в EIdentityMap

Метод getUnsentMessages() в EIdentityMap

```
76:   public Collection getUnsentMessages() {
77:      Collection c = new ArrayList();
78:      Iterator it = msgPKToOID.values().iterator();
79:      while (it.hasNext()){
80:         Object o = findMessage(((Integer)
                                it.next()).intValue());
81:         if(o != null) c.add(o);
82:      }
83:      return c;
84:   }
```

18.6. Пакет Mediator

Пакет `mediator` — посредник (рис. 18.14) подробно рассмотрен в главе 15. Здесь он включен для полноты.

Основные изменения в пакете `mediator` между итерацией 1 и итерацией 2, как объяснено в главе 15, включают следующее:

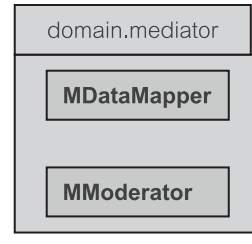


Рис. 18.14. Классы пакета Mediator из EM 2

- `MDataMapper` (класс «преобразователь данных» пакета `mediator`) представлен как класс, который отвечает за преобразование между строками в БД и их объектами пакета `entity` (сущность) в кэше.
- `MDataMapper` сотрудничает с `EIdentityMap` (класс «коллекция идентичности объектов» пакета `entity`). `EIdentityMap` используется как менеджер кэша для объектов пакета `entity`.
- Все запросы загрузки и выгрузки объектов пакета `entity` координируются с помощью объекта `MModerator` (класс «координатор» пакета `mediator`).

18.6.1. Класс MModerator

Класс `MModerator` — класс «координатор» пакета `mediator` (рис. 18.15) перехватывает все извлечения объектов пакета `entity` (сущность) и переадресовывает эти запросы к `EIdentityMap` (класс «коллекция идентичности объектов» пакета `entity`) или к `MDataMapper` (класс «преобразователь данных» пакета `mediator`). Извлечения сообщений делятся на три категории: извлечение по параметрам служащего, по параметрам отдела и, наконец, всех непосланных исходящих сообщений независимо от параметров относящегося к ним служащего или отдела.

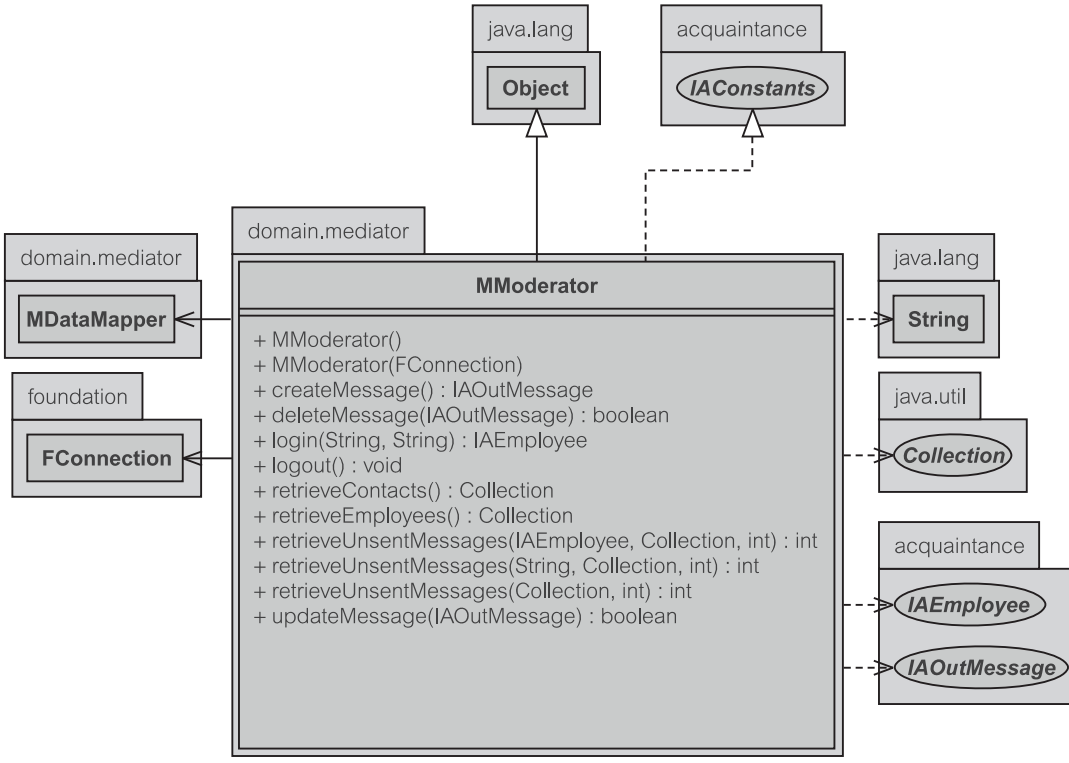


Рис. 18.15. Класс MModerator в EM 2

Листинг 18.33 показывает, как в MModerator реализована проверка регистрационного имени. Первоначально FConnection (класс «соединение» пакета foundation) проверяет, связан ли пользователь с БД (строка 72), и если нет, выполняется соединение (строка 74). Как только соединение будет установлено, данные о служащем извлекаются из MDataMapper (метод retrieveEmployeeByName() — извлечение информации о служащем по имени — в строке 80).

Листинг 18.33. Метод login() в MModerator

Метод login() в MModerator

```
70:   public IAEmployee login(String username, String passwd) {
71:      try {
72:         if (!connection.isConnected()) {
74:            if (!connection.connect(username, passwd))
75:               return null;
76:         }
80:         IAEmployee emp = mapper.retrieveEmployeeByName
                                                    (username);
82:         return emp;
83:      } catch (Exception exc) {
84:      }
85:      return null;
86:   }
```

18.6.2. Класс MDataMapper

Класс MDataMapper (класс «преобразователь данных» пакета mediator) обеспечивает сервисы для загрузки данных строк в объекты пакета entity (сущность) и выгрузки их в БД (рис. 18.16). Извлечения объектов уровня entity обеспечиваются с помощью вызовов методов типа retrieveEmployeeByID() (извлечение информации о служащем по идентификатору), retrieveContactByID() (извлечение делового партнера по идентификатору) и т. д. Метод unloadMesssage() (выгрузить сообщение) выгружает объекты-исходящие сообщения в БД. В настоящее время нет никакой необходимости иметь другие методы выгрузки — для делового партнера или служащего — поскольку они никогда не изменяются приложением.

Чтобы позволить объекту MDataMapper взаимодействовать с БД, требуются ассоциации с FReader (класс «чтение» пакета foundation) и FWriter (класс «запись» пакета foundation) — листинг 18.34. EIdentityMap (класс «коллекция идентичности объектов» пакета entity) служит как менеджер кэша для всех объектов пакета entity. Список OID (идентификатор объекта) ведется только с целью обеспечения того, чтобы но-

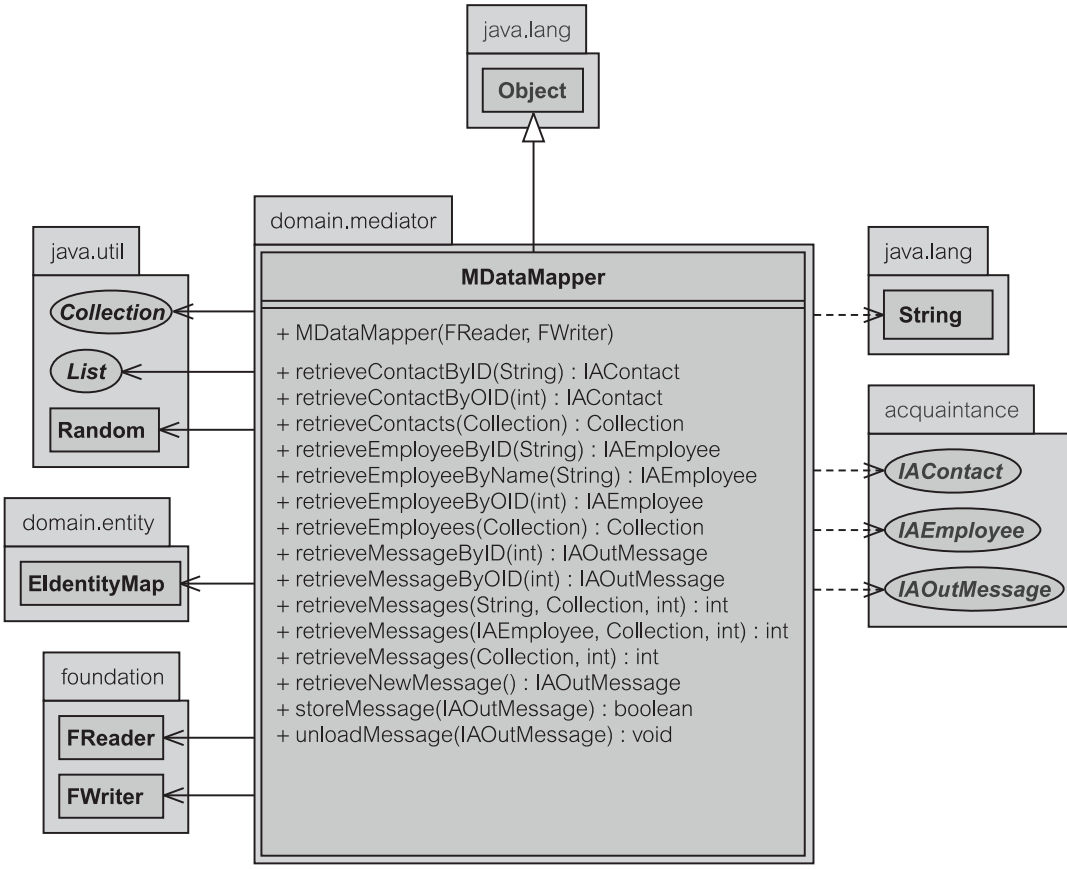


Рис. 18.16. Класс MDataMapper в EM 2

вые созданные OID не противоречили уже существующим. Случайное число, представленное параметром `rand` (случайный), используется, чтобы сформировать новые OID для новых исходящих сообщений, еще не размещенных в БД. Как только объект пакета `entity` будет размещен в БД, ему назначается надлежащий OID. Наконец, только что созданные исходящие сообщения размещаются в своих собственных коллекциях по имени `newMsgOID` (OID нового сообщения). Именно поэтому они требуют специальной обработки в смысле удаления или сохранения в пакете `entity`. Удаление только что созданных объектов пакета `entity` не требует удаления соответствующих записей в БД, пока они не размещены в БД. Иначе они должны быть удалены и из памяти, и из БД.

Листинг 18.34. Поля в MDataMapper

```
Поля в MDataMapper

33:     private FReader reader;
34:     private FWriter writer;
35:     private EIdentityMap cache;
36:     private List OIDs;
37:     private boolean contactFullyRetrieved,
                                    employeeFullyRetrieved;
38:     private Random rand;
39:     Private Collection newMsgOIDs;
```

Извлечение и загрузка исходящих сообщений

Имеются моменты, когда исходящее сообщение становится **измененным** и должно быть перезагружено. Листинг 18.35 показывает, как исходящее сообщение заново извлекается с помощью вызова метода `retrieveMessageByID()` — извлечение сообщения по идентификатору (строка 62). Должен использоваться специальный подход, если входной параметр `msg` (сообщение) определяет только что созданное исходящее сообщение (еще не размещенное в БД, строка 60). Если это только что созданное исходящее сообщение, то оно не перезагружается из БД (там нет никакой записи относительно этого сообщения).

Листинг 18.35. Метод reretrieveMessage() в MDataMapper

```
Метод reretrieveMessage() в MDataMapper

57: private IAOutMessage reretrieveMessage(IAOutMessage msg {
60:     if(newMsgOIDs.contains(
        new Integer(((EOutMessage)msg).getOID()))) return null;
62:    return retrieveMessageByID(msg.getMessageID());
63:   }
```

Итерация 2 обеспечивает три различных типа извлечения исходящего сообщения. Первый — извлечение исходящих сообщений, назначенных конкретному служащему, второй — для исходящих сообщений, назначенных служащим конкретного отдела, и третий — для всех непосланных исходящих сообщений независимо от их назначенных служащих или отделов (листинг 18.36).

Операции извлечения запрашивают исходящие сообщения из `EIdentityMap` (`cache` — кэш в строке 171). Просмотренные исходящие сообщения заново извлекаются, если они оказываются измененными (строки 175–176). Возвращается только некоторое подмножество исходящих сообщений. Оно зависит от величины `numMsgs` (число сообщений), как показано в методе `fillArrayUpto()` — заполнить массив до... (строка 180). Метод `retrieveMessages()` (извлечь сообщения) возвращает число исходящих

сообщений в кэше (и БД), чтобы вызывающий знал число исходящих сообщений, которые могут быть далее извлечены. Это число рассчитывается в строке 185. Метод `fillArrayUpto()` возвращает нуль, если общее количество в коллекции меньше, чем `numMsgs`, иначе он возвращает полное число исходящих сообщений в коллекции.

Извлечение исходящих сообщений из БД — основная обязанность метода `doRetrieveMessages()` — выполнить извлечение сообщений (листинг 18.37). Этот метод выполняет запрос извлечения объекта `FReader` (строка 200), высняет общее количество объектов, которые существуют в БД (строка 201), и заполняет кэш (строка 209) созданными объектами исходящих сообщений (строки 207–208). Исходящие сообщения помещаются в коллекцию результатов (строка 210).

Листинг 18.36. Метод retrieveMessages() в MDataMapper

Метод retrieveMessages() в MDataMapper — независимо от отдела и служащего

```
169:    public int retrieveMessages(Collection msgs,
                                                    int numMsgs) {
171:       Collection cc = cache.getUnsentMessages();
172:       ArrayList c = new ArrayList(cc.size());
173:       for (Iterator it = cc.iterator(); it.hasNext();) {
174:          IAOutMessage m = (IAOutMessage) it.next();
175:          if (m.isDirty())
                           //повторное извлечение сообщения,
                           //если оно изменено
176:             m = reretrieveMessage(m);
177:          if(m != null && !c.contains(m)) c.add(m);
178:       }
179:       String sql=
        "Select * from outmessage where date_emailed is null";
180:       int total = fillArrayUpto(numMsgs,c,sql);
181:       msgs.addAll(c);
185:       return total > 0 ? total - numMsgs : total;
186:    }
```

Листинг 18.37. Метод doRetrieveMessages() в MDataMapper

Метод doRetrieveMessages() в MDataMapper

```
191:    private int doRetrieveMessages(String sql,
              Collection results,
194:    int numMsgs) {
195:       int count = 0, total = 0;
197:       EOutMessage msg = null;
198:       java.sql.ResultSet rs = null;
```

```
199:        try {
200:           rs = reader.query(sql);
201:           total = rs.getStatement().getMaxRows();
202:           while (rs.next ()) {
203:              count++;
204:              if (count > numMsgs)
205:                 break;
207:              msg = (EOutMessage) createOutMessage();
208:              fillOutMessage(rs, msg);
209:              cache.registerMessage(msg);
210:              results.add(msg);
211:           }
             ...[вырезано для сохранения места]
216:        return total;
217:     }
```

MDataMapper содержит некоторые методы, позволяющие клиенту запрашивать данные. Метод retrieveMessageByOID() (извлечь сообщение по идентификатору объекта) возвращает исходящее сообщение, которое соответствует данному OID (или null, если исходящее сообщение не существует) — листинг 18.38. Метод просто переадресует запрос кэшу класса EIdentityMap. Другие методы типа retrieveEmployeeByOID() (извлечь служащего по идентификатору объекта) и retrieveContactByOID() (извлечь делового партнера по идентификатору объекта) выполняют аналогичные действия.

Листинг 18.38. Метод retrieveMessageByOID() в MDataMapper

Метод retrieveMessageByOID() в MDataMapper

```
237:     public IAOutMessage retrieveMessageByOID(int oid) {
238:        return cache.findMessage(oid);
239:     }
```

Создание нового объекта исходящего сообщения, который будет заполнен и затем помещен в БД, выполняется методом retrieveNewMessage() — извлечь новое сообщение (листинг 18.39). Новый объект исходящего сообщения создается (строка 221), и его OID (идентификатор объекта) формируется случайным образом (строка 224). OID должен быть уникальным для приложения и поэтому проверяется, не существует ли он в кэше (вызывая findMessage() — найти сообщение из EIdentityMap) — строка 225. ID (идентификатор) нового исходящего сообщения должен быть извлечен из БД, чтобы гарантировать, что исходящее сообщение имеет надлежащий первичный ключ (строка 228, retrieveNewMessageID() — извлечь идентификатор нового сообщения). Все только что созданные исходящие сообщения регистрируются в newMsgOID. Наконец, исходящее сообщение должно быть помечено как измененное и зарегистрировано в кэше (строки 231–232).

Листинг 18.39. 5Метод retrieveNewMessage() в MDataMapper

Метод retrieveNewMessage() в MDataMapper

```
220:    public IAOutMessage retrieveNewMessage(){
221:       EOutMessage msg = (EOutMessage) createOutMessage();
222:       int msgOID;
223:       do{ //попытка сохранить, когда используется msgOID
224:          msgOID = rand.nextInt();
225:       }while(cache.findMessage(msgOID) != null);
227:       msg.setOID(msgOID);
228:       msg.setMessageID(retrieveNewMessageID());
229:       newMsgOIDs.add(new Integer(msgOID));
231:       cache.registerMessage(msg);
232:       msg.setDirty(true);
233:       return msg;
234:    }
```

Листинг 18.40. Метод unloadMessage() в MDataMapper

Метод unloadMessage() в MDataMapper

```
134:  public void unloadMessage(IAOutMessage msg)
                                          throws Exception{
135:     msg.setDirty(true);
136:     cache.unregisterMessage((IEObjectID)msg);
138:     Integer oid = new Integer(((EOutMessage)msg).
                                                 getOID());
139:     if(newMsgOIDs.contains(oid)) {
140:        newMsgOIDs.remove(oid);
141:        return;
142:     }
144:     PreparedStatement st = null;
145:     try{
146:        st = writer.delete(
              "delete from outmessage where message_id = ? ");
147:        st.setInt(1, msg.getMessageID());
148:        st.execute();
           ...[вырезано для сохранения места]
```

Сохранение и выгрузка исходящего сообщения

Выгрузка сообщения требует двух шагов (листинг 18.40). Первый должен выгрузить его из кэша, как сделано в строке 136. Второй — удаление из БД, как показано в строках 146–148. Если сообщение — только что созданное сообщение, которое еще не размещено в БД, то удалять его из БД нет необходимости (строки 139–142).

Листинг 18.41 показывает, как исходящее сообщение возвращается в БД. Первоначально исходящее сообщение должно быть проверено, чтобы удостовериться, что оно только что создано и не существует в БД (строки 278–282). Если это новое сообщение, выполняется SQL-оператор размещения (строка 287), иначе должна быть выполнена корректировка уже существующих данных (строка 288). Строки 290–306 подготавливают `PreparedStatement` (подготовленный оператор) для размещения в БД в зависимости от того, является ли это оператором размещения или корректировки (строки 302 и 306). Если размещение произошло успешно, то `messageOID` (идентификатор объекта сообщения) удаляется из накопителя элементов `newMsgOID` (строка 305).

Имеется другой возможный способ корректировки исходящего сообщения. Исходящее сообщение может быть помечено как отосланное (строка 308, `getSentDate()` — получить дату отсылки). Если исходящее сообщение будет помечено как отосланное, то необходимо удалить его из кэша (строки 310–323). EM-приложение имеет дело только с непосланными исходящими сообщениями. Однако если исходящее сообщение не отмечено как посланное, то его необходимо заново зарегистрировать (строки 325–330), чтобы заменить старую ссылку в кэше.

Листинг 18.41. Метод storeMessage() в MDataMapper

Метод storeMessage() в MDataMapper

```
274:   public boolean storeMessage(IAOutMessage msg) {
275:     boolean newMsg = false;
276:     Integer oid = new Integer(((EOutMessage)msg).
                                        getOID());
277:     try {
278:        if(newMsgOIDs.contains(oid)){
                        //поместить новое сообщение
279:             newMsg = true;
282:        }
285:        String sql = null;
287:        if(newMsg) sql = "insert into outmessage(
              sender_emp_id,creator_emp_id,contact_id,
              message_subject,message_text,date_scheduled,
              date_emailed,department_code,message_id,date_
                                                    created)
              values(?,?,?,?,?,?,?,?,?,?)";
288:       else sql = "Update OutMessage set sender_emp_id = ?,
              creator_emp_id = ?, contact_id = ?,
              message_subject = ?, message_text = ?,
              date_scheduled = ?, date_emailed = ?,
              department_code = ? where message_id = ?";
290:       PreparedStatement st = writer.prepareStatement(sql);
```

```
291:       if(msg.getSenderEmployeeID() == null)
              st.setNull(1,java.sql.Types.VARCHAR);
292:       else st.setString(1, msg.getSenderEmployeeID());
293:       st.setString(2, msg.getScheduledEmployeeID());
294:       st.setString(3, msg.getCreatorEmployeeID());
295:       st.setString(4,msg.getContactID());
296:       st.setString(5, msg.getSubject());
297:       st.setString(6, msg.getMessageText());
298:       st.setDate(7,msg.getScheduleDate());
299:       st.setDate(8,msg.getSentDate());
300:       st.setString(9, msg.getDepartmentCode());
301:       st.setInt(10, msg.getMessageID());
302:       if(newMsg){
303:          st.setDate(11,msg.getCreationDate());
304:          st.execute();
305:          newMsgOIDs.remove(oid);
306:       }else st.executeUpdate();
308:       if(msg.getSentDate() != null){
310:          msg.getContact().removeOutMessage(msg);
311:          msg.getCreatorEmployee().
                 removeOutMessageCreator(msg);
312:          msg.getScheduledEmployee().
                 removeOutMessageScheduler(msg);
313:          msg.getSenderEmployee().
                 removeOutMessageSender(msg);
315:          msg.setContact(null);
316:          msg.setCreatorEmployee(null);
317:          msg.setSenderEmployee(null);
318:          msg.setScheduledEmployee(null);
319:          msg.setDirty(true);
321:          cache.unregisterMessage((IEObjectID) msg);
322:          return true;
323:       }
325:       cache.registerMessage((IEObjectID)msg);
328:       msg.getContact().addOutMessage(msg);
329:       msg.getCreatorEmployee().addOutMessageCreator(msg);
330:       msg.getSenderEmployee().addOutMessageSender(msg);
331:    } catch (Exception exc) {
332:       System.out.println("ERROR in update/insert");
333:       exc.printStackTrace();
334:       return false;
335:    }
336:    return true;
337: }
```

18.7. Уровень Presentation: версия апплета

Как рассмотрено в разделе 17.1.3, имеются два различных типа *апплетов* — тонкий апплет и толстый апплет. **Тонкий апплет** — множество классов, которые составляют приложение, выполняющее большинство своих операций на сервере. **Толстый апплет** — множество классов, эквивалентных приложению, выполняющему большинство своих операций на стороне клиента, а не на сервере.

Версия апплета, реализованная в итерации 2 учебного примера EM, является толстым апплетом. Он медленнее загружается, потому что большая часть его кода находится на клиенте. Однако его легче разработать, поскольку большая часть кода уже доступна в итерации 2 для пользовательского интерфейса рабочего стола. Фактически, единственными требуемыми изменениями должно быть изменение класса `PWindow` (класс «окно» пакета `presentation`), чтобы он наследовал от `Applet` (апплет), а не от `Frame` (структура), плюс некоторые дополнительные модификации, чтобы поддержать жизненный цикл `Applet`.

Как показано на рис. 18.17, `PWindow` изменен, чтобы наследовать от класса `JApplet` (класс «апплет» пакета Swing). Он все еще имеет ссылку на `Frame`, так как апплет должен быть загружен внутри `Frame`. `PWindow` должен быть реализован так, чтобы он мог запускаться и как приложение, и как апплет. Метод `main()` (главный) служит входной точкой, чтобы запустить программу как приложение. Методы типа `getAppletInfo()` (получить информацию об апплете), `getParameter()` (получить параметр) и `getParameterInfo()` (получить информацию о параметре) нужны для того, чтобы обеспечить различные сервисы, необходимые для реализации апплета, и должны быть доступными среде, в которой существует апплет (обычно Web-браузер или просмотрщик апплетов). Метод `init()` (инициализировать) добавлен, чтобы конструировать и запускать апплет в Web-браузере.

Конструирование `PWindow` почти такое же, как и в разделе 18.3.1. Единственное различие — удаление строк 37–45 (листинг 18.42). Эти строки перемещены в метод `init()`, гарантируя, что апплет пользовательского интерфейса появится на экране. Жизненный цикл апплета требует, чтобы все конструирование пользовательского интерфейса было помещено именно в секцию `init()`, а не в другие секции. Если строки 37–45 не будут перемещены в `init()` и оставлены в конструкторе, апплет не будет показывать никакого GUI, кроме простого незаполненного экрана.

Метод `GetParameter()` позволяет апплету запросить свою среду относительно некоторых параметров или свойств (листинг 18.43). Сигнатура `getParameter()` определяет, что метод найдет величину, соответствующую данному ключу, и если такой ключ в системе не будет найден, он возвратит значение по умолчанию (`def`). Если этот апплет выполняется как автономное приложение, то функция запрашивает свойства системы, которые поддерживает `java.lang.System` (строка 51). Однако если этот апплет выполняется как апплет, то запросы переадресуются через метод `getParameter()` родительского класса (строка 52). Строка 52 говорит, что

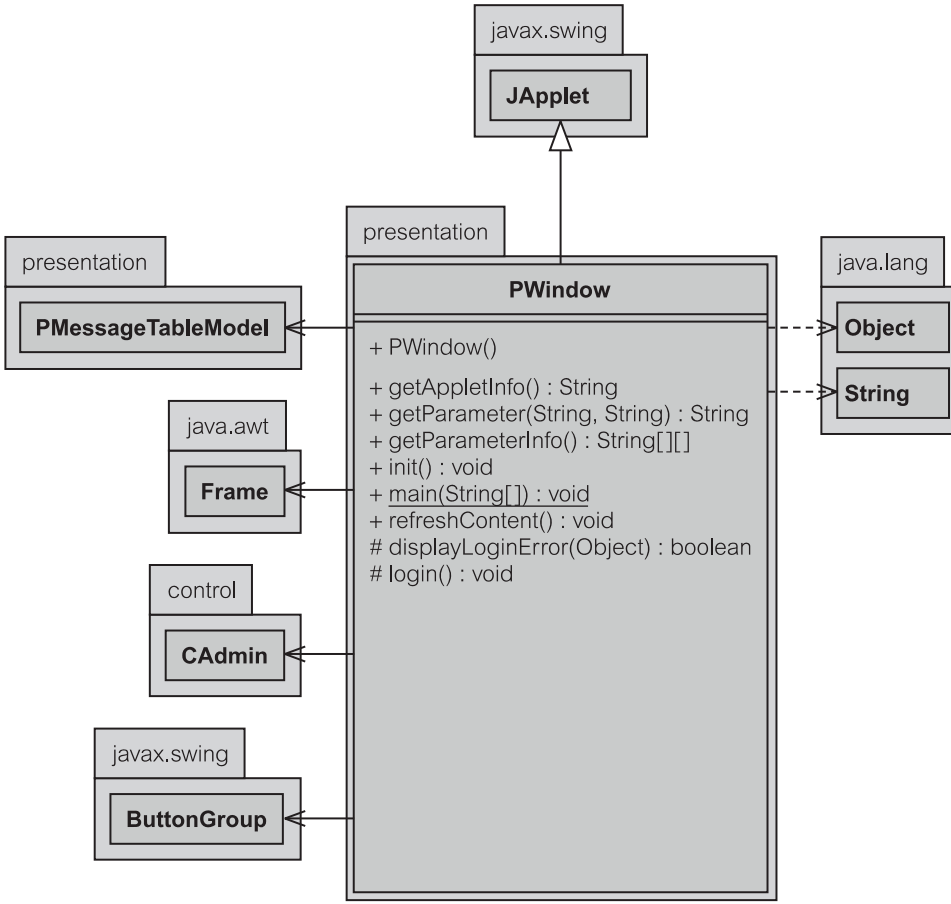


Рис. 18.17. Класс PWindow в апплете ЕМ 2

если `getParameter()` возвращает значение `null`, то вместо этого будет возвращено значение по умолчанию `def`.

Листинг 18.42. Конструктор в PWindow.java (апплет)

Конструктор в PWindow.java

```
31:     public PWindow() throws Exception {
32:        designMode = true;
33:        admin = new CAdmin();
34:
35:        //получение регистрационного имени
36:        login();
37:  /*       model = new PMessageTableModel(retrieveData(1));
38:        initComponents();
```

```
39:         model.attachSortHeading(viewTable);
                                          //добавить сортировку
40:         populateContact();
41:         populateDepartment();
42:         setButtonsEnable(false);
43:
44:         designMode = false;
45:   */
46:     }
```

Листинг 18.43. Метод getParameter() в PWindow (апплет)

Метод getParameter() в PWindow.java

```
50: public String getParameter(String key, String def) {
51:    return isStandalone ? System.getProperty(key, def) :
52:      (getParameter(key) != null ? getParameter(key) : def);
53:    }
```

Метод `init()` класса `PWindow` просто конструирует GUI объекта `PWindow`. Он не добавляет никакого специального кода и поэтому здесь не обсуждается.

18.8. Уровень Presentation: версия сервлета

Термин «сервлет» (servlet) является сокращением термина «серверный апплет» (server applet). Сервлет ведет себя подобно апплету, но расположен на сервере. Он обслуживает запросы пользователя на основе HTTP-стандарта. Сервлет отображает HTML-формат (или другой, допустимый в Web-технологии), доступный для просмотра пользователем. Версия сервлета итерации 2 учебного примера EM (раздел 17.5) требует модификаций на уровне `presentation` (представление). Все классы уровня `presentation` заменены двумя классами: `PEMS` (класс «сервлет EM» пакета `presentation`) и `PEMSEdit` (класс «редактор сервлета EM» пакета `presentation`).

Аналогично апплету, сервлет имеет некоторые методы, которые должны быть переопределены в реализации. Методы — `getServletInfo()` (получить информацию о сервлете), `doPost()` (выполнить передачу), `doGet()` (выполнить получение), `init()` (инициализировать) и `destroy()` (удалить). Метод `doPost()`, как считают, обслуживает действия *post* (передача) в HTML. Метод `doGet()` вызывается, чтобы ответить на действия *get* (получить) в HTML. По умолчанию действие *get* вызывается, когда сервлет загружается из Web-браузера, если конкретно не требуется *post*. Методы `init()` и `destroy()` используются, чтобы уведомить сервлет, что он готов к загрузке/выгрузке в/из среду(ы) (сервер) соответственно.

18.8.1. Класс PEMS

Класс `PEMS` (класс «сервлет ЕМ» пакета `presentation`) обеспечивает основное взаимодействие пользователя и подсистемы ЕМ (рис. 18.18). Кроме стандартных методов, поддерживающих жизненный цикл сервлета, `PEMS` также поддерживает действия отправления и просмотра исходящих сообщений.

В `PEMS` имеются три поля (листинг 18.44). Поля используются, чтобы определить, находится ли пользователь в режиме *View Department Messages* — просмотр сообщений отдела (строка 27), *View Own Messages* — просмотр собственных сообщений (строка 28) или *View All Messages* — просмотр всех сообщений (строка 29).

Метод `init()` класса `PEMS` (листинг 18.45) отличается от его копии в аплете (`PWindow` — класс «окно» пакета `presentation`). В `PWindow` необходимо поместить весь код конструирования GUI в `init()`. В сервлете это больше не применимо, поскольку GUI сервлета выводится только тогда, когда пользователь запрашивает его либо через `doGet()` (выполнить получение), либо через `doPost()` (выполнить передачу). Вот почему `init()` в `PEMS` вызывает только `init()` своего суперкласса, чтобы инициализировать сервлет.

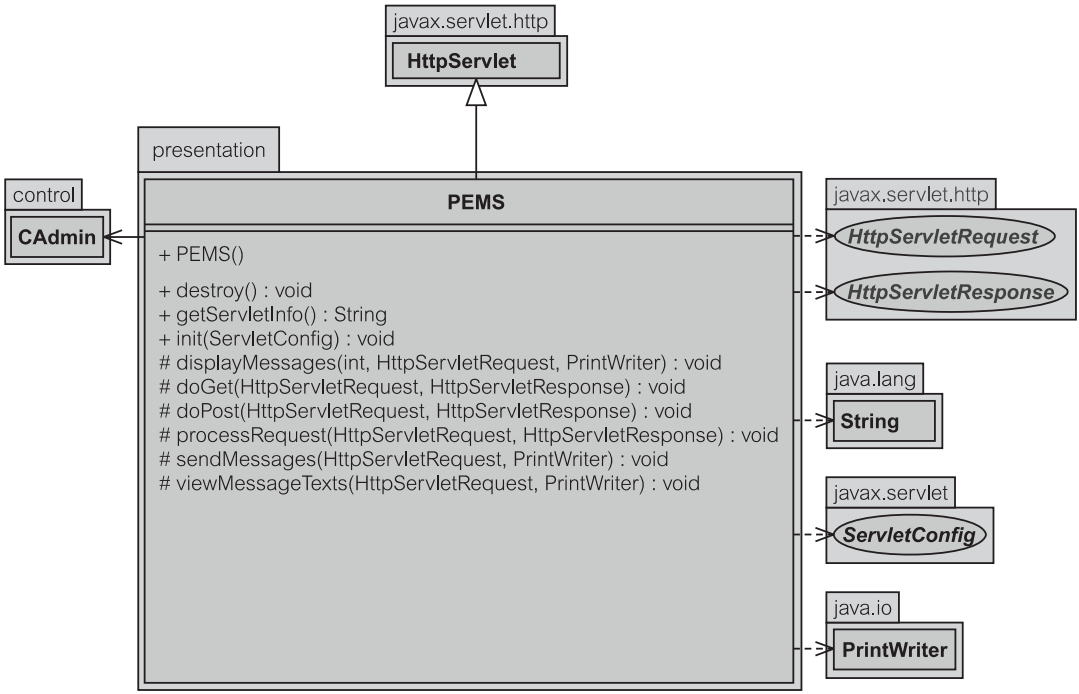


Рис. 18.18. PEMS в ЕМ 2 (сервлет)

Листинг 18.44. Поля в PEMS.java

```
Поля в PEMS.java

25:     public class PEMS extends HttpServlet {
26:        //отображать ли сообщения отдела, служащего или все
27:        static final int DEPT_ONLY = 1;
28:        static final int EMP_ONLY = 2;
29:        static final int ALL = 3;
```

Листинг 18.45. Метод init() в PEMS

```
Метод init() в PEMS

33:    public void init(ServletConfig config) throws
                                              ServletException {
34:       super.init(config);
35:
36:    }
```

Регистрационное имя в сервлете

Методы `doGet()` и `doPost()` — те же самые (листинг 18.46). Они отправляют запросы к `processRequest()` (передать запрос).

Листинг 18.46. Метод doGet() в PEMS

```
Метод doGet() в PEMS

349:    protected void doGet(HttpServletRequest request,
                             HttpServletResponse response)
350:    throws ServletException, java.io.IOException {
351:       processRequest(request, response);
352:    }
```

Первоначально выходной формат сервлета объявлен в строке 245 (листинг 18.47) и указан как «text/html» (текст/HTML). Это означает, что данный конкретный сервлет возвратит результирующую страницу в формате, приемлемом для HTML. Затем `PEMS` проверяет, зарегистрирован ли текущий пользователь ранее в текущем сеансе (строка 248). Если пользователь не зарегистрирован, но ранее передал свое пользовательское имя и пароль (строки 307–309), то `PEMS` пробует выполнить его регистрацию (строки 311–322). Созданный объект `CAdmin` (класс «администратор» пакета `control`) размещается в сеансе (строка 312) вместе с признаком того, оказалась ли фиксация регистрационного имени успешной (строка 315). Затем поток программы переадресовывает пользователя к следующей стадии `processRequest()` (строка 316).

Любые неожиданные ошибки обрабатываются в строках 320–321, куда PEMS передает ошибку, и, следовательно, сообщение об ошибке может быть отображено на экране. Если регистрационное имя не передано в строке 314, то пользователь соответственно информируется в строках 324–326 совместно со строками 333–343. Строки 327–343 также используются как отправная точка для сервлета, когда пользователь сначала запрашивает *http-get* (получить HTTP). Это просто отображает окно регистрационного имени.

Листинг 18.47. Метод processRequest() в PEMS — регистрационное имя

Метод processRequest() в PEMS — регистрационное имя

```
243:    protected void processRequest(HttpServletRequest
                  request, HttpServletResponse response)
244:    throws ServletException, java.io.IOException {
245:       response.setContentType("text/html");
246:       java.io.PrintWriter out = response.getWriter();
247:
248:       Object logged = request.getSession().
                                    getAttribute("logged");
          ...[вырезано, обсуждается в следующем листинге]
306:       }else{ //пока не загружено, проверка, введено ли
                  пользовательское имя/пароль
307:          String username = (String) request.
                                 getParameter("username");
308:          String passwd = (String) request.
                                 getParameter("passwd");
309:          if(username != null && passwd != null){
310:             try{
311:                CAdmin admin = new CAdmin();
312:                request.getSession().
                          setAttribute("admin",admin);
313:                Object o = admin.login(username,passwd);
314:                if(CAdmin.LOGIN_OK.equals(o)){
315:                   request.getSession().setAttribute
                             ("logged", new Boolean(true));
316:                   processRequest(request,response);
317:                   return;
318:                }
319:             }catch(Exception exc){
320:                exc.printStackTrace(out);
321:              throw new ServletException(exc.getMessage());
322:             }
323:             //отображение сообщения об ошибке
                   регистрационного имени
```

```
324:            out.println("<html><head><title>EMS
                                  Login</title></head><body>");
325:            out.println("<h1><center>PSE's Email Management
                   System<br/>Servlet Version</center></h1>");
326:            out.println("<p><H2>Login ERROR...Please try
                                           again</H2>");
327:          }else{
328:             //отображение стандартного заголовка
329:            out.println("<html><head><title>EMS
                                 Login</title></head><body>");
330:            out.println("<h1><center>PSE's Email Management
                    System<br/>Servlet Version</center></h1>");
331:          }
332:
333:          out.println("<p><center>You need to login before
                               further processing</center><p>");
334:          out.println("<form method=post>");
335:          out.println("<center><table>");
336:          out.println("<tr><td>Username</td><td><input
                             name=username type=text maxlength=32
                                          size=16></td></tr>");
337:          out.println("<tr><td>Password</td><td><input
                           type=password name=passwd maxlength=32
                                        size = 16></td></tr>");
338:          out.println("<tr><td align=center>
               <input type = submit value=Submit></td>
               <td align=center> <input type=reset value=Clear>
               </td></tr>");
339:          out.println("</table></center></form>");
340:          out.println("</body></html>");
341:       }
342:       out.close();
343:    }
```

Изображение исходящих сообщений в сервлете

Если пользователь успешно зарегистрирован (строка 249 в листинге 18.48), далее выполняется проверка `PEMS`, требуется ли *send* (послать) исходящее сообщение (строка 264) или это был запрос *view message text* — просмотр текста сообщения (строка 266). В любом случае `PEMS` посылает или показывает содержимое исходящего сообщения перед продолжением обработки. Параметр `actionType` (тип действия) рассмотрен в листинге 18.49. Строки 253–257 определяют два различных цвета, которые можно использовать в таблице, чтобы отображать исходящие сообщения. Цвета необходимы, чтобы различать последовательные строки.

Листинг 18.48. Метод processRequest() в PEMS — просмотр сообщения

Метод processRequest() в PEMS — просмотр сообщения

```
243:    protected void processRequest(HttpServletRequest
                    request, HttpServletResponse response)
244:    throws ServletException, java.io.IOException {
           ...[вырезано — ссылка на листинг выше]
249:       if(logged != null){ //регистрация успешная
251:          out.println("<html><head><title>EMS
                                        Messages</title>");
253:          //определение раскраски строки для таблиц
254:          out.println("<style type=\"text/css\">");
255:          out.println("tr.green { background: #00cc00;
                                        color: #ccffcc }");
256:          out.println("tr.blue { background: #0000ff;
                                        color: #ffff00 }");
257:          out.println("</style>");
259:          out.println("</head><body>");
261:          out.println("<h1><center>PSE's Email Management
              System <br/>Servlet Version</center></h1><br>");
263:          //попытка послать эти выделенные сообщения, если
                пользователь укажет это
264:          if("Send".equals
                        (request.getParameter("actiontype")))
265:             sendMessages(request,out);
266:          else viewMessageTexts(request,out);
268:          String typeSelected =
                              request.getParameter("type");
269:          int type = EMP_ONLY;
270:          if(typeSelected != null){
271:             try{
272:                type = Integer.parseInt(typeSelected);
273:             }catch(Exception exc){
274:             }
275:          }
277:          //отображение списка сообщений
278:          if(type == EMP_ONLY){
279:             out.println("<center><font size=\"5\">
                                  <a href = EMS  ?type="+ALL+
280:                "">View all messages</a> | 
                                       <a href=EMS?type="+
281:                DEPT_ONLY+">View department messages
                                     </a> | "+
282:              "<a href=\"PEMSEdit\">Create new message</a>
                                     </font></center><br>");
```

```
283:
284:            out.println("<H3><center>Messages addressed to
                                    <b>you</b></center></H3>");
285:          }else if(type == DEPT_ONLY){
286:        out.println ( "<center><font size=\"5\"><a href=EMS
                                                    ?type="+ALL+
287:               ">View all messages</a> | 
                                               <a href=EMS?type="+
288:               EMP_ONLY+">View your messages
                                                </a> | "+
289:               "<a href=\"PEMSEdit\">Create new message
                                           </a> </font></center><br>");
291:            out.println("<H3><center>Messages addressed to
                        <b>your department</b></center></H3>");
292:          }else{
293:        out.println ("<center><font size=\"5\"><a href=EMS
                                             ?type="+DEPT_ONLY+
294:            ">View department messages</a> | "+
295:               "<a href=EMS?type="+EMP_ONLY+">View your
                                                 messages</a>"+
296:               " | <a href=\"PEMSEdit\">Create
                                                   new message
297:               </a>"+"</font></center><br>");
299:            out.println("<H3><center>All types of messages
                                               </center></H3>");
300:          }
302:          displayMessages(type,request,out);
304:          out.println("</body></html>");
305:          return;
```

Когда выясняется, что пользователь находится на странице *View Own Messages* — просмотр собственных сообщений (строка 278), ей/ему будет показан список гиперсвязей, помеченных как «Отображение всех сообщений», «Отображение сообщений отдела», «Создание нового сообщения» (строки 279–282). Это дает ему/ей гибкость перемещения от одного сценария использования к другому, как должно быть реализовано в итерации 2 EM. Подобные связи предлагаются для страницы *View department messages* (просмотр сообщений отдела) за исключением лишь того, что ярлыки связей будут следующие — «Отображение всех сообщений», «Отображение ваших сообщений» и «Создание нового сообщения». Наконец, страница *View all messages* (просмотр всех сообщений) будет иметь гиперсвязи, помеченные как «Отображение сообщений отдела», «Отображение ваших сообщений» и «Создание нового сообщения». После отображения заголовков выводится список исходящих сообщений (строка 302).

Листинг 18.49. Метод displayMessages() в PEMS

Метод displayMessages() в PEMS

```
45:     protected void displayMessages(int type,
           HttpServletRequest request,java.io.PrintWriter out){
47:        try{
48:           CAdmin admin=(CAdmin)request.getSession().
                                    getAttribute("admin");
49:           Collection msgs = new  LinkedList();
50:           int remainder = 0;
55:           if(type == EMP_ONLY)
56:           remainder=admin.getMsgSeeker().
                 retrieveMessagesForCurrentEmployee(msgs,
58:              IAConstants.MAX_MESSAGE, null);
60:           else if(type == DEPT_ONLY)
61:              remainder = admin.getMsgSeeker().
                    retrieveMessagesForDepartment(
62:                 admin.getEmployee().getDepartmentCode(),
63:                 msgs,IAConstants.MAX_MESSAGE, null);
66:           else
67:              remainder = admin.getMsgSeeker().
                                     retrieveAllMessages(
68:                 msgs, IAConstants.MAX_MESSAGE, null);
72:           if (msgs.isEmpty()){
73:              out.println("No unsent query messages.");
74:              return;
75:           }
77:           List params = Collections.list(request.
                                    getParameterNames());
78:           StringBuffer buf = new  StringBuffer();
79:           buf.append("<form method=post><br>\n");
80:           buf.append("<center><table border=2
                                       width=\"70%\">");
81:           buf.append("<tr><th>Selected</th>");
82:           buf.append("<th>Contact Name</th>");
83:           buf.append("<th>Company Name</th>");
84:           buf.append)"<th>Subject</th>");
85:           buf.append("<th>Created Date</th></tr>\n");
87:           int counter = 0;
88:           for (Iterator it = msgs.iterator();
                                    it.hasNext();) {
89:              IAOutMessage msg = (IAOutMessage) it.next();
90:              counter++;
93:              buf.append("<tr ").append(counter%2==0?
                           "class=green>":"class=blue>");
96:            buf.append("<td><input type=checkbox name=ck").
                                 append(msg.getMessageID());
```

```
99:              if(params.contains("ck"+msg.getMessageID()))
100:                buf.append(" checked");
101:             buf.append(">");
102:             buf.append("<td>");
104:             IAContact contact = msg.getContact();
105:             buf.append(contact.getFamilyName());
106:             buf.append(", ").append(contact.
                                          getFirstName());
108:             buf.append("</td><td>" ).append(contact.
                                         getOrganization());
109:             buf.append("</td><td>").append(msg.
                                             getSubject());
110:             buf.append("</td><td>").append(msg.
                                         getCreationDate());
111:             buf.append("</td></tr>\n");
112:          }
113:          buf.append("</table></center><br>");
114:          buf.append("<center><table><tr>");
115:          buf.append("<td><input type=submit
                          name=actiontype value=Send></td>");
116:          buf.append("<td><input type=submit
                          name=ctiontype value=View></td>");
117:          buf.append("<td><input
                          type=reset value=Clear></td>");
118:          buf.append("</tr></table></center>\n");
120:          buf.append("</form>");
121:          if(remainder >0)
122:             buf.append("<br>There are ").
                        append(remainder).append
                        (" messages still left in the server");
123:          out.println(buf.toString());
124:       }catch(Exception exc){
126:       }
127:    }
```

Три различных типа извлечения исходящих сообщений приведены в строках 55–67. Извлеченные исходящие сообщения отображаются в таблице с заголовком, определенным в строках 79–85, и содержимым, полученным в строках 88–112. Строка 93 переключает цвет строки с синего на зеленый и наоборот. Строка 99 проверяет, было ли выбрано конкретное исходящее сообщение на предыдущем экране, и если это так, то его следует показать как выбранное и на текущем экране. Атрибут `actiontype` задается в строках 115 и 116 как `Send` (послать) и `View` (просмотреть). Атрибут устанавливается в зависимости от нажатой пользователем кнопки. `ActionType` управляет следующей страницей, которую следует показать пользователю.

18.8.2. Класс PEMSEdit

Класс `PEMSEdit` (класс «редактор сервлета EM» пакета `presentation`) управляет просмотром и редактированием исходящего сообщения (рис. 18.19). Методы, поддерживающие жизненный цикл этого сервлета, подобны `PEMS` (класс «сервлет EM» пакета `presentation`). Методы `init()` (инициализировать), `destroy()` (удалить), и `getServletInfo()` (получить информацию о сервлете) просто вызывают методы своих суперклассов (или не обеспечивают никакой реализации). И `getContactOptions()` (получить параметры делового партнера), и `getEmployeeOptions()` (получить параметры служащего) используются, чтобы заполнить выпадающие списки в форме. `doGet()` (выполнить получение) и `doPost()` (выполнить передачу), действуют тем же самым образом, что и в `PEMS.java`. Метод `processRequest()` (передать запрос) форматирует HTML-результат и переадресует опции пользователя (или редактировать сообщение, или создать

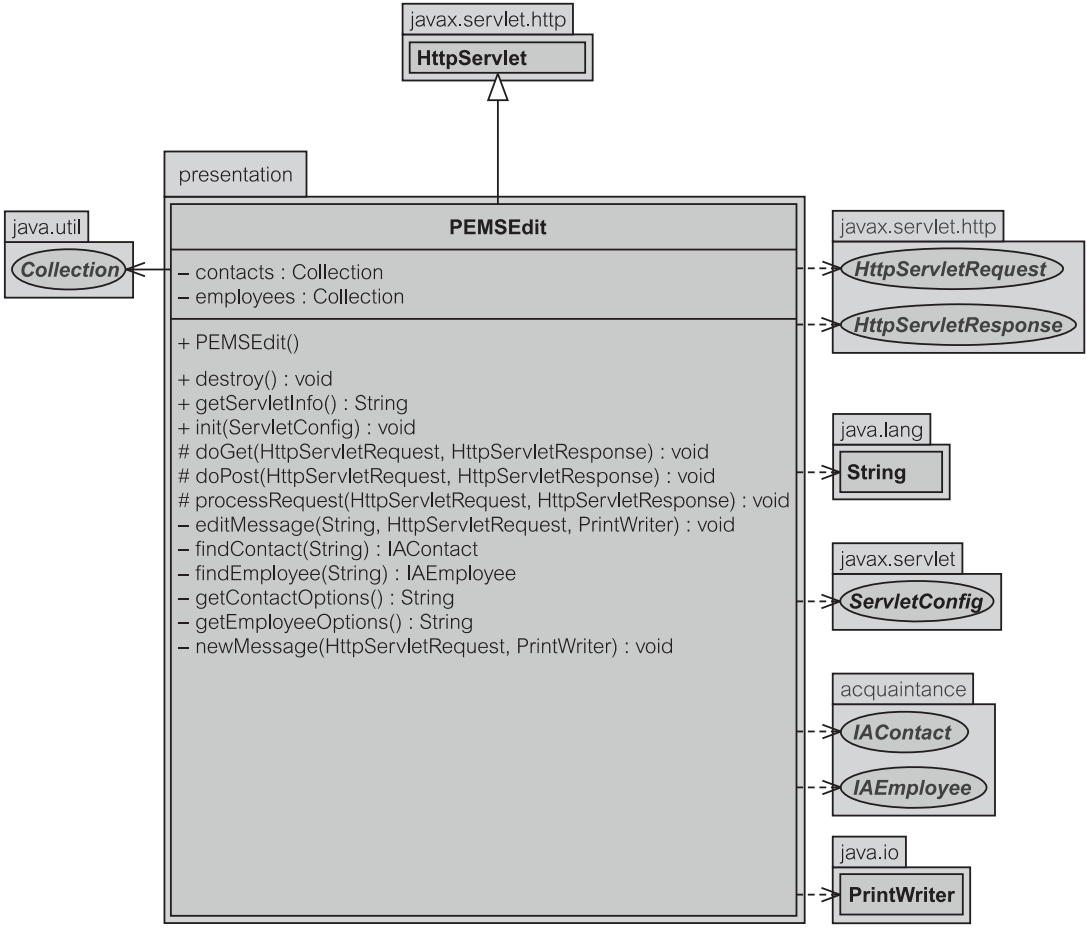


Рис. 18.19. Класс PEMSEdit в EM 2 (сервлет)

новое сообщение) к `editMessage()` (редактировать сообщение) и `newMessage()` (новое сообщение). `PEMSEdit` имеет две коллекции, чтобы обслуживать список служащих и список деловых партнеров (листинг 18.50). Это те же самые ссылки, что и принадлежащие классу `PMessageDetailWindow` в разделе 18.3.1. Те ссылки требовались, чтобы заполнить выпадающий список для создания нового исходящего сообщения.

Листинг 18.50. Поля в PEMSEdit.java (сервлет)

Поля в PEMSEdit.java

```
23:     private Collection employees = null;
24:     private Collection contacts = null;
```

Листинг 18.51. Метод newMessage() в PEMSEdit

Метод newMessage() в PEMSEdit

```
166:  private void newMessage(HttpServletRequest request,
                                    java.io.PrintWriter out)
167:  throws ServletException, java.io.IOException {
168:   CAdmin admin=(CAdmin)request.getSession().
                                      getAttribute("admin");
169:   if(contacts == null) contacts = admin.listContacts();
170:   if(employees == null) employees = admin.listEmployees();
172:   DateFormat df = DateFormat.getDateInstance
                                         (DateFormat.SHORT);
173:   StringBuffer buf = new StringBuffer();
174:   buf.append("<form method=post>\n");
175:   buf.append("<table border=0 width=\"70%\">\n");
177:  buf.append("<tr><td width=\"30%\">Subject</td><td>");
178:  buf append("<input type=text name=subject></td></tr>\n");
180:  buf append("<tr><td width=\"30%\">Creator</td><td>");
181:  buf.append(admin.getEmployee().getFamilyName()).
                                              append(",");
182:  buf.append(admin.getEmployee().getFirstName()).
                                       append("</td></tr>\n");
184:  buf.append("<tr><td width=\"30%\">Creation Date
                                              </td><td>");
185:  buf.append(df.format(new java.util.Date())).
                                    append("</td></tr>\n");
187:  buf.append("<tr><td width=\"30%\">Addressed
                                             To</td><td>");
188:  buf.append(getContactOptions()).append("</td></tr>\n");
           ...[вырезано для сохранения места]
```

Создание нового исходящего сообщения (листинг 18.51) достигается в итерации 2 EM с помощью объекта `PMessageDetailWindow` — класс «окно деталей сообщения» пакета `presentation` (метод `showNewMessage()` — показать новое сообщение), рассмотренного в разделе 18.3.1, и объекта `CAdmin` — класс «администратор» пакета `control` (метод `newMessage()` — новое сообщение), рассмотренного в разделе 18.3.2. Объект `CAdmin`, созданный ранее, получается из сеанса (строка 168), чтобы извлечь необходимые данные (строки 169–170). Данные отображаются в строках 173–188 в форме выпадающих списков, находящихся в строке 188.

Резюме

1. Код Java для итерации 2 строго соответствует PCMEF-структурному шаблону.
2. Чтобы подчеркнуть ясность и выдвинуть на первый план простоту PCMEF-шаблона, итерация 2 ограничивает число классов в каждом пакете.
3. Итерация 2 имеет три различных версии: 1) приложение, 2) апплет и 3) версия сервлета.
4. Большинство изменений между итерациями 1 и 2 относится к уровню `presentation` (представление). Другие уровни изменены единственно для того, чтобы обслужить дополнительные функциональные возможности, требуемые спецификациями итерации 2.
5. Итерация 2 реализует толстый апплет с небольшой модификацией класса `PWindow` (класс «окно» пакета `presentation`).
6. Пакет `acquaintance` (знакомство) все еще состоит из четырех интерфейсов, как определено в итерации 1: `IAConstants` (интерфейс «константы» пакета `acquaintance`), `IAContact` (интерфейс «деловой партнер» пакета `acquaintance`), `IAEmployee` (интерфейс «служащий» пакета `acquaintance`) и `IAOutMessage` (интерфейс «исходящее сообщение» пакета `acquaintance`). Они дополнены новыми функциональными возможностями, чтобы поддержать идею относительно *измененных* объектов.
7. Пакет `presentation` (представление) содержит шесть классов: `PWindow`, `PMessageDetailWindow` (класс «окно деталей сообщения» пакета `presentation`), `PSendPreview` (класс «предварительный просмотр посылаемого сообщения» пакета `presentation`), `PMessageTableModel` (класс «модель таблицы для сообщений» пакета `presentation`), `PDisplayList` (класс «список отображения» пакета `presentation`) и `PDisplayData` (класс «отображаемые данные» пакета `presentation`).
8. Пакет `control` (управление) содержит три класса: `CAdmin` (класс «администратор» пакета `control`), `CMsgSeeker` (класс «поиск сообщения» пакета `control`) и `CMsgSender` (класс «передача сообщения» пакета `control`).
9. Четыре класса и интерфейс представляют пакет `entity` (сущность). Классы — это `EContact` (класс «деловой партнер» пакета `entity`),

`EEmployee` (класс «служащий» пакета `entity`), `EOutMessage` (класс «исходящее сообщение» пакета `entity`) и `EIdentityMap` (класс «коллекция идентичности объектов» пакета `entity`). Интерфейс — это `IEObjectID` (интерфейс «идентификатор объекта» пакета `presentation`).

10. Пакет `mediator` (посредник) состоит из двух классов: `MDataMapper` (класс «преобразователь данных» пакета `mediator`) и `MModerator` (класс «координатор» пакета `mediator`).
11. Пакет `foundation` (основание) состоит из трех классов: `FConnection` (класс «соединение» пакета `foundation`), `FReader` (класс «чтение» пакета `foundation`) и `FWriter` (класс «запись» пакета `foundation`). Этот пакет фактически не изменен по сравнению с итерацией 1.
12. Версия сервлета формирует HTML-результат. Он (сервлет) управляется двумя классами: `PEMS` (класс «сервлет EM» пакета `presentation`) и `PEMSEdit` (класс «редактор сервлета EM» пакета `presentation`).

Ключевые термины

Итерация 2. Вопросы и упражнения

Ниже приведен небольшой набор вопросов и упражнений, относящихся к коду итерации 2 EM. Большее количество вопросов и упражнений можно найти на Web-сайте книги.

1. Какие бы потребовались необходимые изменения, чтобы преобразовать версию толстого апплета итерации 2 EM (раздел 18.7) в версию тонкого апплета?
2. Обратимся к разделу 18.8.1 и листингам 18.46 и 18.47. Почему методы `doPost()` (выполнить передачу) и `doGet()` (выполнить получение) направляют запросы к `processRequest()` (передать запрос) вместо самостоятельного выполнения работы?
3. Покажите и объясните фрагмент кода итерации 2, который отвечает за отображение рис. 16.21 (раздел 16.5). Рисунок воспроизведен ниже как рис. 18.20 для простоты ссылки.
4. Покажите и объясните фрагмент кода итерации 2, который отвечает за удаление исходящего сообщения, когда нажимается кнопка `Delete` (удалить) на рис. 16.20 (раздел 16.5). Рисунок воспроизведен ниже как рис. 18.21 для простоты ссылки.
5. Итерация 2 не позволяет иметь существующие исходящие сообщения, которые нужно скорректировать до того, как они будут переданы по электронной почте. Сделайте возможную реализацию для таких корректировок.

Рис. 18.20. Ошибка «Невозможно послать»

6. Предположим, что итерация 2 требует двойного нажатия мышью таблицы исходящих сообщений, чтобы просмотреть отобранное исходящее сообщение для его корректировки. Как это можно реализовать?
7. Обратимся к разделу 18.3.3 и следующему варианту метода `filter()` (фильтр) — листинг 18.52. Объясните причины проверки в строках 68–70.

Листинг 18.52. Метод filter() в PMessageTableModel — только дата

Метод filter() в PMessageTableModel — только дата

```
67:     public void filter(Object datefilter){
68:        if(datefilter instanceof String){
69:           filter(datefilter.toString());
70:           return;
71:        }
72:        if(ALL_DAYS_FILTER.equals(datefilter)){
73:           data.setFilterDate(3);
74:        }else if(LAST_DAYS_FILTER.equals(datefilter)){
75:           data.setFilterDate(0);
76:        }else if(TODAY_FILTER.equals(datefilter)){
77:           data.setFilterDate(1);
```

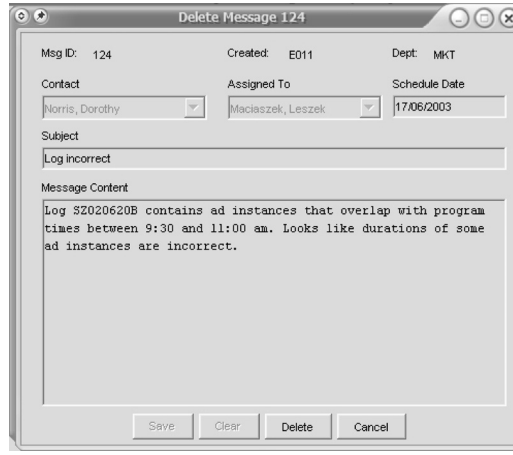


Рис. 18.21. Окно для удаления исходящего сообщения

```
78:         } else if(FUTURE_DAYS_FILTER.equals(datefilter)){
79:            data.setFilterDate(2);
80:         }
81:         fireTableDataChanged();
82:      }
```

8. Объясните цель следующего фрагмента кода (листинг 18.53).

Листинг 18.53. Метод getRawData() в PMessageTableModel

Метод getRawData() в PMessageTableModel

```
145:     public Object getRawObject(int row){
146:        return data.getOutMessage(row);
147:     }
```

9. Следующее — возможная реализация `SortMouseListener` (приемник сортировки с помощью мыши) в итерации 2 ЕМ (листинг 18.54). Объясните, что происходит в этом коде.

Листинг 18.54. Внутренний класс SortMouseListener в PMessageTableModel

Внутренний класс SortMouseListener в PmessageTableModel

```
217:     private class SortMouseListener extends
            java.awt.event.MouseAdapter{
218:           JTable tableView;
219:           PMessageTableModel model;
220:           public SortMouseListener(
               JTable table,PMessageTableModel model){
221:              this.tableView = table;
222:              this.model = model;
223:           }
224:          public void mouseClicked(java.awt.event.
                                        MouseEvent e) {
225:              TableColumnModel columnModel = tableView.
                                           getColumnModel();
228:              int viewColumn = columnModel.
                                  getColumnIndexAtX(e.getX());
229:              int column=tableView.
                        convertColumnIndexToModel(viewColumn);
230:              model.sort(column);
231:           }
232:     }
```

Часть

4

Разработка данных и бизнес-компоненты

Предыдущие три части книги и предыдущие две итерации учебного примера охватывали весь типичный материал, который обычно имеется в учебнике по программной инженерии. Однако обещание этой книги — идти далее простых проектов программной инженерии и представить принципы и практику разработки *больших систем предприятия*. Эти принципы и практика должны распространяться на главный ресурс в любом предприятии — *информационный ресурс*. Важная часть мудрости, приписываемой Бобу Эпштейну из фирмы Sybase, гласит: «Приложения приходят и уходят; данные остаются навсегда». Часть 4 этой книги делает должный акцент на источниках данных.

Не следует считать, что *источники данных* игнорировались на предшествующих страницах книги. Глава 10 представила все основные проблемы проектирования и программной инженерии БД. Многие разделы других глав были явно посвящены роли и месту источников данных в создании ПО. Существенные аспекты структурного рассмотрения ПО, обсужденные в главах 9 и 15, связаны с организацией и манипуляцией источниками данных в программах.

Эта часть книги продолжается, укрепляется и основывается на знании источников данных, рассмотренных ранее. Последующее обсуждение охватывает главные концепции и проблемы инженерии данных. *Инженерия данных* — отдельная область знаний со своими собственными журналами, конференциями, техническими комитетами и т. д. Инженерия данных занимается проектированием, хранением, управлением и использованием данных, информации и знаний в системах ПО. Ее главный акцент — базы данных (БД). Современная инженерия данных охватывает многие проблемы прикладных разработок и касается таких тем, как хранение данных, получение данных с помощью Web-технологии, мультимедийные данные, метаданные и XML.

Бизнес-компоненты занимают твердое место между источниками данных и прикладной логикой. Они являются компонентами многократного использования, которые входят в различные изображения и формы, наиболее популярными из которых являются компоненты Enterprise JavaBeans. Они могут использоваться, чтобы помочь в программировании, и фактически создают прикладной код. Компоненты могут быть размещены на сервере приложения или сервере БД. Бизнес-компоненты должным образом рассматриваются в этой последней части книги.

Главные цели изучения части 4 и итерации 3 учебного примера включают получение следующих знаний:

- принципы обеспечения безопасности и целостности систем управления БД;
- программирование приложений для авторизованного доступа к данным БД и программам БД;
- программирование элементов управления безопасностью и целостностью в БД;
- принципы управления транзакциями в реляционных и объектно-ориентированных БД;
- программирование параллелизма в бизнес-операциях, выполняемых клиентами приложения;

- сервисы транзакций, поддерживаемые уровнями Web-технологии, приложения и БД;
- взаимодействие пользовательских интерфейсов и бизнес-компонентов, а также бизнес-компонентов в коде приложения и постоянных данных в БД;
- шаблоны генерируемых классов Java и XML-файлы для управления БД и интеграции с приложениями;
- реализация и развертывание бизнес-компонентов на сервере приложения.

Глава

19 Требования к итерации 3 и объектная модель

Итерация 3 учебного примера EM (Email Management — управление электронной почтой) сосредоточена на усовершенствованиях БД и на рефакторингах в более низких PCMEF-уровнях кода приложения, полученного в итерации 2. Насколько возможно, **бизнес-логика** приложения перемещена от клиента к серверу. Это, как ожидается, даст существенную мобильность, удобство сопровождения, масштабируемость, безопасность, качество выполнения функций и связанные с ними выгоды. Принимая во внимание, что предыдущие итерации были сконцентрированы на обеспечении новых функциональных возможностей и (в итерации 2) на лучшей применимости, акцент итерации 3 делается на оставшихся особенностях аббревиатуры FURPS+ — надежность, выполнение функций и возможность сопровождения (раздел 8.4).

Как и в предыдущих итерациях, шаг итерации 3 разработан так, чтобы быть реализованным в пределах двух недель небольшим коллективом разработчиков. Усилия программирования объединяют использование Java на клиенте и программы, размещенные в реляционной БД (хранимые процедуры, хранимые функции, триггеры). Проблема заключается в удалении из Java-приложения кода, который должен быть централизован в БД для целей FURPS+. Эта задача требует дальнейшего рефакторинга Java-кода, в особенности на уровне `foundation` (основание), и воздействия также на уровень `domain` (предметная область).

19.1. Модель сценариев использования

Рис. 19.1 представляет модель сценариев использования для итерации 3. Граница проекта и акторы, которые определяют эту границу, являются неизменными по сравнению с предыдущими итерациями. Поэтому акторы в модели не показаны. Диаграмма представляет сценарии использования и главные отношения между ними. Итерация 3 новых функциональных возможностей охвачена прямоугольниками. Имеются только четыре новых сценария использования: `Apply Authorization Rules` (применение правил авторизации), `Maintain Authorization Rules` (обслуживание правил авторизации), `Produce Daily Report` (формирование ежедневного отчета) и `Modify Message Before Emailing` (корректировка сообщения перед отправкой по электронной почте). Последний является расширением сценариев использования `Email Message` (сообщение электронной почты) и `Display Message Text` (отображение текста сообщения).

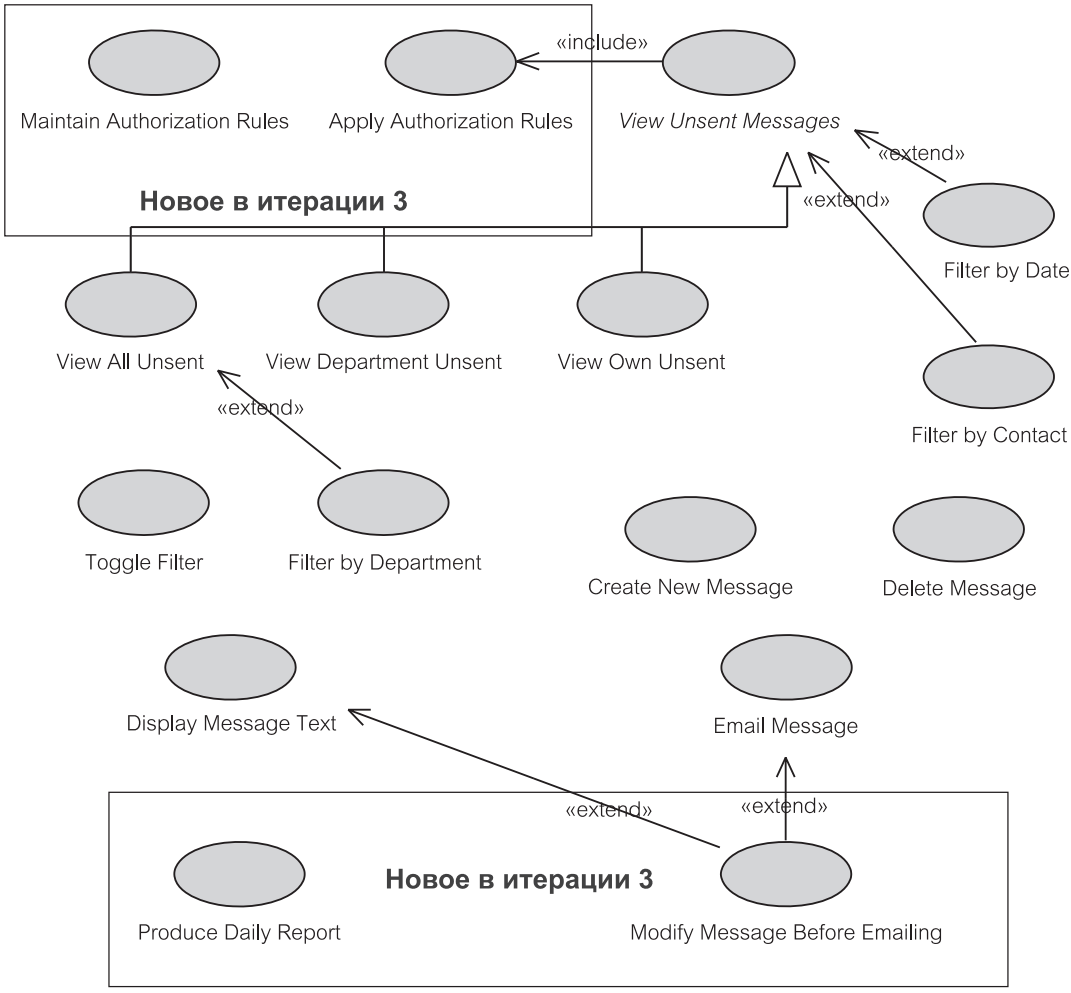


Рис. 19.1. Диаграмма сценариев использования для итерации 3

`Apply Authorization Rules` включен в `View Unsent Messages` (просмотр непосланных сообщений). Это означает, что отображение непосланных исходящих сообщений ограничено правилами авторизации, которые могут применяться к текущему пользователю. Правила зарегистрированы в БД. БД гарантирует, что приложение клиента может отображать только «авторизованные» исходящие сообщения.

`Maintain Authorization Rules` на самом деле не является кодом приложения итерации 3 учебного примера EM. Системный администратор, а не пользователь приложения поддерживает правила авторизации. Этот сценарий использования на самом деле является утилитой, предназначенной администратору системы. По этой причине она разрабатывается как отдельный бизнес-компонент независимо от кода приложения.

`Produce Daily Report` введен, чтобы формировать ежедневные итоговые отчеты относительно операций по электронной почте для всех деловых партнеров. Деловые партнеры, которым никакие сообщения по электронной почте в данный день не были посланы, не вносятся в список отчета за этот день. Отчеты отображаются на экране, прежде чем они будут напечатаны.

`Modify Message Before Emailing` обеспечивает пользователю возможность корректировать предмет исходящего сообщения или его текст перед передачей по электронной почте. Корректировки хранятся в БД. Изменять исходящее сообщение перед передачей по электронной почте может только служащий, который создал его, или контролер (менеджер) этого служащего.

19.2. Документ сценария использования

Область действия документа сценария использования для итерации 3 — объединение требований двух предыдущих итераций и новых требований итерации 3. Иными словами, область действия соответствует модели сценариев использования на рис. 19.1. Родительским сценарием использования, который охватывает все итерации, является `Manage Email` (управление электронной почтой) — рис. 8.1.

Как и требуется при итеративной и пошаговой разработке, итерация 3 обеспечивает новый шаг на вершине итерации 2. Документ сценария использования описывает не только функциональные расширения, но также и указывает, как должны быть реализованы другие особенности FURPS+.

19.2.1. Краткое описание, предусловия и постусловия

Краткое описание

Итерация 3 сценария использования `Manage Email` (управление электронной почтой) позволяет служащему создавать, удалять, отображать, изменять и отправлять по электронной почте исходящие сообщения деловым партнерам. Она также позволяет формировать основные отчеты работы с электронной почтой. Одновременно может быть передано по электронной почте только одно исходящее сообщение.

Сценарий использования может отображать, изменять и передавать по электронной почте исходящие сообщения, предварительно размещенные в БД другими процессами. Он также позволяет создать новое исходящее сообщение, а также отобразить его, удалить, скорректировать или передать по электронной почте, когда это требуется. Отображение исходящих сообщений может быть ограничено множеством условий показа (критериями поиска). Текущее отображение может быть далее отфильтровано на основе ряда условий фильтрации. Только авторизованные пользователи могут просматривать конкретные исходящие сообщения и выполнять конкретные операции.

Предусловия

1. Пользователь (актор) — это служащий (Employee), который работает в отделе работы с клиентами (Customer Department) или иным образом упол-

номочен системным администратором (System Administrator) иметь доступ к ЕМ-приложению.

2. БД ЕМ содержит исходящие сообщения, которые можно передать по электронной почте деловым партнерам. Создание новых исходящих сообщений не является никакой заменой для размещения исходящих сообщений в БД другими процессами АЕМ-системы (глава 6).
3. Служащий связан с почтовым сервером и является авторизованным пользователем БД.

Постусловия

1. Программа обновила БД ЕМ, чтобы зафиксировать каждую успешную передачу по электронной почте исходящих сообщений.
2. БД ЕМ остается в неповрежденном состоянии, если произойдет любое исключение или ошибка.
3. У служащего, оставляющего приложение, закрываются все соединения БД и весь рабочий стол приложения или допустимые GUI-окна Web-технологии.

19.2.2. Основной поток

Совместимое с подходом, принятым в предыдущих итерациях, описание потока событий включает прототипы GUI-окон. Итерация 3 сохраняет GUI-представление итерации 2 и расширяет его, чтобы приспособить к новым требованиям итерации 3.

Основной поток

Сценарий использования начинается, когда служащий хочет создать, просмотреть и/или передать по электронной почте исходящие сообщения деловым партнерам.

Система отображает информационное сообщение и запрашивает у служащего пользовательское имя и пароль (рис. 19.2). Окно регистрационного имени по сравнению с итерацией 2 не изменилось.

1. Система пытается соединить служащего с БД ЕМ.
2. После успешного соединения приложение отображает главное окно приложения, через которое пользователь может взаимодействовать с системой (рис. 19.3). Главное окно приложения не изменилось по сравнению с итерацией 2.

Рис. 19.2. Прототип окна регистрационного имени

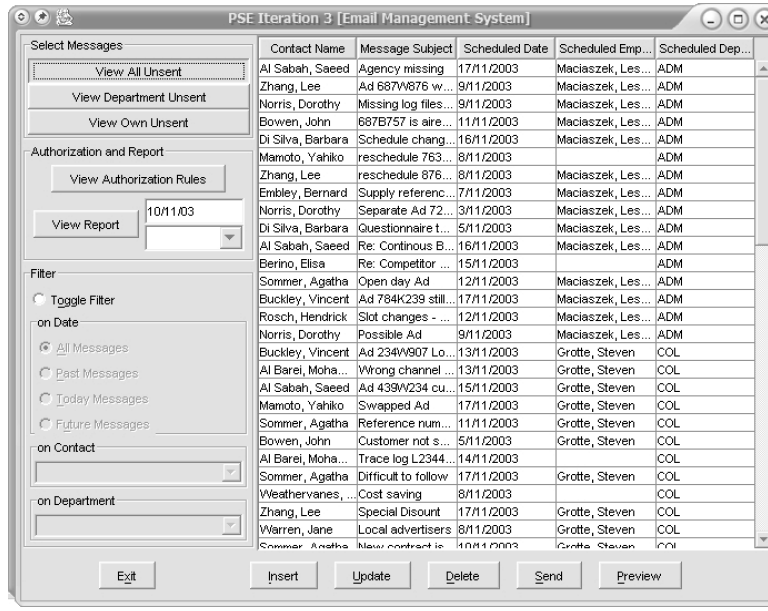


Рис. 19.3. Прототип главного окна приложения

3. Пользователь приложения может выполнять следующие задачи:
   а) просмотр непосланных исходящих сообщений деловым партнерам (см. ниже «S1 — View Unsent Messages» — просмотр непосланных сообщений);
   б) фильтрацию отображения исходящих сообщений по различным критериям (см. ниже «S2 — Filter Messages» — фильтрация сообщений);
   в) отображение текста выбранного исходящего сообщения (см. ниже «S3 — Display Message Text» — отображение текста сообщения);
   г) передачу по электронной почте выбранного исходящего сообщения (см. ниже «S4 — Email Message» — сообщение электронной почты);
   д) создание нового исходящего сообщения (см. ниже «S5 — Create New Message» — создание нового сообщения);
   е) удаление исходящего сообщения (см. ниже «S6 — Delete Message» — удаление сообщения);
   ж) корректировку исходящего сообщения (см. ниже «S7 — Modify Message Before Emailing» — корректировка сообщения перед передачей по электронной почте);
   з) создание ежедневного отчета о деятельности (см. ниже «S8 — Produce Daily Report» — создание ежедневного отчета);
   и) завершение приложения, при котором сценарий использования заканчивается.
4. Системный администратор может дополнительно выполнять задачу поддержания правил авторизации (см. ниже «S9 — Maintain Authorization Rules» — поддержка правил авторизации).

19.2.3. Подпотоки

S1 — View Unsent Messages (просмотр непосланных сообщений)

Этот подпоток состоит из четырех опций:

- S1.1 — Apply Authorization Rules (применение правил авторизации);
- S1.2 — View All Unsent (просмотр всех непосланных сообщений);
- S1.3 — View Department Unsent (просмотр непосланных отделом сообщений);
- S1.4 — View Own Unsent (просмотр собственных непосланных сообщений).

S1.1 — Apply Authorization Rules (применение правил авторизации)

Каждая из трех опций, *определенных во* `View Unsent Messages`, управляется строгими **правами доступа**, показанными как отношение «`include`» (включить) на рис. 19.1. Права доступа зависят от отдела служащего. В настоящее время служащие делятся на пять групп, определенных как акторы AEM-системы в модели сценариев использования на рис. 6.3. Группы авторизации следующие:

- COL — Data Collection Employee — служащий по сбору данных (то есть служащий, работающий для отдела сбора данных);
- QUA — Quality Control Employee — служащий контроля качества (то есть служащий, работающий в отделе контроля качества);
- CUS — Customer Services Employee — служащий по обслуживанию клиентов;
- ADM — System Administrator — системный администратор (то есть служащий, ответственный за подсистему EM);
- IND — Independent Employee — независимый служащий (то есть служащий, не принадлежащий какому-либо отделу, или внешний консультант, типа Rates Consultant — консультант по расценкам — на рис. 6.3).

Следующая матрица (таблица 19.1) управляет **правами** доступа для просмотра исходящих сообщений. Служащий, работающий в отделе, соответствующем строке матрицы, может рассматривать исходящие сообщения отделов, перечисленных в списке столбцов, если в соответствующей ячейке находится «Yes» (да).

Таблица 19.1. Матрица авторизации для просмотра исходящих сообщений

	COL	QUA	CUS	ADM	IND
Data Collection (COL)	YES	NO	NO	NO	NO
Quality Control (QUA)	YES	YES	NO	NO	NO
Customer Services (CUS)	YES	NO	YES	NO	YES
System Administrator (ADM)	YES	YES	YES	YES	YES
Independent Employee (IND)	NO	NO	NO	NO	YES

Обратите внимание, что добавление Independent Employee в таблице авторизации означает, что IND становится законным кодом для отдела. Практически же IND-отдел не существует. Это только неофициальная группировка независимых служащих. Такой «виртуальный» IND-отдел необходим, чтобы облегчить управление дополнительными бизнес-правилами, которые БД должна предписать с точки зрения разрешения (раздел 19.5).

S1.2 — View All Unsent (просмотр всех непосланных сообщений)

Когда пользователь выбирает опцию `View All Unsent`, приложение представляет список непосланных исходящих сообщений, которые доступны пользователю для просмотра.

На основе матрицы авторизации в таблице 19.1 система представляет список всех непосланных исходящих сообщений в главном окне приложения (рис. 19.3). Далее приведены свойства списка:

- Список содержит именованные столбцы: `Contact Name` (имя делового партнера), `Message Subject` (тема сообщения), `Scheduled Date` (намеченная для отправки дата), `Scheduled Employee` (намеченный служащий) и `Scheduled Department` (намеченный отдел).
  - `Contact Name` отображает фамилию и имя делового партнера.
  - `Scheduled Employee` отображает фамилию и имя служащего или ничего (если намеченный служащий неизвестен); `Scheduled Department` отображает название отдела или ничего (если намеченный отдел неизвестен).
- Список сортируется в порядке возрастания `Contact Name` и `Scheduled Date`.
- Список может прокручиваться, чтобы учесть ситуации, когда число извлеченных строк превышает размер окна.
- Исходящие сообщения в списке можно выбрать мышью. Должны поддерживаться обычные функциональные возможности: `Ctrl-Click` (нажатие левой клавиши мыши при нажатой клавише `Ctrl` клавиатуры), чтобы выбрать несколько несмежных исходящих сообщений, и `Shift-Click` (нажатие левой клавиши мыши при нажатой клавише `Shift` клавиатуры), чтобы выбрать диапазон смежных исходящих сообщений, хотя эти функциональные возможности не используются в итерации 3.

S1.3 — View Department Unsent (просмотр непосланных отделом сообщений)

Когда пользователь выбирает опцию `View Department Unsent`, приложение представляет список исходящих сообщений, намеченных для передачи по электронной почте служащими отдела, в котором работает текущий служащий (то есть, текущий пользователь). Эта опция не относится к независимым служащим, которые могут рассматривать только свои собственные исходящие сообщения (см. подпоток S1.4).

GUI-представление использует то же самое окно, что и на рис. 19.3. В окне имеется четкий признак того, что из БД извлечены и могут просматриваться только исходящие сообщения, намеченные для конкретного отдела.

S1.4 — View Own Unsent (просмотр собственных непосланных сообщений)

Когда пользователь выбирает опцию `View Own Unsent`, приложение представляет список исходящих сообщений, намеченных для передачи по электронной почте текущим служащим (то есть, текущим пользователем). GUI-представление использует то же самое окно, что и на рис. 19.3.

В окне имеется четкий признак того, что из БД извлечены и могут просматриваться только исходящие сообщения, намеченные для зарегистрированного служащего.

S2 — Filter Messages (фильтрация сообщений)

Этот подпоток состоит из четырех опций:

- S2.1 — Toggle Filter (переключатель фильтра);
- S2.2 — Filter by Date (фильтр по дате);
- S2.3 — Filter by Contact (фильтр по деловому партнеру);
- S2.4 — Filter by Department (фильтр по отделу).

S2.1 — Toggle Filter (переключатель фильтра)

Панель фильтров обеспечивает набор элементов управления для ограничения данных, выделенных для показа после того, как они были извлечены из БД подпотоком `View Unsent Messages` (просмотр непосланных сообщений). Панель не активна, пока не отображен список исходящих сообщений с помощью `View Unsent Messages` и не нажата кнопка `Toggle Filter`.

По умолчанию опция `Toggle Filter` выключена. Пользователь может включить ее, чтобы использовать другие опции фильтра: `Filter by Date`, `Filter by Contact` и/или `Filter by Department`.

Фильтр изображается активным, когда список отображенных исходящих сообщений ограничен условиями, заданными опциями `Filter by Date`, `Filter by Contact` и/или `Filter by Department`. Когда фильтр установлен, использование опции `Toggle Filter` может выключить его. Это приводит к повторному отображению списка исходящих сообщений, как требуется подпотоком `View Unsent Messages` до того, как был применен какой-либо фильтр.

S2.2 — Filter by Date (фильтр по дате)

Панель `Filter by Date` обеспечивает набор переключающих кнопок, чтобы отфильтровать исходящие сообщения с намеченной датой (то есть, намеченные для передачи по электронной почте) в прошлом, будущем, сегодняшней датой или все (с любой датой). В любое время может быть активна только одна из этих кнопок. По умолчанию, когда активизируется панель, активной становится кнопка `All` (все).

S2.3 — Filter by Contact (фильтр по деловому партнеру)

Панель фильтров реализуется комбинированной строкой ввода (полем со списком), которая позволяет выбрать делового партнера из выпадающего списка имен деловых партнеров или которая может быть оставлена незаполненной (рис. 14.3). Фильтр автоматически реализуется, когда отбирается деловой партнер. Это воздействует на список исходящих сообщений, отобража-

емых, чтобы показать лишь исходящие сообщения, предназначенные для выбранного делового партнера, исключая все исходящие сообщения, отфильтрованные текущим заданным фильтром даты.

S2.4 — Filter by Department (фильтр по отделу)

Здесь также используется комбинированная строка ввода, чтобы позволить пользователю отфильтровать список исходящих сообщений отдела, который выбирается из выпадающего списка названий отделов (рис. 14.3). Выводится только список исходящих сообщений, предназначенных конкретному отделу (который выбран с помощью комбинированной строки ввода). Пустое значение комбинированной строки ввода означает, что фильтрация по отделу не выполняется. Этот фильтр автоматически выполняется, когда отобрано название отдела. Фильтр изменяет список отображаемых исходящих сообщений, чтобы показать только те сообщения, которые назначены конкретному отделу или, если быть точными, исходящие сообщения, которые назначены служащим выбранного отдела. Список исключает любые исходящие сообщения, отфильтрованные по заданной дате и по деловому партнеру.

S3 — Display Message Text (отображение текста сообщения)

Исходящее сообщение, выбранное (подсвеченное) в окне списка исходящих сообщений, может быть показано полностью в отдельном диалоговом окне (рис. 19.4[1]). Это окно представляет полный текст исходящего сообщения вместе с другой описательной информацией. Панель с текстом можно прокручивать. Окно содержит три кнопки:

- `Email this Message` (отправить по электронной почте это сообщение) — см. подпоток S4;

Рис. 19.4. Диалоговое окно, показывающее текст текущего выбранного сообщения

[1] К сожалению, на рисунке имеется ошибка — не показана кнопка Modify this Message (модифицировать это сообщение), о которой упоминается в тексте. (*Прим. перев.*)

Рис. 19.5. Информационное сообщение после передачи по электронной почте

- `Modify this Message` (скорректировать это сообщение) — см. подпоток S7;
- `Return to List of Messages` (вернуться к списку сообщений) — чтобы закрыть это окно и возвратиться к главному окну приложения.

S4 — Email Message (передача сообщения по электронной почте)

Диалоговое окно на рис. 19.4, которое позволяет передать по электронной почте исходящее сообщение, может быть создано подпотоком S3 или данным подпотоком. В подпотоке S3 двойное нажатие клавишей мыши на выбранном исходящем сообщении создает диалоговое окно. В подпотоке S4 то же самое получается нажатием кнопки `Email Selected Message` (передать по электронной почте выбранное сообщение) в основном окне приложения (рис. 14.3).

Успешная передача по электронной почте завершается обновлением БД и информационным сообщением, отображаемым пользователю (рис. 19.5).

S5 — Create New Message (создание нового сообщения)

Итерация 2 обеспечила опцию для создания нового исходящего сообщения и сохранения его в БД. Визуально эта опция в итерации 3 не изменилась, но она включает несколько большую работу на заднем плане, чтобы обеспечить права доступа, представленные в таблице 19.1. Служащий, который не может

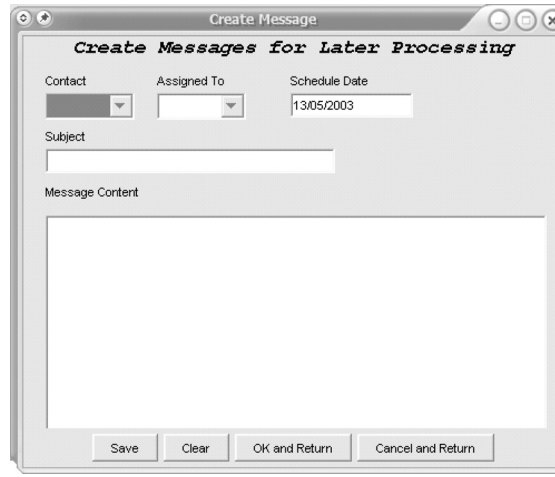


Рис. 19.6. Диалоговое окно для создания новых исходящих сообщений

Рис. 19.7. Подтверждение удаления сообщения

просматривать исходящие сообщения, намеченные другим служащим или отделам, аналогично не может создавать исходящие сообщения для этих служащих и отделов.

Нажатие кнопки `Create New Message` (создание нового сообщения) в основном окне приложения создает диалоговое окно для задания нового исходящего сообщения, как показано на рис. 19.6.

Диалоговое окно позволяет выбрать делового партнера и наметить служащего и/или отдел, которые будут ответственны за передачу. Тема и содержание исходящего сообщения могут быть напечатаны в соответствующих полях. Окно содержит четыре командные кнопки:

- `Save` (сохранить) — сохранить новое исходящее сообщение в БД, но оставить диалоговое окно открытым, чтобы иметь возможность создать следующее исходящее сообщение;
- `OK and Return` (все в порядке и вернуться) — сохранить новое исходящее сообщение в БД, а затем закрыть это окно и вернуться в главное окно приложения;
- `Clear` (очистить) — очистить текущее содержимое диалогового окна и разрешить пользователю заново создать новое исходящее сообщение;
- `Cancel and Return` (отменить и вернуться) — отменить операцию, закрыть это окно и вернуться к главному окну приложения.

S6 — Delete Message (удаление сообщения)

Когда исходящие сообщения больше не нужны, возможно, из-за исправлений, сделанных другими служащими, они могут быть удалены из БД. Нажатие кнопки `Delete Message`, когда исходящее сообщение выбрано в главном окне, активизирует этот подпоток. В этом случае отображается простое окно, чтобы запросить пользователя подтвердить удаление, как показано на рис. 19.7. Если пользователь решит отменить удаление, никаких изменений не произойдет.

S7 — Modify Message Before Emailing (корректировка сообщения перед передачей по электронной почте)

Данная опция расширяет подпотоки S3 и S4. Эти два подпотока используют одно и то же диалоговое окно (рис. 19.4). Подпоток S7 активизируется, когда пользователь нажимает кнопку `Modify this Message`. Нажатие этой кнопки приводит к диалоговому окну (рис. 19.8), которое является по существу тем же самым, что и диалоговое окно для подпотока S5 (рис. 19.6), за ис-

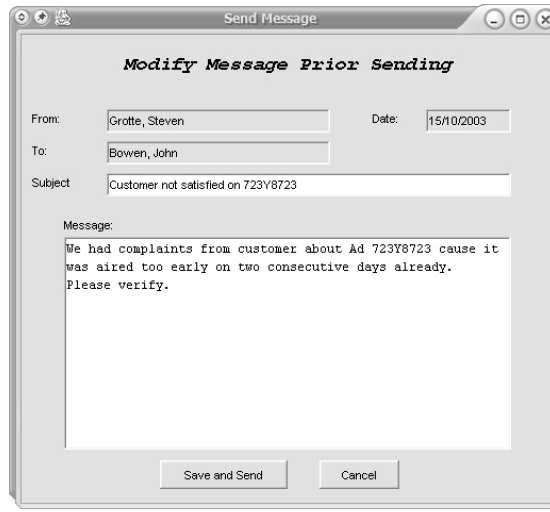


Рис. 19.8. Диалоговое окно для корректировки исходящего сообщения

ключением того, что показывается информация существующего исходящего сообщения.

Диалоговое окно позволяет выбирать делового партнера и назначать намеченного служащего и/или намеченный отдел. Тема и содержимое сообщения могут быть изменены в соответствующих полях. Окно содержит две командные кнопки:

- `OK and Return` (все в порядке и вернуться) — сохранить скорректированное исходящее сообщение в БД и затем удалить это окно и вернуться к окну `Display Message Text`.
- `Cancel and Return` (отменить и вернуться) — отменить операцию, удалить это окно и вернуться к окну `Display Message Text`.

S8 — Produce Daily Report (формирование ежедневного отчета)

Эта опция активизируется из главного окна приложения (рис. 19.3). Она позволяет конструировать итоговый отчет, который показывает все операции по электронной почте в течение конкретного дня с деловым партнером. После активизации этой опции пользователю выводится элемент управления — календарь, который позволяет выбрать дату, на которую формируется отчет. Отчет отображает следующую информацию:

- дата операций по электронной почте;
- информация о деловом партнере (имя, фамилия, организация);
- число созданных сообщений для электронной почты (для данного делового партнера на данную дату);
- число сообщений, посланных по электронной почте (для данного делового партнера на данную дату);

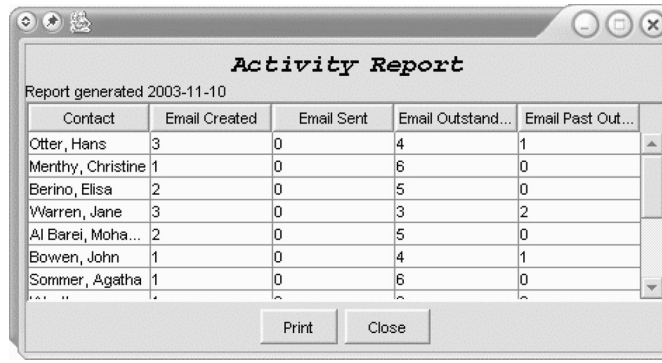


Рис. 19.9. Формат отчета по ежедневным операциям

- число невыполненных сообщений для электронной почты (для данного делового партнера и намеченных на данную и последующие даты, но все еще не посланных);
- число невыполненных в прошлом сообщений для электронной почты (для этого делового партнера и намеченных в прошлом, но все еще не посланных).

Рис. 19.9 показывает формат отчета по ежедневным операциям. После просмотра отчет может быть распечатан.

S9 — Maintain Authorization Rules (поддержка правил авторизации)

Эта опция может или не может быть доступной из главного окна приложения EM (рис. 19.3). Она может быть реализована как отдельная функция «системы сопровождения» приложения, доступная системным администраторам.

Диалоговое окно для S9 (рис. 19.10) представляет собой таблицу, подобную таблице 19.1. Только системному администратору (ADM-отдел) позволено изменять значения true/false ячеек таблицы.

Некоторые исходящие сообщения в таблице OutMessage — исходящие сообщения (рис. 19.15) могут и не быть намечены для какого-либо служащего или отдела (то есть, и `sched_emp_id` — идентификатор намеченного слу-

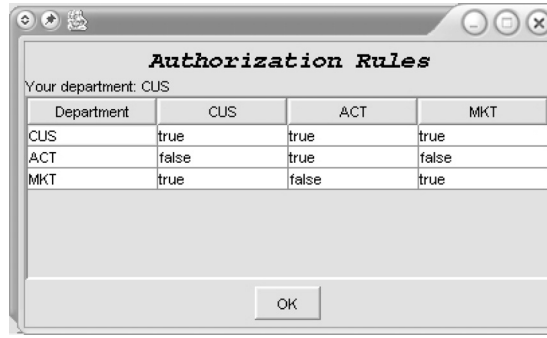


Рис. 19.10. Поддержка правил авторизации

жащего, и `sched_dept_code` — код намеченного отдела могут иметь значения NULL). Исходящие сообщения в таком случае неявно намечаются для Quality Control (QUA) — отдела контроля качества. Это означает, что EM-приложение должно внести их в список для передачи по электронной почте служащим QUA.

19.2.4. Потоки исключений

Любое исключение, возникающее в течение времени выполнения, должно быть выявлено и передано пользователю посредством информационного окна сообщения. Программа не должна нарушать свою работу при любом исключении, поступающем от БД, от сетевых отказов электронной почты или от приложения.

E1 — Incorrect username or password (неправильное пользовательское имя или пароль)

Если в *основном потоке* актор вводит неправильное пользовательское имя или неправильный пароль, система отображает сообщение об ошибке (рис. 19.11). Сообщение выглядит следующим образом: «Invalid login. Please try again. This will be your second attempt.» (Неправильное регистрационное имя. Пожалуйста, попробуйте снова. Это будет ваша вторая попытка.) Сценарий использования продолжается и позволяет пользователю заново ввести пользовательское имя и пароль. Вторая неудачная попытка завершается сообщением об ошибке: «Invalid login. Please try again. This will be your last attempt.» (Неправильное регистрационное имя. Пожалуйста, попробуйте снова. Это будет ваша последняя попытка.) Сценарий использования продолжается и позволяет пользователю заново ввести пользовательское имя и пароль.

Актору даются три возможности, чтобы задать правильное пользовательское имя и пароль. Если все три раза будут неудачны, система выводит сообщение: «Exceeded the maximum of three times to login. Application will quit.» (Превышен максимум из трех попыток ввода регистрационного имени. Приложение будет завершено.) — рис. 19.12. Сценарий использования завершается.

E2 — Email could not be sent (сообщение по электронной почте не может быть послано)

Если в подпотоке S4 почтовый сервер возвращает сведение об ошибке, что сообщение по электронной почте не могло быть послано, система сообщает актору, что сообщение не было послано. Сообщение содержит текст: «Failed in emailing the message. No network connection or invalid email address.»

Рис. 19.11. Информационное окно для неправильного регистрационного имени

Рис. 19.12. Информационное окно после трех неудачных попыток ввести регистрационное имя.

Рис. 19.13. Информационное окно при ошибке передачи по электронной почте

(Ошибка при передаче сообщения по электронной почте. Нет сетевого соединения или неправильный адрес электронной почты.) — рис. 19.13. Сценарий использования продолжается после того, как пользователь подтверждает получение сообщения об ошибке, нажав кнопку `OK`.

19.3. Концептуальные классы и реляционные таблицы

Концептуальная модель классов для итерации 3 содержит шесть классов: `Employee` (служащий), `Department` (отдел), `Contact` (деловой партнер), `OutMessage` (исходящее сообщение), `DailyReport` (ежедневный отчет) и `Authorization` (авторизация) — рис. 19.14. Линии ассоциаций снабжены именами ассоциаций и множественностями. Роли ассоциаций в большинстве случаев поименованы. Ассоциации от `Authorization` к `Department` являются однонаправленными (обозначено стрелками на линиях ассоциаций). Первичные атрибуты (идентификаторы) напечатаны полужирным шрифтом.

`Employee` может быть `employed_by` (служащим) максимум одного `Department`. `Department` нанимает многих `Employee`, но он может существовать, даже если в нем не служит ни один `Employee`. Административная структура служащих отражена в отношении `employment_hierarchy` (иерархия служащих).

`Department` может иметь намеченные `OutMessage` (исходящее сообщение). Может быть максимум один `Department` для каждого `OutMessage`. Имеется точно один `Contact` для каждого `OutMessage`.

`Contact` может быть связан с нулем или многими `OutMessage` и с нулем или многими `DailyReports`. Каждый объект `DailyReport` идентифицирован параметрами `date_of_report_day` (дата дня отчета) и `contact_id` (идентификатор делового партнера).

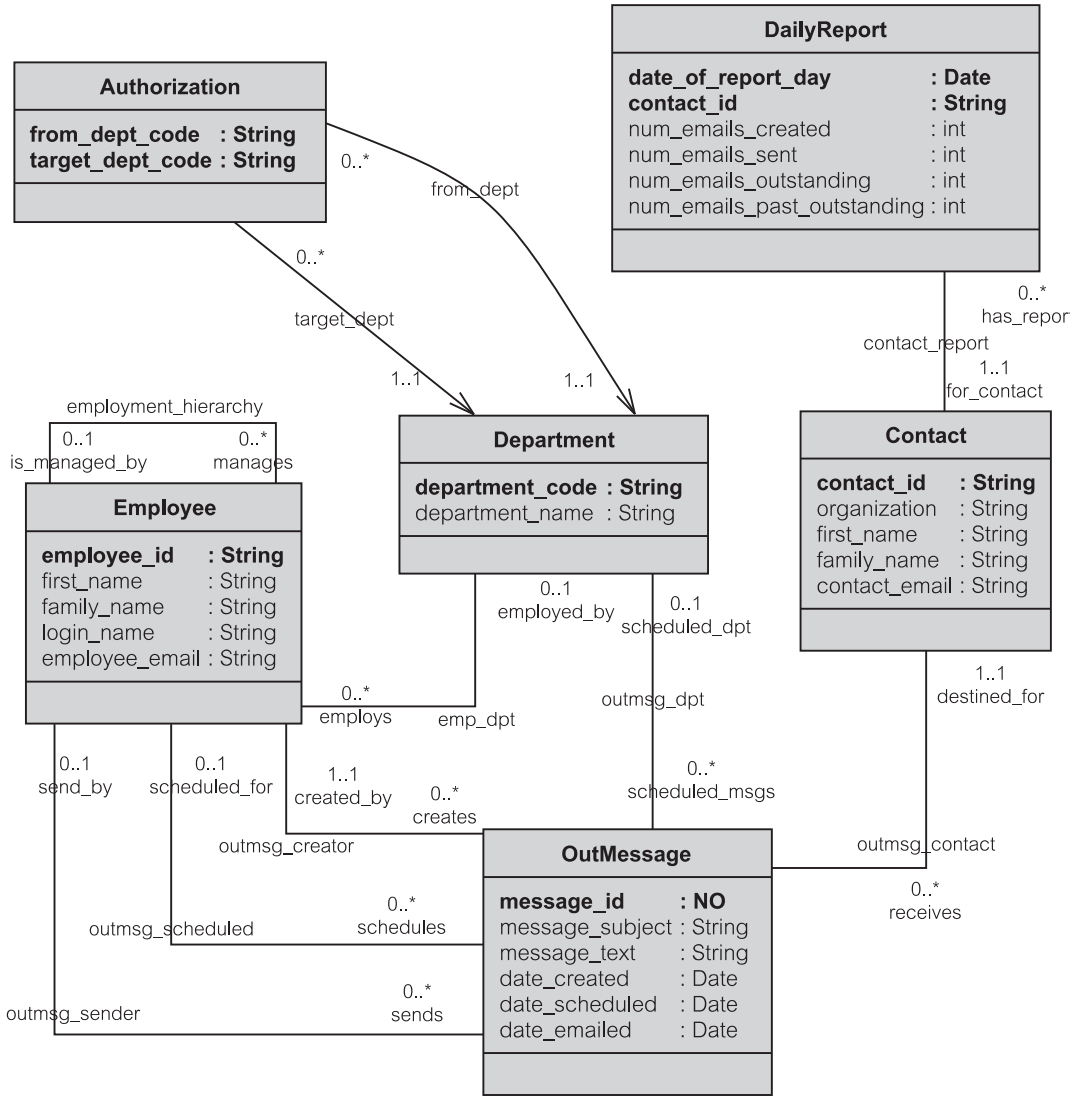


Рис. 19.14. Концептуальные классы для итерации 3

Ассоциация по имени `outmsg_creator` (создатель исходящего сообщения) связывает `OutMessage` с `Employee`, который создал это `OutMessage` в БД. Ассоциация по имени `outmsg_scheduled` (намеченный для исходящего сообщения) определяет `Employee`, который намечен для передачи исходящего сообщения по электронной почте. Эта ассоциация связывается с нулем `Employee`, если `OutMessage` не назначено никакому `Employee`. Ассоциация по имени `outmsg_sender` (отправитель исходящего сообщения) определяет `Employee`, который передает по электронной почте `OutMessage`. Эта ассоциация связана с нулем `Employee`, пока `OutMessage` не отослано по электронной почте.

Authorization содержит только два атрибута: from_dept_code (код исходного отдела) и target_dept_code (код целевого отдела). Этого достаточно, чтобы реализовать требования авторизации, определенные в таблице 19.1. from_dept_code соответствует строкам, а target_dept_code — столбцам таблицы 19.1. Таблица Authorization содержит величины только для тех ячеек в таблице 19.1, которые имеют значение true (истина). Значения false (ложь) явно не помещаются в Authorization.

Ассоциации от Authorization к Department являются однонаправленными. Это характеризует различие между пользователями EM-приложения и системными администраторами, ответственными за управление доступом к БД. Конечно, EM-приложение все еще может запрашивать (когда это необходимо) таблицу Authorization, однако нет никакой навигационной зависимости от Department к Authorization. Это означает, что преобразование концептуальной модели к реляционной схеме (рис. 19.15) не может охватить односторонние навигации (потому что понятие ссылочной целостности реляционных БД не является «навигационным»).

Большинство атрибутов имеет тип string (строка). Тип атрибута message_id — number (число), обозначаемый как NO. Четыре атрибута дат имеют тип date (дата).

Концептуальная модель классов может быть автоматически преобразована в **модель реляционной БД** с помощью правил объектно-реляционного соответствия, рассмотренных в главе 10. Рис. 19.15 показывает реляционную модель, полученную из концептуальной модели на рис. 19.14. Модель определяет таблицы и отношения между ними. Каждая таблица определяет столбцы, индексы и определенные пользователем триггеры (где они применимы). Имена наиболее важных хранимых процедур, определенных пользователем, показываются в нижней части диаграммы.

Отделение (область) индексов содержит индексы, созданные на первичных ключах (PK — Primary Key), на внешних ключах (FK — Foreign Key) и на уникальных столбцах (UN — Unique). Таблица Authorization не имеет явно заданного индекса на своем первичном ключе, потому что это очень маленькая таблица (однако первичный ключ, конечно, все еще требуется в БД).

Имеется несколько изменений в таблице OutMessage по сравнению с итерацией 2. Столбец department_code (код отдела) был переименован в sched_dpt_code (код намеченного отдела), чтобы подчеркнуть его цель. Для единообразия столбец scheduled_emp_id (идентификатор намеченного служащего) был переименован в sched_emp_id (то же название, но используется другое сокращение). Наконец, тип message_text (текст сообщения) был изменен на LONG (длинный). В Oracle тип LONG означает тип данных, представляющий последовательность символов переменной длины с максимальным размером до 2 Гб.

19.4. Дополнительная спецификация

Итерация 3 существенно воздействует на особенности FURPS+ EM-реализации, и все же это воздействие не всегда непосредственно отражается в дополнительных спецификациях. Особенности **FURPS+**, которые в первую очередь

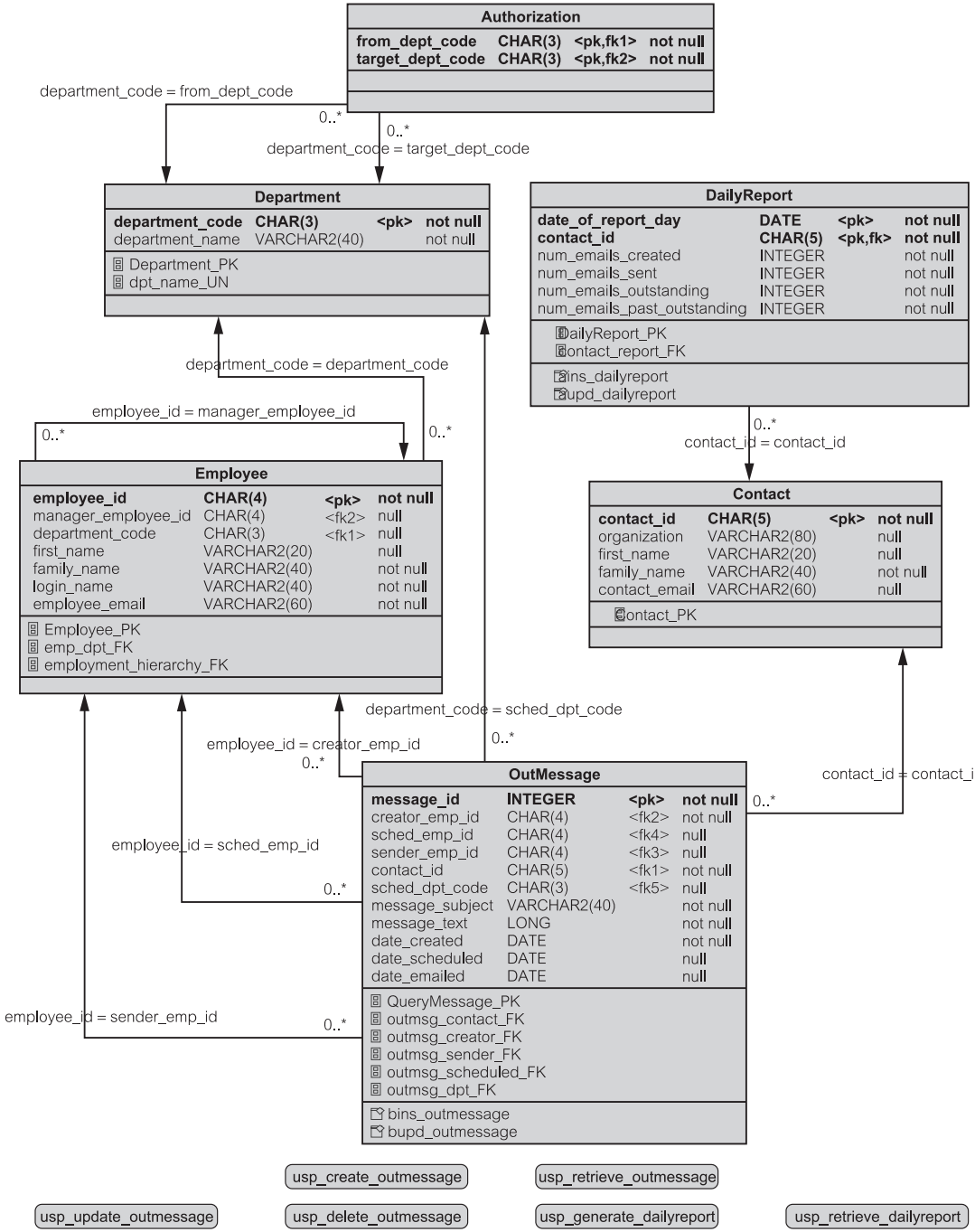


Рис. 19.15. Реляционная схема для итерации 3

характеризуют итерацию 3, — это выполнение функций и возможность сопровождения. Перемещая большую часть бизнес-логики из программы клиента в БД сервера, итерация 3 делает переход от «программирования в малом» к «программированию в большом».

- Функциональные возможности
  - EM — многопользовательское приложение.
  - Авторизация пользователя и права доступа к различным средствам приложения управляются централизованно из БД, с которой соединена прикладная программа.
- Применимость
  - Для грамотного в отношении к компьютерам пользователя не требуется никакого обучения, чтобы иметь возможность использовать итерацию 3 программы. Достаточно простого объяснения цели и основных особенностей приложения, чтобы использовать программу.
  - Итерация 3 представляет собой GUI-программу, которая обеспечивает интуитивный подход к использованию различных опций приложения. Одновременно может потребоваться только одна опция.
  - Итерация 3 выполняется и может быть развернута на рабочем столе GUI ОС Windows или как GUI на основе Web-технологии.
- Надежность
  - Приложение должно быть доступно 24 часа в каждый день недели. Не должно быть никакого связанного с БД времени простоя. О любом намеченном отключении для профилактики сервера электронной почты EM-пользователи электронной почты должны быть уведомлены, по крайней мере, за 24 часа.
  - Отказ программы не должен нарушить правильность и целостность БД. Пользователь должен иметь возможность повторно начать программу после отказа и удостовериться, что информация БД является непротиворечивой и не пострадала в результате отказа.
- Выполнение функций
  - Нет никакого верхнего ограничения числа одновременно работающих пользователей.
  - Время отклика системы должно оставаться стабильным для до 100 одновременно работающих пользователей. Небольшая потеря производительности функций приемлема для большего числа пользователей.
  - Время отклика должно быть меньше 5 секунд в 90 процентах случаев.
- Возможность сопровождения
  - Структурное проектирование системы должно соответствовать шаблону PCMEF+, чтобы обеспечить надлежащее удобство сопровождения и масштабируемость.

- Для создания кода используется управляемая тестированием разработка. Для проверки кода используются приемочные испытания. Испытательные модули, полученные при управляемой тестированием разработке и в процессе приемочных испытаний, используются для регрессионного тестирования, если код итерации 3 будет изменен.

- Другие ограничения
  - Проект должен использовать БД Oracle, но он должен быть легко переносимым на другие реляционные БД.
  - Итерация 3 должна использовать Java и JDBC или SQLJ для доступа из программы к БД Oracle.
  - Всякий раз, когда это возможно, бизнес-логика должна быть реализована на сервере БД в виде хранимых процедур или хранимых функций. Сложные бизнес-правила должны быть запрограммированы в триггерах.

19.5. Спецификация БД

Итерация 3 предъявляет специальные требования к возможности современных БД хранить программы, которые приложения могут вызывать. Эта возможность имеет огромные преимущества для клиентов приложения и для всех характеристик информационных систем предприятия. Преимущества касаются выполнения функций системы, ее безопасности, целостности, возможности сопровождения и т. д. Далее приведены главные требования, обеспечиваемые БД итерации 3 ЕМ.

- Там, где это практично, замените в приложении все SQL-операторы к БД на JDBC/SQLJ-вызовы к **хранимым процедурам**. Как минимум, все операторы модификации (insert — добавить, update — скорректировать, delete — удалить) должны быть заменены вызовами хранимых процедур. Некоторые высоко интерактивные операторы выбора (`select`) из БД, в которых условия запроса (в выражении `where` — где) динамически вводятся пользователем, вероятно, придется сохранить в клиенте приложения.
- Если все данные, требуемые клиентом приложения, могут быть получены с помощью хранимых процедур, пользователи приложения лишаются возможности *добавлять*, *корректировать*, *удалять* и (возможно также) *выбирать* данные из/в БД. Замените эти функции предоставлением пользователям возможности *выполнять* все хранимые процедуры, доступные приложению. Это приведет к усилению **безопасности** БД, поскольку пользователи не смогут повредить объекты БД, соединяясь с БД за счет ЕМ и непосредственно обращаясь к данным, используя интерактивный инструмент SQL (то есть, средство запросов SQL типа SQL*PLUS СУБД Oracle) [13].
- Выполните рефакторинг проекта приложения для **обработки транзакций**. Идентифицируйте транзакции, определите, где они начинаются и завершаются, установите уровни изоляции транзакций, чтобы максимизировать параллелизм (считайте, что уровень изоляции может быть применен к таблице, нежели ко всему соединению или транзакции) и т. д. Всякий раз, когда будет возможно, управляйте границами транзакций в хранимых процедурах, а не в приложении.

- Требуемые хранимые процедуры и/или функции:
  - Извлечение исходящих сообщений и связанных с ними данных из связанных таблиц (`usp_retrieve_outmessage` — хранимая процедура извлечения исходящего сообщения):
    - просматриваемых конкретным служащим;
    - назначенных конкретному отделу;
    - намеченных конкретному служащему.
  - Создание исходящего сообщения (`usp_create_outmessage` — хранимая процедура формирования исходящего сообщения).
  - Удаление исходящего сообщения (`usp_delete_outmessage` — хранимая процедура удаления исходящего сообщения).
  - Изменение исходящего сообщения (`usp_modify_outmessage` — хранимая процедура изменения исходящего сообщения).
  - Формирование ежедневного отчета (`usp_generate_dailyreport` — хранимая процедура формирования ежедневного отчета).
  - Извлечение ежедневного отчета (`usp_retrieve_dailyreport` — хранимая процедура извлечения ежедневного отчета).

- Требуемые **триггеры**:
  - При вставке/обновлении в таблице `OutMessage` — исходящее сообщение (`bins_outmessage` — вставка исходящего сообщения и `bupd_outmessage` — корректировка исходящего сообщения на рис. 19.15):
    - Проверьте, что `sched_emp_id` (идентификатор намеченного служащего) и `sched_dpt_code` (код намеченного отдела) удовлетворяют ссылочной целостности между таблицами `Employee` (служащий) и `Department` (отдел).
    - Обеспечьте условие, что если существует `sched_emp_id`, то должен существовать и `sched_dpt_code` (фактически EM-приложение должно запросить БД заполнить `sched_dpt_code` при любой вставке/корректировке `sched_emp_id`).
    - Обеспечьте условие, что если не существуют `sched_emp_id` и `sched_dpt_code`, то параметру `sched_dpt_code` задано значение Quality Control Department (отдел контроля качества).
    - Обеспечьте условие, что `manager_employee_id` (идентификатор менеджера служащего) для каждой записи `Employee` равен или NULL, или указывает на служащего того же самого отдела, что и запись `Employee`, за исключением независимого служащего, который может иметь менеджера из любого отдела.
    - `date_scheduled` (запланированная дата) и `date_emailed` (дата передачи по электронной почте) должны быть больше чем или равняться `date_created` (дата создания).

- При вставке/обновлении в таблице `DailyReport` — ежедневный отчет (`ains_dailyreport` — вставка ежедневного отчета и `aupd_dailyreport` — корректировка ежедневного отчета на рис. 19.15):
  - Обеспечьте для каждой записи, что:

```
num_emails_created = num_emails_sent +
num_emails_outstanding + num_emails_past_outstanding
```

Резюме

1. Итерация 3 нацелена на перемещение *бизнес-логики* от кода приложения к серверу БД. Вся обработка БД выполняется хранимыми процедурами, которые вызываются кодом Java-приложения, когда это требуется.
2. Итерация 3 повышает возможности надежности, выполнения функций и сопровождения FURPS+ системы.
3. В итерации 3 имеются четыре новых сценария использования: `Apply Authorization Rules` (применение прав доступа), `Maintain Authorization Rules` (поддержка прав доступа), `Produce Daily Report` (формирование ежедневного отчета) и `Modify Message Before Emailing` (корректировка сообщения перед передачей по электронной почте).
4. Итерация 3 определяет и реализует строгие *права доступа* к БД пользователями приложения.
5. *Концептуальная модель классов* для итерации 3 содержит шесть классов: `Employee` (служащий), `Department` (отдел), `Contact` (деловой партнер), `OutMessage` (исходящее сообщение), `DailyReport` (ежедневный отчет) и `Authorization` (авторизация). Классы преобразуются в шесть таблиц в *модели реляционной БД*.
6. Итерация 3 определяет и реализует *границы транзакций* для доступа к данным из приложения.
7. Итерация 3 использует триггеры, чтобы управлять целостностью БД.
8. Хотя это и не рассмотрено в данной главе, варианты итерации 3 используют преимущества шаблонов, которые подходят для *бизнес-компонентов*.

Ключевые термины

Обзорные вопросы

1. В каком смысле сценарий использования `Maintain Authorization Rules` (поддержка прав доступа) отличается от других сценариев использования в итерации 3?
2. Объясните цель прав доступа в итерации 3. Могут ли права быть легко добавлены или изменены?
3. Итерация 3 имеет целью переместить бизнес-логику в БД. Что это означает в действительности? Как это может быть достигнуто?
4. Как вы думаете, какие функции итерации 3 должны быть помещены на границы транзакций? Каковы последствия реализации включения транзакций в текущий код?
5. Какова роль триггеров в итерации 3? Как они связаны с хранимыми процедурами?

Глава

20 Безопасность и целостность

Программная инженерия тесно связана с *инженерией баз данных*. Глава 10 охватила основные принципы БД с точки зрения программной инженерии. Она представила их реляционную модель. Темы, рассмотренные в главе 10, включают: реляционные структуры БД, декларативную и процедурную ссылочную целостность, хранимые процедуры, индексы и преобразование объектов в записи БД. Имеется много других проблем в программной инженерии, которые переплетаются с инженерией БД. Эта и следующая главы рассматривают проблемы проектирования БД с точки зрения программной инженерии.

Концентрация итерации 3 на проблемах БД поднимает темы проектирования *безопасности* и *целостности* (рассмотренные в этой главе) и проектирования *транзакций* и *параллелизма* (глава 21). Эти темы охватывают все уровни многоуровневых систем клиент/сервер, но уровень сервера БД принимает на себя большую часть обязанностей за фактическое выполнение задач. С точки зрения программной инженерии безопасность и целостность — механизмы задания бизнес-правил, в то время как транзакции и параллелизм — средства реализации логики приложения (и приспособление к правилам целостности в процессе).

Бизнес-правила диктуют, как должны быть реализованы проблемы безопасности и целостности. Они охватывают различные приложения и различных пользователей. Пользователю (или скорее рабочей роли, которую пользователю позволяют выполнять на предприятии) может быть разрешено работать с различными приложениями клиента, но авторизация безопасности, предоставленная этому пользователю/роли для работы с данными, должна быть одной и той же, независимо от того, с какого приложения осуществляется доступ к данным. Точно так же декларативные и процедурные ограничения целостности (раздел 10.1.4) не должны быть нарушены никаким приложением или пользователем, независимо от уровня авторизации, предоставленного ему.

С точки зрения коллектива пользователей безопасность имеет персональное (индивидуальное) измерение. Права доступа и уровень защиты назначаются конкретным пользователям и ролям, которые они исполняют. Целостность, с другой стороны, имеет мульти-персональное (группа пользователей) измерение. Декларативные и процедурные ограничения целостности, так же как и другие проверки целостности, применяются одинаково ко всем пользователям БД.

20.1. Проектирование безопасности

Рамакришнан и Герке [84] перечисляют три показателя для проектирования **безопасности** в информационных системах предприятия:

- **Секретность** — чтобы запретить раскрытие информации неавторизованным пользователям и приложениям.
- **Целостность** — чтобы запретить корректировку данных неавторизованными пользователями и приложениями.
- **Доступность** — чтобы позволить доступ в любое время авторизованным пользователям и приложениям.

Чтобы обеспечить эти показатели, пользователи и приложения, обращающиеся к БД, сначала идентифицируются, а затем им предоставляются права доступа, называемые **полномочиями**. **Идентификация** — задача проверки личности пользователя/приложения, пытающегося соединиться с БД. В простом сценарии идентификация выполняется на основе комбинации идентификатора пользователя и пароля.

Предоставление полномочий доступа к объектам БД является технологией так называемой **контролируемой авторизации** или **контролируемого управления доступом** (раздел 20.1.1). Полномочия имеют несколько измерений. Пользователям/приложениям могут быть предоставлены полномочия на:

- *объекты схемы* (типа запрещения создания новых таблиц);
- *объекты данных* (типа запрещения доступа к некоторым столбцам таблицы);
- *SQL-операции* (типа запрещения каких-либо изменений в БД);
- *выполнение процедур/функций* (типа разрешения выполнения некоторых конкретных хранимых процедур).

Контролируемая авторизация не эффективна в информационных системах предприятия с различными приложениями и большим числом пользователей, обращающихся к одной или нескольким БД. Такие системы нуждаются в более сложных схемах **авторизации**, обычно реализуемых поверх контролируемого управления.

Одна сравнительно устойчивая схема, которая хорошо работает во многих военных организациях и агентствах безопасности, — с однонаправленным потоком информации и уровнями защиты — называется **принудительной авторизацией** (раздел 20.1.2). При принудительной авторизации [84] пользователям/приложениям назначаются **уровни полномочий**, а объектам БД назначаются соответствующие *уровни безопасности*. Тогда при успешном соединении с БД система определяет, какой доступ к БД может иметь пользователь/приложение. SQL-стандарт определяет только контролируемое разрешение, но большинство продавцов БД имеет свои собственные средства, помогающие реализовать принудительную авторизацию.

Для ориентированных на бизнес организаций с частой реорганизацией штата и динамическими изменениями в приложениях требуется более гибкая схема. К сожалению, современные СУБД предлагают немного вариантов поддержки даже притом, что такие сложные схемы авторизации должны

быть размещены и управляться из самой БД непосредственно. За неимением лучшего термина эти более сложные схемы здесь называются **авторизацией предприятия** (раздел 20.1.3). Авторизация предприятия требует, чтобы права доступа всех пользователей и приложений были размещены в *БД авторизации предприятия*, которую каждое приложение должно анализировать, чтобы определить, что текущий пользователь может, а что не может делать в отношении объектов БД и в отношении объектов приложения (типа возможности использовать элемент меню).

20.1.1. Контролируемая авторизация

Профессиональная фразеология контролируемой авторизации определяется стандартом SQL. Она сосредоточена на понятии **полномочий**. Все **права доступа** к объектам должны быть явно предоставлены как полномочия. SQL-команды `grant` (предоставить) и `revoke` (отменить) используются, чтобы давать или удалять полномочия. Некоторые системы БД, типа SQL Server, дополнительно реализуют команду `deny` (запретить), которая запрещает пользователю/приложению выполнять некоторые действия с объектом.

Полномочия могут быть предоставлены/отменены пользователям и/или ролям. **Роль** обычно относится к виду работы (например, роль преподавателя). Авторизация, основанная на ролях, позволяет определять права доступа в терминах организационной структуры предприятия. *Пользователь* может быть назначен на одну или большее количество ролей. Каждая роль имеет набор полномочий, предоставленных ей. Поэтому права пользователя определяются объединением наборов полномочий, предоставленных всем ролям, на которые назначен пользователь. Листинг 20.1 — пример SQL-команд, чтобы предоставить полномочия пользователям и ролям.

Листинг 20.1. Предоставление полномочий пользователям и ролям

```
Предоставление полномочий пользователям и ролям

create user anne identified by annepsswd;
create user michael identified by michaelpsswd;
create role student;
create role teacher;
grant create session to anne;
grant create session to michael;
grant select on grades to student;
grant update on grades to teacher;
grant student to anne;
grant student to teacher;
grant teacher to michael;
```

В листинге 20.1 создаются пользователи `anne` (Энн) и `michael` (Майкл) и роли `student` (студент) и `teacher` (преподаватель). Пользователю предоставлены полномочия `session` (сеанс). Полномочия `session` необходимы для связи с БД и проверки регистрационного имени. Ролям затем предостав-

ляются дополнительные полномочия. Пользователь `anne` получает роль `student`. Роль `student` затем предоставляется роли `teacher`. Это означает, что `michael` в роли `teacher` «наследует» полномочия `student` и он может как `select` (выбрать), так и `update` (корректировать) таблицу `grades` (оценки).

Как только полномочия роли будут установлены, изменения их будут происходить редко. Задачи авторизации будут тогда зависеть от назначений пользователям ролей. Это совместимо с тем, что происходит в организациях — описания работ (роли) относительно стабильны, но пользователи (служащие) часто меняют работу. Это также является причиной, почему понятие роли эффективно заменило прежнее SQL-понятие **группы** пользователей. Создание «виртуальных» групп пользователей (то есть ролей) и предоставление им групповых учетных записей заменяет потребность в формальном понятии SQL-группы.

Системные и объектные полномочия

Полномочия группируются в системные полномочия и объектные полномочия. **Системные полномочия** — право выполнить конкретное действие на системе БД в целом или действие в отношении схемы БД (логическая или физическая структура). `create session` (создать сеанс) в листинге 20.1 — пример системных полномочий. Другие примеры системных полномочий — возможность создавать, изменять и удалять таблицы или создавать и удалять индексы. В общем, системные полномочия предоставляются только администраторам БД и инженерам ПО. Пользователи приложения требуют только объектные полномочия (и системные полномочия соединиться и зарегистрироваться в БД).

Объектные полномочия определяют, как пользователи/приложения могут работать с содержимым БД (то есть, **экстенсивная работа** с БД в противоположность **интенсивной работе**, определяемой схемой). Объектные полномочия определяют, какие SQL-операции могут быть выполнены над конкретными объектами БД. Для различных типов объектов возможны различные полномочия. Объектные полномочия для таблиц и представлений включают права `select` (выделить), `insert` (добавить), `update` (скорректировать) и `delete` (удалить). Для хранимых процедур и функций используется единственное объектное полномочие — `execute` (выполнить) — право выполнить процедуру/функцию.

Полномочия можно передавать от пользователя/роли к другому пользователю/роли. Предоставление роли пользователю является одним из способов *передачи полномочий*. Другой способ (только для пользователей) — использование выражения SQL `with grant option` (с опцией предоставления), например,

```
grant delete on grades to diana with grant option;
```

Конечно, пользователь может передавать полномочия другому пользователю, только если он сам имеет эти полномочия. Скажем, пользователь, который создает таблицы, должен, прежде всего, иметь все необходимые полномочия на это и тогда может передавать эти полномочия другим. Процессы

передачи полномочий могут быть изображены в виде **графа авторизации** [84, 93]. Узлы графа представляют пользователей/роли. Дуги (стрелки) между узлами обозначают передачу полномочий. Передача полномочий начинается в корне графа, который представляет **администратора БД** (database administrator — DBA). Пользователь/роль имеет конкретные полномочия, если существует путь от корня до узла, представляющего пользователя/роль. Рис. 20.1 является примером такого графа.

Граф авторизации на рис. 20.1 различает узлы пользователей (тонкие круги) и узлы ролей (утолщенные круги); `DBA` — корневая роль, которая представляет любого пользователя, выполняющего эту роль (в большинстве систем БД роль `DBA` представляет собой предопределенную стандартную роль с наиболее мощными полномочиями). Граф иллюстрирует определения листинга 20.1.

Дуги снабжаются информацией о передаваемых полномочиях. Например, `(diana, teacher, delete on grades)` информирует, что пользователь `diana` (Диана) передал полномочия `delete on grades` (удалять оценки) роли `teacher`. Обратите внимание, что `michael`, который находится в роли `teacher`, имеет полномочия `select`, `update` и `delete` по отношению к `grades`.

Если желательно, роли могут быть **идентифицированы**. Это означает, что ролям могут быть назначены пароли. Это также означает, что все узлы (пользователи и роли) являются в технических терминах SQL *идентификаторами авторизации*. Однако в большинстве систем БД `with grant option` (с опцией предоставления) не применяется к объектным полномочи-

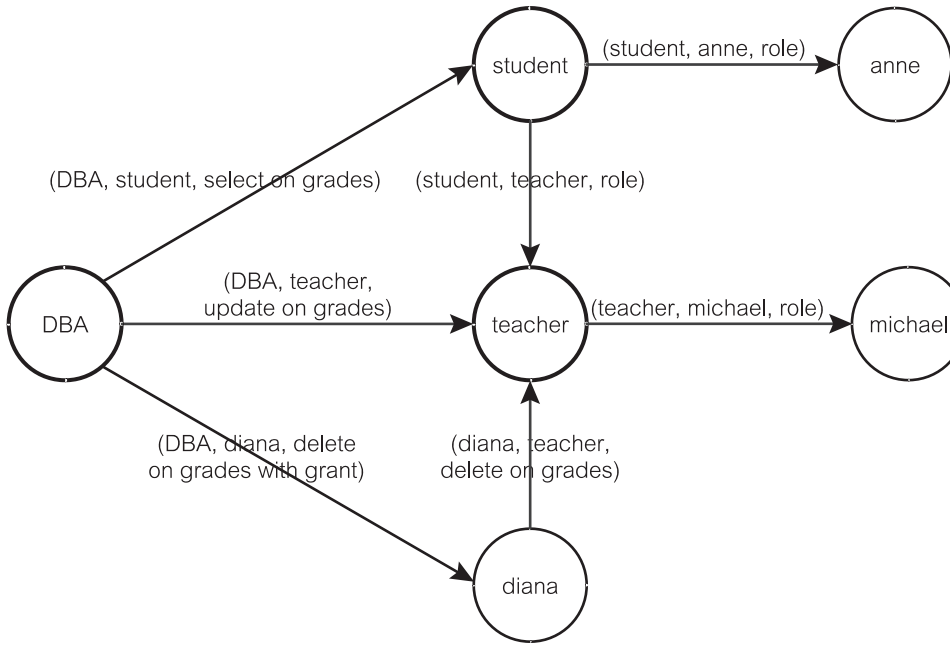


Рис. 20.1. Граф авторизации

ям, предоставленным ролям, независимо от того, идентифицированы они или нет. Это не разрешает пользователям, которым предоставлены некоторые роли, распространять объектные полномочия этих ролей на других пользователей/роли.

Для предоставленных полномочий можно **отменить разрешение**. Оператор SQL `revoke` используется, чтобы удалить полномочия от пользователя или роли. Синтаксис оператора `revoke` подобен оператору `grant`, например:

```
revoke delete on grades from diana;
```

В некоторых системах, типа SQL Server, полномочия могут также быть **запрещены** путем использования нестандартного SQL-оператора `deny`. Запрещение полномочий на любом уровне (пользователь или любой уровень ролей) имеет приоритет по отношению к тем же самым полномочиям, предоставленным на другом уровне. Оператор `deny` переопределяет другие полномочия при объединении их от многих ролей. Если пользователь `michael` является участником ролей `student` и `teacher`, и роль `student` предоставляет полномочия на объект, но роль `teacher` запрещает те же самые полномочия, то для `michael` будет запрещен доступ. Это происходит потому, что запрещение полномочий имеет приоритет по отношению к предоставленным полномочиям.

В присутствии аннулирования и отмены полномочий **эффективные полномочия**, предоставленные пользователю по отношению к объекту, определяются объединением всех предоставленных, запрещенных или отмененных полномочий, заданных для пользователя и для ролей, которым он принадлежит. Рис. 20.2 показывает UML-диаграмму деятельности для изображения того, как вычисляются и предоставляются эффективные полномочия [64].

Пользователь `Mary` (Мэри) делает попытку доступа к объекту `X`. Условие защиты объявляет, что `Mary` не имеет прямых полномочий по отношению к `X`, предоставленных ее учетной записью. Чтобы определить, имеет ли `Mary` доступ к `X`, система получает объединение полномочий, которые `Mary` наследует от ее ролей. Если одна из ролей имеет требуемые полномочия к `X` *и* если эти полномочия по отношению к `X` явно не запрещены в любых других ролях `Mary`, то `Mary` может иметь доступ к `X`. Иначе `Mary` запрещен доступ к `X`, пока оператор `deny` не будет отменен.

Программная контролируемая авторизация

Системные и объектные полномочия представляют **декларативные** SQL-пути реализации контролируемой авторизации. Системные и объектные полномочия объявляются. Однако SQL обеспечивает также **программные** (процедурные) пути управления контролируемой авторизацией. Программная авторизация требует дополнительного SQL-программирования с определенной целью защитить доступ к данным. Результирующий код авторизации или объекты, которые являются результатами этого кода, последовательно назначают декларативные полномочия.

Имеются три главные цели программной контролируемой авторизации: представления, синонимы и хранимые процедуры. Они рассмотрены ниже.

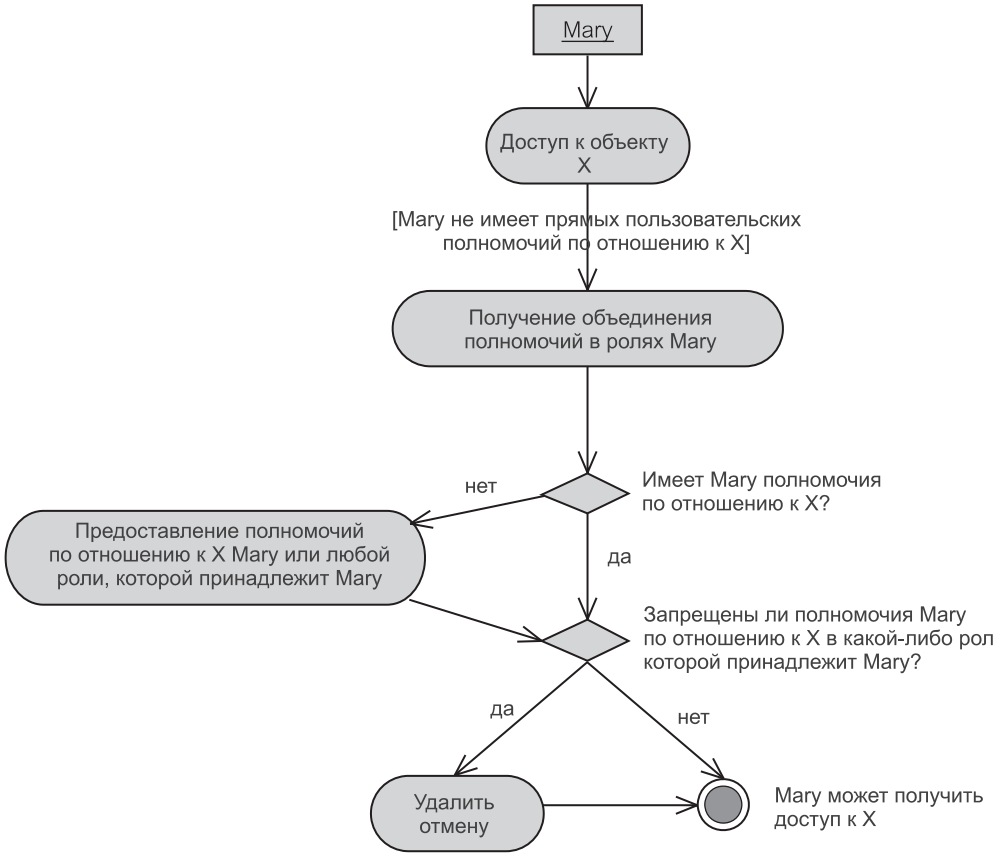


Рис. 20.2. Вычисление и предоставление эффективных полномочий

Использование представлений

Представление (или просматриваемая таблица) является результатом SQL-запроса (оператора выбора), которому дают имя, и его можно вызывать по этому имени. При вызове SQL-запрос выполняется и формирует результаты в *виртуальной таблице*. Таким образом, представление претендует быть таблицей, но в действительности оно всего-навсего является результатом выполнения своего запроса над одной или несколькими конкретными таблицами и/или над одним или несколькими другими представлениями.

Рис. 20.3 иллюстрирует три представления, определенные в БД MovieActor («Фильм-Актер») из главы 10 (рис. 10.5), которые здесь слегка расширены. Представления изображены графически, а также показаны соответствующие операторы `create view` — создать представление (двойные кавычки вокруг имен позволяют работать с чувствительными к регистру именами). Представления `actor_no_personal_info` (актер без персональной информации) и `expensive_actor` (дорогостоящий актер) определены на одной и той же таблице, а именно на таблице `actor` (актер). Первое является *представлением отображения*, которое запрещает видимость двух соответствующих столбцов

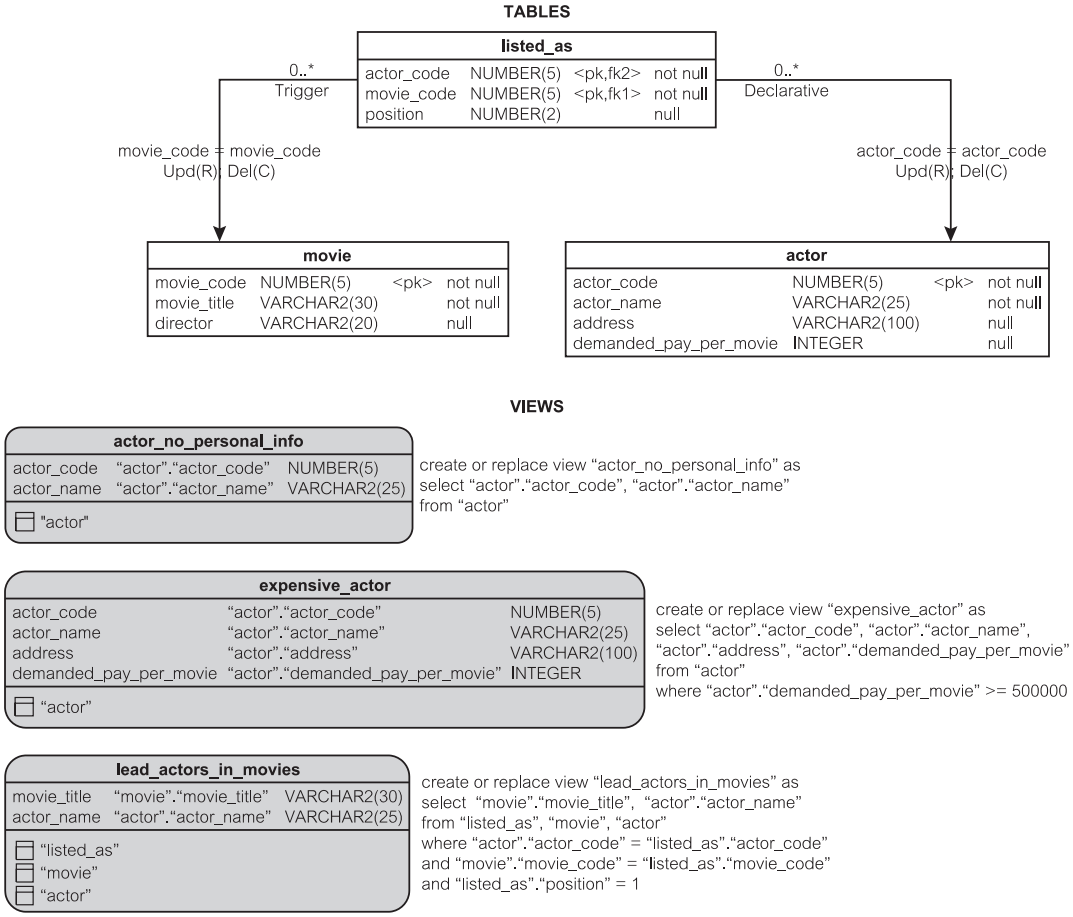


Рис. 20.3. Представления

в таблице `actor`. Второе является *представлением ограничения*, которое исключает видимость строк, где `demanded_pay_per_movie` (требуемая плата за фильм) меньше, чем 500 000. Представление `lead_actors_in_movies` (ведущие актеры в фильмах) является *присоединяющим представлением*, которое показывает только название кинофильма и имена актеров, которые перечислены как ведущие актеры (с `position` — позиции, равной 1).

Представления имеют много преимуществ, но также и ряд существенных ограничений. SQL-операторы выбора являются ориентированными на множества запросами, действующими на все таблицы. Но некоторым пользователям и ролям нельзя позволить видеть некоторые строки или некоторые столбцы таблиц. Решение разбить данные таблицы на несколько таблиц и предоставить ограниченные полномочия для каждой таблицы — не очень изящное решение для реализации и по другим причинам. Представления обеспечивают желаемый ответ; они обеспечивают дополнительный уровень безопасности таблицы, ограничивая доступ к предопределенному множеству строк и/или столбцов таблицы.

Главное ограничение заключается в том, что наиболее интересные представления не корректируемы, то есть их нельзя использовать как источник обновления (через них) реальных данных в конкретных таблицах. Следовательно, они не очень полезны для управления полномочиями операций добавления, корректировки и удаления.

Создание представления не требует так называемых **полномочий ресурса**, которые являются системными полномочиями, чтобы создать объекты БД. Чтобы создавать представление, пользователь должен иметь первоначально только полномочия `select` всех таблиц, упомянутых в представлении. После того как эти полномочия `select` предоставляются пользователю с `with grant option`, пользователь может передавать полномочия `select` представления другим.

Использование синонимов

Некоторые СУБД, включая Oracle вплоть до Oracle 8.1.6, объединяют концепции пользователя и схемы. Это непосредственное преобразование означает, что **схема** создана для каждой учетной записи. Такой подход удобен инженерам ПО и разработчикам, но он хуже в контексте пользователей приложений, которые могут иметь только доступ к совместно используемой БД и которые никогда не создают свои собственные объекты в своей собственной схеме.

Имеется множество решений этой дилеммы. Ни одно из них не совершенно за исключением полного разделения пользователя/схемы. Некоторые решения следующие [13]:

- Позволить всем пользователям совместно использовать единственную схему и, следовательно, одно и то же пользовательское имя и пароль. Это решение противоречит основным принципам авторизации и безопасности.
- Все еще позволить всем пользователям совместно использовать единственную схему, но размещать пользовательские имена и пароли в БД (управляемые той же самой схемой), чтобы идентифицировать пользователей системы.
- Использовать многократные **локальные схемы** (одна схема на пользователя), синонимы и одну **глобальную схему**. При этом решении каждый пользователь имеет свою собственную схему, но эти схемы не содержат никаких таблиц или данных; они только заполнены *синонимами*. Синонимы соединяют каждую схему с центральной схемой. Все запросы, выполненные на схемах пользователя, переадресуются к глобальной схеме через синонимы.

Синоним — это псевдоним (альтернативное имя) для таблицы, представления, хранимой процедуры, функции или другого синонима. Чтобы успешно использовать синоним, пользователь/роль должен иметь надлежащие полномочия к использованию целевых объектов БД. Синоним может быть создан в одной схеме для доступа к объектам в другой схеме. Он просто позволяет обеспечить доступ к объектам в другой схеме без использования имени квалификатора схемы. Чтобы создать синоним, используется оператор `create synonym` (создать синоним). Оператор, приведенный ниже, создает синоним,

называемый `actor` в одной схеме, чтобы обратиться к таблице `actor` в схеме `MovieActor` (фильм-актер):

```
create synonym actor for MovieActor.actor;
```

Синонимы поддерживают прозрачность местоположения и независимость данных. Используя синонимы, пользователь не знает реальное местоположение объектов. Достигается также определенный уровень независимости данных, потому что некоторые изменения в объектах БД, включая изменения имени, не будут влиять на приложения, работающие с синонимами.

С точки зрения авторизации синонимы не помогают с проблемой назначения полномочий, но они могут обеспечить улучшенную идентификацию пользователя в системах, которые объединяют схемы и пользователей. Когда это необходимо, полномочия назначаются объектам в *глобальной схеме*, типа MovieActor, которая содержит все объекты для приложения. В локальных схемах пользователей нет никаких определенных объектов БД кроме синонимов. Пользователи авторизованы в своих собственных *локальных схемах*, и все локальные схемы идентичны (они содержат один и тот же набор синонимов).

Интересная точка зрения использования синонимов — реализовать среду разработки ПО — предложена в [13]. В сценарии авторов разработчики имеют дело с глобальными объектами схемы через синонимы в их локальных схемах. Отбирая от разработчиков некоторые полномочия типа `alter` (изменить) или `drop` (удалить), синонимы не допускают беспорядка в глобальной схеме. Когда объект должен быть изменен, разработчик создает объект в своей локальной схеме, чтобы временно занять место синонима. Как только все программирование и тестирование будет закончено, разработчик может запросить, чтобы локальный объект заменил объект в глобальной схеме.

Использование хранимых процедур и функций

Синонимы — это только псевдонимы, и поэтому они не блокируют некоторые данные от видимости клиента (и всегда будет иметься некоторая информация в БД, которую не позволено показывать некоторым пользователям/ролям). Представления могут обеспечивать помощь в этом отношении, но они имеют некоторые важные недостатки. Лучшее решение управления доступом предлагается хранимыми процедурами и функциями (раздел 10.1.5). Процедуры и функции могут определить уровень доступа связанного клиента и обслуживать только разрешенные операции.

Хранимые процедуры и **хранимые функции** — предварительно скомпилированные части кода, которые имеют имена и могут иметь параметры. Процедуры и функции упрощают программирование приложения, перемещая ответственность за выполнение задачи от приложений непосредственно к БД. Процедуры и функции программируются однажды и могут быть вызваны из многих различных приложений. Это очень облегчает сопровождение программ. Будучи предварительно компилируемыми, хранимые процедуры/функции выполняются намного быстрее, чем соответствующие интерактивные SQL-операторы из кода клиента. Они также уменьшают сетевой трафик, поскольку вызов процедур и функций требует намного более коротких строк текста по сравнению с интерактивными командами SQL. То же самое касается возможного объема данных, возвращенных из БД.

Хранимые процедуры и функции неоценимы с точки зрения авторизации, поскольку они полностью инкапсулируют действия, выполняемые над БД. Некоторые из них могут быть проверками авторизации. Для пользователей хранимые процедуры и функции — черные ящики, возвращающие данные или только выполняющие некоторые действия без возврата данных. Пользователи не знают, что представляют собой команды SQL внутри процедуры/функции, и они не могут изменять команды до выполнения.

Единственное полномочие авторизации, присваиваемое хранимой процедуре или функции — полномочие `execute`. Пользователь может только вызывать хранимую процедуру или функцию, передавая ей параметры, и получать назад результаты из БД. Желательное дополнение заключается в том, что поскольку доступ приложения ко всем данным реализуется с помощью хранимых процедур и функций, то администратор БД может отменять у всех пользователей и ролей приложения полномочия `select`, `insert`, `update` и `delete` данных. Важное преимущество такого подхода — это то, что пользователи не могут обойти авторизацию, получая незаконный доступ к БД, используя интерактивное инструментальное средство SQL.

Листинг 20.2 является примером хранимой функции, называемой `usp_retrieve_employee_login` (хранимая процедура извлечения регистрационного имени служащего). Функция получает значения двух параметров: `login` — регистрационное имя, введенное пользователем, и `usern` — регистрационное имя/пользовательское имя, как оно известно в БД. Если две величины соответствуют друг другу, хранимая функция извлекает полную запись информации относительно `Employee` (служащий) и возвращает ее (в `REF_CURSOR` — ссылочный курсор) приложению.

Листинг 20.2. Хранимая функция usp_retrieve_employee_login

Хранимая функция usp_retrieve_employee_login

```
create or replace function usp_retrieve_employee_login(login IN
varchar2, usern in varchar2) return EMS.REF_CURSOR is
   c EMS.REF_CURSOR;
   dummy integer;
begin
   dummy := null;
   select 1 into dummy from dual where upper(login) =
                                                  upper(usern);
   open c for select * from "Employee" where "login_name" =
                                                       login;
   return c;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         raise_application_error(-20023,
'usp_retrieve_employee_login: unmatch username and loginname');
end;
```

Листинг 20.3 иллюстрирует, как хранимая функция, представленная в листинге 20.2, вызывается из приложения. Метод `retrieveEmployeeByName()` (извлечь служащего по имени) формирует строку вызова как параметр сообщения `query()` (запрос) объекта `reader` (читатель) — строка 437. Этот вызов возвращает множество результатов, которое размещается в переменной `rs` (результат). Строка комментария (строка 438) показывает, как непосредственный SQL-оператор `select` мог бы заменить вызов хранимой функции.

Внимательный читатель заметил бы, что запрос к `usp_retrieve_employee_login()` имеет только один параметр, а процедура в листинге 20.2 ожидает два параметра. Объяснение этой очевидной ошибки касается непосредственного преобразования между пользователем и схемой в некоторых СУБД.

Вместо использования синонимов для хранимых процедур и функций в локальных схемах разработчик может создавать имитации (скелеты) глобальных хранимых процедур в локальных схемах и использовать эти скелеты для переадресации выполнения к глобальной схеме. Листинг 20.4 представляет такое решение. Оператор `call` в листинге 20.3 вызывает хранимую функцию из листинга 20.4 теперь с одним параметром — строку `login`. Оператор `select` системной таблицы `user_users` (пользователь-пользователи) поставляет второй параметр хранимой функции в листинге 20.2.

Таблица `user_users` хранит пользовательское имя подключенного пользователя. Из-за непосредственного преобразования между пользователем и схемой может быть только одно пользовательское имя, связанное с локальной схемой. Это пользовательское имя (`usern`) — второй параметр в вызове `usp_retrieve_employee_login` глобальной схемы (вызывается в листинге 20.4 оператором `return` — возвратить). Обратите внимание, что попытка лишь переадресовать запрос к глобальной схеме и к `select username` (выбрать пользовательское имя) в этой глобальной схеме будет терпеть неудачу, потому что пользователь глобальной схемы отличается от текущего пользователя приложения (а пользователь приложения не знает сертификат или полномочия того пользователя).

Листинг 20.3. Вызов хранимой функции из метода приложения retrieveEmployeeByName()

Вызов хранимой функции из метода приложения retrieveEmployeeByName()

```
433: public IAEmployee retrieveEmployeeByName(String username) {
434:     EEmployee emp = null;
435:     java.sql.ResultSet rs=null;
436:     try {
437:        rs = reader.query("{ ? = call
              usp_retrieve_employee_login(?)}", new
                                    Object[]{username});
438:        //reader.query("select * from employee where
                        login_name = ' " + username + " ' ");
```

```
439:
440:         if (rs.next()) {
441:            emp = (EEmployee) createEmployee();
442:            fillEmployee(rs, emp);
443:            cache.registerEmployee(emp);
444:         }
445:      } catch (Exception exc) {
446:         exc.printStackTrace();
447:         return null;
448:      }finally{
449:         if(rs != null) reader.closeResult(rs);
450:      }
451:      return emp;
452:   }
```

Листинг 20.4. Хранимая функция usp_retrieve_employee_login в локальной схеме

Хранимая функция usp_retrieve_employee_login в локальной схеме

```
create or replace function usp_retrieve_employee_login(login IN
varchar2)
   return PSE3.EMS.ref_cursor is
   usern varchar2(20);
begin
   select username into usern from user_users;
   return PSE3.usp_retrieve_employee_login(login, usern);
end;
/
```

20.1.2. Принудительная авторизация

Принудительная авторизация нацелена на устранение некоторых лазеек контролируемой авторизации относительно идентификации пользователя с точки зрения его разрешений. Во-первых, принудительная авторизация проверяет, действительно ли пользователь, передающий для проверки пользовательское имя/пароль, является именно тем, кем он/она себя заявляет. Во-вторых, принудительная авторизация определяет нисходящие **уровни полномочий** для пользователей и **классы безопасности** в отношении к используемым объектам, чтобы устранить некоторые возможные обходы безопасности. Принудительная авторизация применяется в системах, где *секретность* может иметь большее значение, чем информационная целостность.

Первый аспект принудительной авторизации относится больше к *идентификации*, чем авторизации. Идентификация проверяет сертификат пользователя. Это может быть сделано рядом способов. Для более безопасных сцена-

риев может использоваться **подход «запрос-ответ»**, включающий **шифрование** [93]. Процесс может быть следующим:

1. Пользователь/приложение пробует соединиться с БД.
2. БД посылает строку запроса пользователю/приложению.
3. Пользователь/приложение использует предопределенный пароль (ключ шифрования), чтобы зашифровать строку запроса, и посылает зашифрованную строку назад к БД.
4. БД расшифровывает строку с тем же самым паролем и позволяет соединение, только если она получает первоначальную строку запроса.

Шифрование имеет то преимущество, что пароли не путешествуют по сети. Разновидность шифрования, использующая **открытый ключ,** имеет то дополнительное преимущество, что пароли не должны даже храниться в БД. Шифрование с помощью открытого ключа используется также в **цифровых подписях**.

Второй аспект принудительной авторизации — аспект авторизации по существу. Идея состоит в том, чтобы ограничить доступ к объектам, основанный на чувствительности содержащейся информации (классы безопасности) и базирующийся на уровнях полномочий пользователей, пытающихся осуществить доступ к этой информации. Классы безопасности и полномочия идут «рука об руку»; пользователь должен иметь полномочия, по крайней мере, столь же высокие, как и классификация объектов, чтобы получить доступ к ним. Схема называется принудительной, чтобы подчеркнуть требование, что классификации и документы поддерживаются централизованно **администраторами безопасности** и не могут быть изменены пользователями.

Широко известный вариант принудительной авторизации, называемый **модель Bell-LaPadula**, имеет дело с четырьмя концепциями [84]:

- объекты (например, таблицы, представления, столбцы, строки, хранимые процедуры);
- субъекты (например, пользователи, прикладные программы);
- классификации объектов;
- полномочия субъектов.

Как было сказано, классификации и полномочия идут вместе. К обоим применяется одно и то же множество уровней. Одна простая возможность состоит в том, чтобы иметь четыре уровня классификаций/полномочий:

- совершенно секретно;
- секретно;
- конфиденциально (для служебного пользования);
- несекретно.

Как только объектам будут даны уровни классификации, а субъектам назначены уровни полномочий, субъект с некоторыми полномочиями может иметь доступ только к объектам с той же самой или пониженной классификацией. Это правило составляет дополнительную проверку после того, как субъект будет удовлетворять элементам управления контролируемого доступа.

20.1.3. Авторизация предприятия

Технологии контролируемой и принудительной авторизаций недостаточны, когда они сталкиваются с информационными системами предприятия, содержащими, возможно, сотни приложений, тысячи пользователей и несколько общих БД. В таких ситуациях должна быть добавлена системная политика авторизации, которая не может быть изменена индивидуальными пользователями или даже разработчиками. Это является предпосылкой **авторизации предприятия**.

Контролируемая и принудительная авторизации концентрируются на объектах и пользователях БД и пренебрегают объектами программы клиента и приложениями. В действительности же проблемы авторизации охватывают программы клиента и БД. Например, пользователь приложения, которому не разрешено видеть оценки студентов в БД, даже не должен иметь доступной в программе опции меню, позволяющей просматривать оценки. Другими словами, полномочия на объекты БД и на объекты приложения должны быть непротиворечивы и синхронны. Кроме того, один и тот же пользователь может действовать во многих ролях, каждая с различными полномочиями по отношению к БД. Система авторизации должна быть способна определить полномочия, которые относятся к пользователю, когда этот пользователь использует конкретную прикладную программу.

Так как роль определяется как множество полномочий, она не может охватывать несколько различных приложений, обращающихся к одной и той же БД. По этой причине увеличивающееся число систем БД вводит понятие **прикладной роли**, отличающейся от роли пользователя. Использование прикладных ролей предоставляет новую возможность объединения авторизации на объекты сервера с авторизацией на объекты клиента.

Полномочия на прикладные роли назначаются так же, как и пользователям и ролям пользователя. Однако имеются важные различия. Израэль и Джонс [42] перечисляют следующие различия между прикладной ролью и ролью пользователя в БД SQL Server:

- Прикладные роли не имеют никаких членов.
- Прикладные роли по умолчанию неактивны (они активизируются приложением, когда оно запускается).
- Все эффективные полномочия (раздел 20.1.1) текущего пользователя удаляются и применяются только полномочия, назначенные прикладной роли.

СУБД типа Oracle, которые обычно устанавливают схему для каждого пользователя, обеспечивают *расширенные средства безопасности*, которые реализуют сходные концепции, но не всегда под тем же самым именем. Oracle использует понятие **пользователя предприятия** (или **независимого от схемы пользователя**), чтобы поддержать функциональные возможности прикладной роли. Oracle обеспечивает специальную службу каталогов, чтобы разместить пользователя (так называемое Distinguished Name (DN) — различающее имя пользователя) в пользователе предприятия, когда пользователь имеет доступ к БД через приложение. Приложение связано с так называемой **совместно используемой схемой**, а на совместно используемой схеме определяется множество полномочий для приложения.

В противоположность прикладной роли, используемой в SQL Server, множество полномочий пользователя предприятия в Oracle представляет собой объединение полномочий, связанных с совместно используемой схемой, и любых дополнительных полномочий, которые связаны с ролями, предоставленными пользователю в службе каталогов. Размещение пользователей предприятия через службу каталогов распространяется на многие БД, к которым пользователю потребуется доступ, и многие приложения, которые пользователю нужно будет использовать.

Поддержка, данная существующими СУБД для авторизации предприятия, может быть достаточна для управления доступом прикладных пользователей к объектам БД, но в них все еще нехватает обеспечения управления объединенным доступом как к объектам БД (сервер), так и к объектам приложения (клиент). Кроме того, механизм авторизации доступных решений может быть слишком сложным, чтобы управлять некоторыми организациями. В таких случаях или для расширений существующих решений инженеры ПО могут создать специальную **БД авторизации** в поддержку авторизации предприятия [64]. Каждое приложение, когда оно запущено пользователем, запрашивает БД авторизации, чтобы определить полномочия пользователя и настроить себя на этого пользователя.

Авторизация предприятия через БД авторизации все еще требует, чтобы пользователь связался с сервером БД, используя предопределенное регистрационное имя пользователя (однако само регистрационное имя пользователя может лишь предоставить полномочия соединения с БД). Без регистрационного имени пользователя просмотр конкретных операций пользователя (через приложение) над БД будет невозможен. После осуществления соединения регистрационное имя к прикладной роли становится прозрачно для пользователя, а приложение получает из БД авторизации эффективные разрешения.

Рис. 20.4 представляет собой пример частного проекта БД авторизации [61, 64]. Центральная таблица в этом проекте — `ApplicationRole` (прикладная роль). Прикладная роль активизируется для *соединения* (сеанс пользователя). Настройки авторизации, которые пользователь получает из прикладной роли, остаются в силе, пока регистрационное имя пользователя не будет удалено из программы приложения (и, следовательно, регистрационное имя из сервера БД).

Модель на рис. 20.4 предполагает, что БД авторизации используется программой клиента, чтобы установить настройки авторизации по отношению к объектам как клиента, так и сервера. Это совместимо с требованием, что неавторизованному пользователю должно быть запрещено выполнять неавторизованные операции на клиенте. Если операция клиента может поставить под угрозу целостность БД, то на сервере также реализуется второй уровень защиты.

БД авторизации поддерживает в `ClientObject` (объект клиента) каталог всех *объектов клиента* в приложении, которые являются субъектом для настроек авторизации. Каталог включает окна приложения, поля окон и элементы управления окон. Поддерживаются также любые иерархические отношения между объектами клиента (типа участков окна, полей в окне или ие-

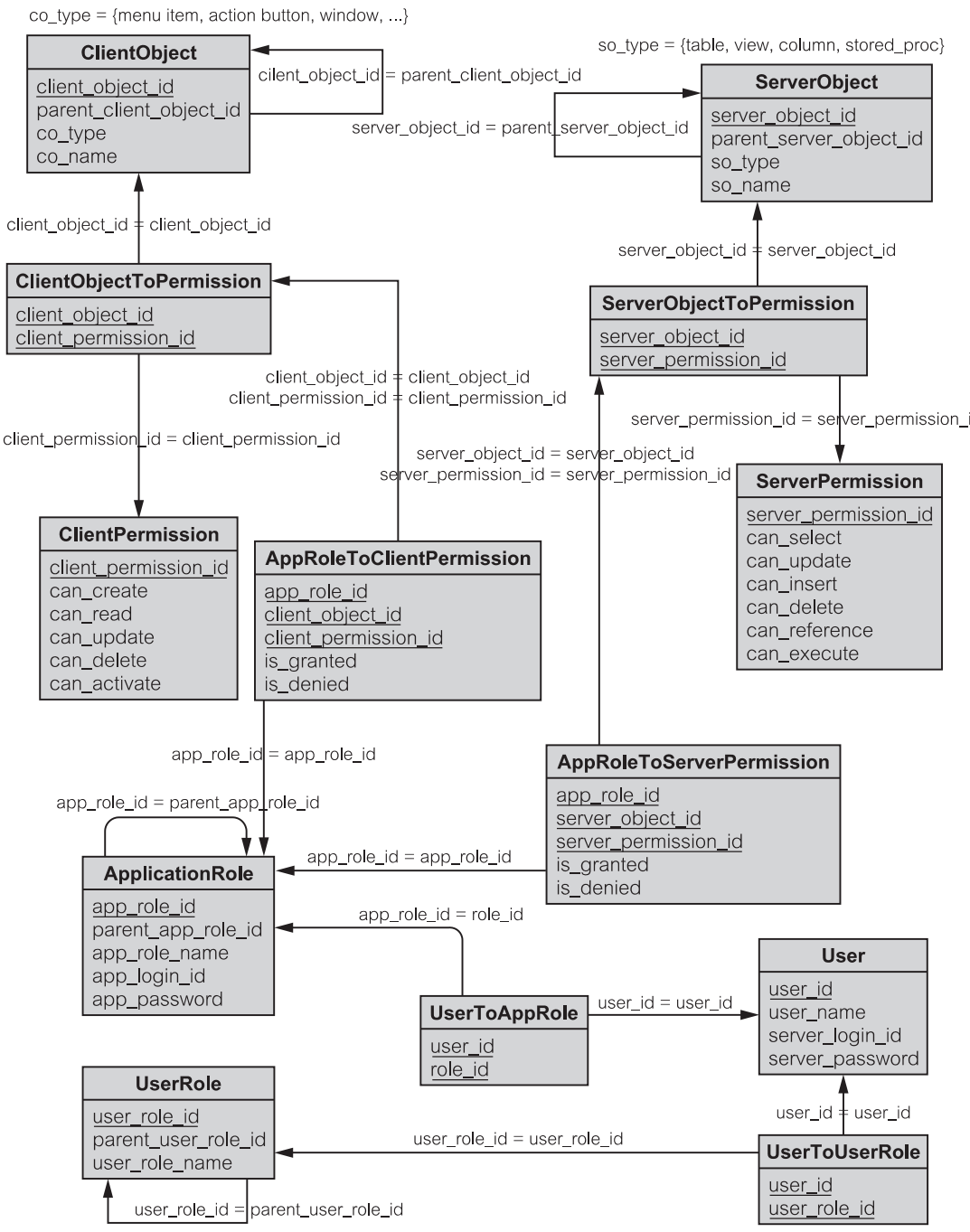


Рис. 20.4. Схема БД в поддержку авторизации предприятия

рархии меню). **Полномочия клиента** в `ClientPermission` (разрешение клиента) относятся к окнам и элементам управления окон.

Каждый объект клиента может предоставлять или запрещать больше чем одно разрешение. *Отмененные разрешения* не размещаются в БД авторизации — они не нужны для определения настроек авторизации текущего пользователя (отмененное разрешение — просто разрешение, которое не было предоставлено).

Допустимые преобразования разрешений в объекты клиента размещены в `ClientObjectToPermission` (объект клиента — в разрешение). Затем прикладная роль назначается ее разрешениям по отношению к объектам клиента приложения. Назначение может быть или тем, что разрешение *предоставляется*, или тем, что оно *запрещается*. Эта информация размещается в `RoleToClientPermission` — роль преобразовать в разрешение клиента (`is_granted` — разрешено и `is_denied` — запрещено).

Из-за возможных иерархических отношений между объектами клиента в БД авторизации реализованы сложные *правила целостности*, чтобы предотвратить несовместимости между предоставленными/запрещенными разрешениями. Например, не имеет смысла предоставлять разрешение на подменю, если разрешение на меню, которое содержит его, запрещено. Точно так же запрещенное разрешение для `can_read` (можно читать) устраняет предоставленное разрешение для `can_update` (можно корректировать).

В конечном счете, безопасность и целостность БД — обязанность самой БД, а не приложений клиента, обращающихся к БД. Авторизация клиентов может только устранить некоторые бреши в безопасности на ранних стадиях процесса и может освободить БД от дублированных и ненужных проверок. Не все потенциальные нарушения безопасности могут быть адресованы клиенту. Например, большинство ограничений целостности, реализованных в триггерах БД, которые могут быть вызваны лишь раз операцией авторизованного клиента, вызывающей добавление, удаление или корректировку, портят таблицу БД.

БД авторизации поддерживает в `ServerObject` — объекте сервера (рис. 20.4) — каталог всех *объектов сервера* для приложения, которые являются субъектами настроек авторизации. Каталог включает таблицы, представления, хранимые процедуры и любые индивидуальные столбцы таблиц, которые должны быть защищены от случайного доступа. Поддерживаются также иерархические отношения между объектами сервера (типа столбцов в таблицах). **Полномочия сервера** в `ServerPermission` (разрешение сервера) касаются объектов БД — таблиц, представлений, столбцов, хранимых процедур и т. д.

Как и у объектов клиента, каждый объект сервера может предоставлять или отменять больше чем одно разрешение. *Отмененные разрешения* не размещаются в БД авторизации.

Допустимые разрешения по отношению к объектам сервера размещаются в `ServerObjectToPermission` (объект сервера — в разрешение). Прикладная роль или предоставляет, или запрещает разрешения объектам сервера, с которыми роль связана через ссылочное отношение. Эта информация размещается в таблице по имени `RoleToServerPermission` — роль преоб-

разовать в разрешение сервера (`is_granted` — разрешена и `is_denied` — запрещена).

Перемещение бремени прикладной авторизации на сервер БД — хороший подход для двухуровневых систем клиент/сервер. Проблема становится более сложной для *систем, основанных на Web-технологии*, поддерживаемых Web-серверами и серверами приложения. Пользователи Web-приложений вряд ли будут иметь учетные записи в БД. Фактически БД может иметь только единственную учетную запись для всех пользователей Web-сервера и/или сервера приложения. В таких системах задачи авторизации перемещаются от сервера БД к Web-серверам/серверам приложения или даже к приложениям клиента. Проектирование авторизации для таких систем — мудреное дело с небольшой поддержкой ПО системы.

20.2. Проектирование целостности

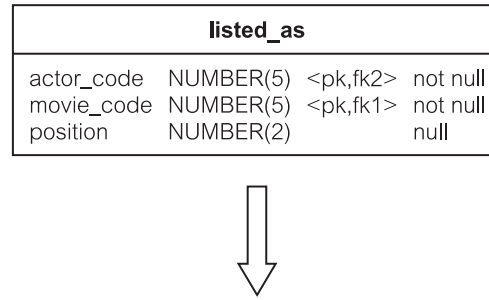
Термин «целостность» в ПО и инженерии данных имеет только слабое сходство со своим обычным значением правильности или единства. Целостность — это нечто вроде корректности и непротиворечивости данных, доступных для программы. Поскольку иногда различие между данными и процессом, который формирует данные, не может быть определено, целостность может также относиться и к процессам (операциям, программам и т. д.).

В контексте БД целостность определяется **ограничениями целостности**, предписанными поведением БД. Ограничения предписываются деловыми правилами, которым БД должна подчиняться. Они определяются в первую очередь в БД, нежели в приложении. Так как БД активны (программируемы), ограничения могут быть декларативными или процедурными (раздел 10.1.4). Типичные СУБД поддерживают следующие типы ограничения целостности [93]:

- null (пустой указатель);
- значение по умолчанию;
- домен (тип данных);
- проверка (правило, утверждение);
- уникальность (альтернативный ключ);
- первичный ключ;
- внешний ключ;
- триггер (активное правило, утверждение).

20.2.1. Null-ограничение и ограничение по умолчанию

Null-ограничение, определенное на столбце таблицы, утверждает, что столбец допускает пустые (`null`) значения. Имеются два главных смысла пустой величины, а именно «присутствует неизвестная величина» или «величина неподходящая» в контексте остальных значений в строке. Null-величина представляется специальным символом БД (то есть, это не нуль (0) или пустая строка).




```
create table "listed_as"  (
   "actor_code"          NUMBER (5)                        not null,
   "movie_code"          NUMBER (5)                        not null,
   "position"            NUMBER (2)                        default 1,
   constraint PK_LISTED_AS primary key ("actor_code", "movie_code")
)
/
```

Рис. 20.5. Null-ограничение и ограничение по умолчанию

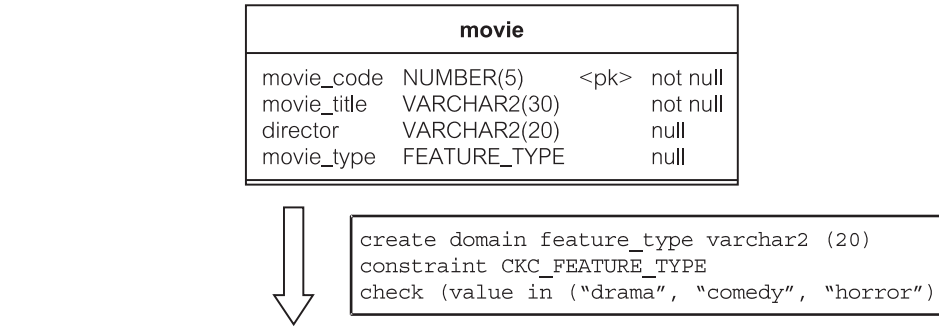
Ограничение по умолчанию, также определенное на столбце таблицы, обеспечивает значение для столбца, если такое значение не задается, когда вставляется в таблицу новая строка. Значение по умолчанию может быть литералом или выражением. Выражение может включать отобранные функции БД типа функции `sysdate` (системная дата) в Oracle, чтобы обратиться к текущей дате и времени. В отсутствие спецификации по умолчанию для столбца его значение по умолчанию устанавливается в `null` (если, конечно, столбец не определен как `not null` — не пустой указатель).

Рис. 20.5 показывает пример определения таблицы `listed_as` (перечисление). Столбцу `position` (положение) задается значение по умолчанию, равное 1. Это означает, что пока не будет задано другое значение, когда выполняется операция `insert` (добавить) для таблицы, актер будет рассматриваться как ведущий актер для кинофильма.

20.2.2. Ограничения «домен» и «проверка»

Домен представляет собой определенный пользователем тип данных. Определение включает возможность предписываемых ограничений (дополнительных данных) на величины, которые позволяет тип данных. Ограничения определяются путем использования ключевого слова **check** (проверка). Проверки могут иметь различную природу, типа допустимых диапазонов значений или разрешенных списков величин. После определения домен может использоваться как тип данных для любого столбца, вместо простых типов данных, обеспечиваемых в СУБД.

Рис. 20.6 показывает пример таблицы `movie` (кинофильм), расширенной столбцом `movie_type` (тип фильма), чей тип является доменом `feature_type` (тип особенности). Домен определен с помощью ограниче-




```
create table "movie"   (
   "movie_code"          NUMBER (5)                    not null,
   "movie_title"         VARCHAR2 (30)                 not null,
   "director"            VARCHAR2 (20),
   "movie_type"          VARCHAR2 (20)
         constraint CKC_MOVIE_TYPE_MOVIE check ("movie_type" is null or
                        ("movie_type" in ('drama', 'comedy', 'horror'))),
   constraint PK_MOVIE primary key ("movie_code")
)
/
```

Рис. 20.6. Ограничения домен и проверка

ния проверки, которое разрешает допустимые значения для типа — drama (драма), comedy (комедия) и horror (ужасы). Когда формируется оператор `create table` (создать таблицу) на основе проекта `movie` и `feature_type`, домен расширяется в определение столбца `movie_type`, чтобы соответствовать требуемому SQL-синтаксису.

20.2.3. Уникальный и первичный ключи

Строки (записи) в таблицах БД должны быть уникально идентифицированы, чтобы отделить одну строку от другой. Такая идентификация обеспечивается в реляционных БД посредством уникальных и первичных ключей. **Первичный ключ** — один из уникальных ключей, указанных для таблицы, который был выбран как более полезный для идентификации, чем другие. Это обычно некоторый код или число, и он обычно определяется на единственном столбце. **Уникальный ключ** также называется **потенциальным ключом** или **альтернативным ключом**.

Рис. 20.7 иллюстрирует таблицу `actor` (актер) с первичным ключом, определенным на столбце `actor_code` (код актера), и уникальным ключом, определенным на двух столбцах: `actor_name` (имя актера) и `address` (адрес).

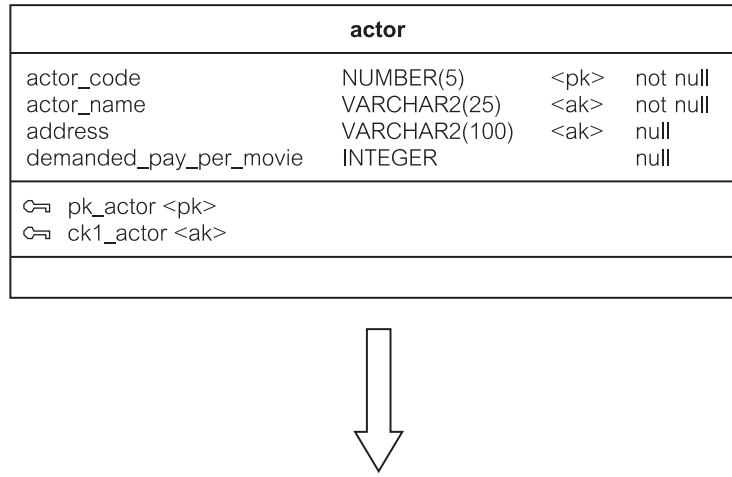
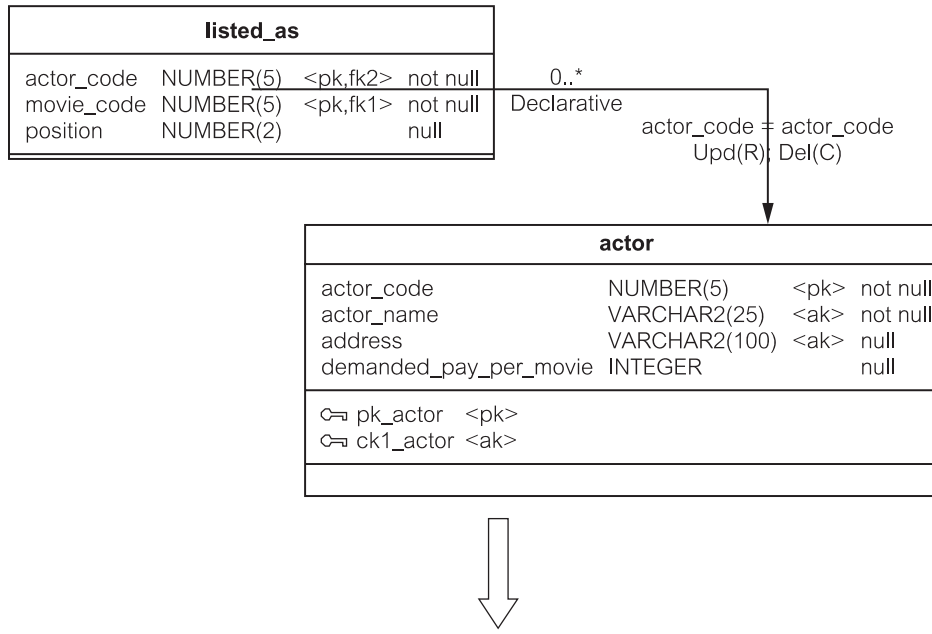


Рис. 20.7. Уникальный и первичный ключи

20.2.4. Внешние ключи

Как определено в разделе 10.1.2, **внешний ключ** представляет собой набор столбцов (часто только один столбец), значения которых соответствуют значениям первичного ключа в другой (или той же самой) таблице. Внешние ключи создают основу для ограничений **ссылочной целостности** (раздел 10.1.4). Ограничения могут быть реализованы декларативно или процедурно с помощью триггеров.

Рис. 20.8 является примером декларативного ограничения целостности, слегка измененного по сравнению с разделом 10.1.4. Ограничение определено на внешнем ключе `actor_code` (код актера) в таблице `listed_as` (перечисление). Это — **связывающее ограничение** для операций корректировки, заявляющее, что если `actor_code` в таблице `actor` (актер) модернизирован, все связанные зависимые записи в таблице `listed_as` должны быть соответственно модернизированы или корректировка будет запрещена. Связывающее ограничение — ограничение по умолчанию. Для операции удаления ограничение определяется как **каскадное**. Это означает, что удаление строки в таблице `actor` приведет к автоматическому удалению всех связанных зависимых строк в таблице `listed_as`.




```
create table "actor"  (
   "actor_code"         NUMBER(5)                    not null,
   "actor_name"         VARCHAR2(25)                 not null,
   "address"            VARCHAR2(100),
   "demanded_pay_per_movie" INTEGER                  default 50000,
   constraint PK_ACTOR primary key ("actor_code"),
   constraint CK1_ACTOR unique ("actor_name", "address")
)
/
create table "listed_as"  (
   "actor_code"          NUMBER(5)                   not null,
   "movie_code"          NUMBER(5)                   not null,
   "position"            NUMBER(2)                   default 1,
   constraint PK_LISTED_AS primary key ("actor_code", "movie_code")
)
/
alter table "listed_as"
   add constraint FK_LISTED_A_REFERENCE_ACTOR foreign key ("actor_code")
      references "actor" ("actor_code")
      on delete cascade
/
```

Рис. 20.8. Внешний ключ и декларативная ссылочная целостность

SQL-стандарт определяет два других декларативных ограничения: null-ограничение и ограничение по умолчанию. **Null-ограничение** означает, что когда ссылочные данные (в первичной таблице) будут скорректированы или удалены, все связанные зависимые данные (во внешней таблице) будут установлены в null. **Ограничение по умолчанию** означает, что когда упомянутые данные (в первичной таблице) будут скорректированы или удалены, всем связанным зависимым данным будут заданы значения по умолчанию.

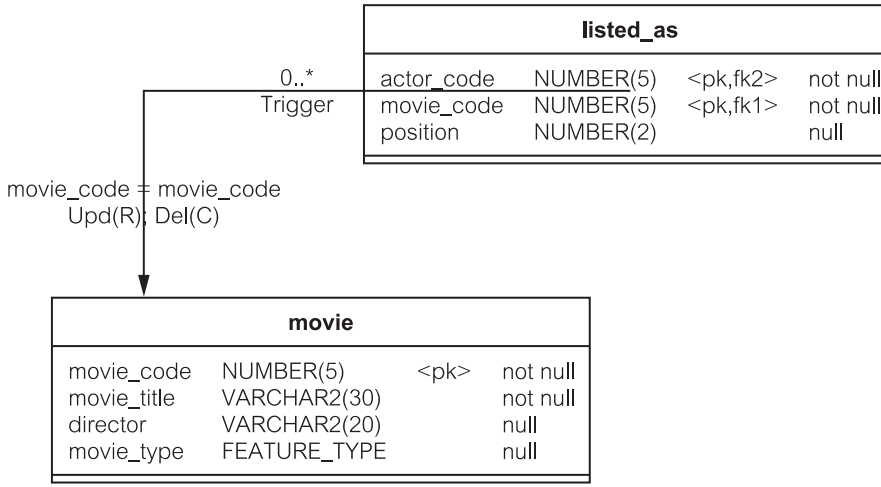


Рис. 20.9. Проект триггеров для таблиц movie и listed_as

20.2.5. Триггеры

Триггеры представляют процедурный способ задания ограничений ссылочной целостности наряду с реализацией других бизнес-правил, которым БД должна удовлетворять (раздел 10.1.4). Эти другие бизнес-правила, «объявленные» триггером, могут быть определены для любого столбца в таблице, а не только для столбцов внешнего ключа. Триггеры — варианты хранимых процедур, которые автоматически запускаются всякий раз, когда возникает предопределенное событие вставки, удаления и/или корректировки.

Рис. 20.9 является примером проекта триггеров для таблиц `movie` (кинофильм) и `listed_as` (перечисление). Листинг 20.5 содержит SQL-определение для этих триггеров в Oracle.

В случае Oracle (как показано в листинге 20.5) **действие триггера** может быть выполнено `before` (прежде), `after` (после) или `instead of` (вместо) оператора добавления, корректировки или удаления. **Триггеры «Прежде»** выполняют свои действия до того, как сработает событие. **Триггеры «После»** выполняют свои действия после того, как событие сработает.

Триггер «Вместо» является удобной технологией для выполнения действий, определенных в триггере, вместо действий, которые обычно обеспечивает выполнение события. Эта технология особенно полезна для сообщения системе, что делать, если осуществляется попытка вставить, скорректировать или удалить данные через представление. Так как многие представления не обновляемы, триггеры «Вместо» обеспечивают искусственное решение модификации представлений, которое иначе сделать было бы невозможно. В листинге 20.5 нет никакого примера триггера «Вместо».

Действия триггера можно запрограммировать, используя `:old` (старое) и `:new` (новое) значения записей, которые были добавлены, удалены или скорректированы, когда вызывались эти действия. Действия могут быть выполне-

ны или однажды `for each row` (для каждой строки), которая была изменена, или только однажды в случае выполнения оператора для всех измененных строк. Первый называется **триггером строки**, второй — **триггером оператора** (в листинге 20.5 нет примеров триггера оператора).

Одна из наиболее важных проблем, стоящих перед разработчиком, — это решить, использовать ли только *декларативную ссылочную целостность*, или заменить ее полностью *триггерами*. Триггеры, соответствующие декларативным ограничениям, могут быть автоматически сгенерированы CASE- и IDE-средствами, так что не потребуется никакого ручного программирования. На этот вопрос лучше ответить, изучив все «за» и «против» для используемой СУБД.

Листинг 20.5. Определение триггеров для movie и listed_as

Триггеры для movie и listed_as

```
-- Триггер "Перед добавлением" ""tib_listed_as"" для таблицы
-- ""listed_as"". Создание триггера "tib_listed_as" "перед
-- добавлением" в "listed_as" для каждой строки
declare
   integrity_error exception;
   errno           integer;
   errmsg          char(200);
   dummy           integer;
   found           boolean;
   -- Объявление ограничения InsertChildParentExist для предка
   -- ""movie""
      cursor cpk1_"listed_as" (var_"movie_code" number) is
         select 1
         from     "movie"
         where    "movie_code" = var_"movie_code"
         and      var_"movie_code" is not null;
begin
   -- Предок ""movie"" должен существовать при помещении
   -- потомка в ""listed_as""
   if :new."movie_code" is not null then
      open  cpk1_"listed_as"(:new."movie_code");
      fetch cpk1_"listed_as" into dummy;
      found := cpk1_"listed_as"%FOUND;
      close cpk1_"listed_as";
      if not found then
         errno  := -20002;
         errmsg := 'Parent does not exist in ""movie"".
                    Cannot create child in ""listed_as"".';
         raise integrity_error;
      end if;
   end if;
```

```
-- Управление ошибками
exception
   when integrity_error then
      raise_application_error(errno, errmsg);
end;
/

-- Триггер "Перед корректировкой" ""tub_listed_as"" для таблицы
-- ""listed_as"". Создание триггера "tub_listed_as" перед
-- корректировкой "actor_code", "movie_code" в "listed_as"
-- для каждой строки
declare
   integrity_error exception;
   errno           integer;
   errmsg          char(200);
   dummy           integer;
   found           boolean;
   -- Объявление ограничения UpdateChildParentExist для предка
   -- ""movie""
   cursor cpk1_"listed_as"(var_"movie_code" number) is
   select 1
   from   "movie"
   where  "movie_code" = var_"movie_code"
   and    var_"movie_code" is not null;
begin
   -- Предок ""movie"" должен существовать при корректировке
   -- потомка в ""listed_as""
   if (:new."movie_code" is not null) then
      open  cpk1_"listed_as"(:new."movie_code");
      fetch cpk1_"listed_as" into dummy;
      found := cpk1_"listed_as"%FOUND;
      close cpk1_"listed_as";
      if not found then
         errno  := -20003;
         errmsg := 'Parent does not exist in ""movie"".
                    Cannot update child in ""listed_as"".';
         raise integrity_error;
      end if;
   end if;

   -- Нельзя корректировать код предка ""movie""
   -- в потомке ""listed_as""
   if (updating('movie_code') and :old."movie_code" ! =
                                          :new."movie_code")
   then
      errno  := -20004;
      errmsg := 'Cannot modify parent code of ""movie""
                 in child ""listed_as"".';
```

```
      raise integrity_error;
   end if;
   -- Нельзя корректировать код предка ""actor""
   -- в потомке ""listed_as""
   if (updating('actor_code') and :old."actor_code" !=
                                              :new."actor_code")
   then
      errno  := -20004;
      errmsg := 'Cannot modify parent code of ""actor""
                                  in child ""listed_as"".';
      raise integrity_error;
   end if;
-- Управление ошибками
exception
   when integrity_error then
      raise_application_error(errno, errmsg);
end;
/

-- Триггер "Перед корректировкой" ""tub_movie"" для таблицы
                                                  ""movie""
create trigger "tub_movie" before update
of "movie_code"
on "movie" for each row
declare
   integrity_error exception;
   errno           integer;
   errmsg          char(200);
   dummy           integer;
   found           boolean;
   -- Объявление ограничения UpdateParentRestrict для
                                                   ""listed_as""
   cursor cfk1_"listed_as"(var_"movie_code" number) is
      select 1
      from   "listed_as"
      where  "movie_code" = var_"movie_code"
      and    var_"movie_code" is not null;
begin
   -- Нельзя корректировать код предка в ""movie"", если
   -- потомок еще существует в ""listed_as""
   if (updating('movie_code') and :old."movie_code" !=
                                              :new."movie_code")
   then
      open  cfk1_"listed_as"(:old."movie_code");
      fetch cfk1_"listed_as" into dummy;
      found := cfk1_"listed_as"%FOUND;
      close cfk1_"listed_as";
```

```
      if found then
         errno  := -20005;
         errmsg := 'Children still exist in ""listed_as"".
                    Cannot modify parent code in ""movie"".';
         raise integrity_error;
      end if;
   end if;
-- Управление ошибками
exception
   when integrity_error then
      raise_application_error(errno, errmsg);
end;
/

-- Триггер "После удаления" ""tda_movie"" для таблицы ""movie""
create trigger "tda_movie" after delete
on "movie" for each row
declare
   integrity_error exception;
   errno           integer;
   errmsg          char(200);
   dummy           integer;
   found           boolean;

begin
   -- Удаление всех потомков в ""listed_as""
   delete "listed_as"
   where "movie_code" = :old."movie_code";

-- Управление ошибками
exception
   when integrity_error then
      begin
      IntegrityPackage.InitNestLevel;
      raise_application_error(errno, errmsg);
      end;
end;
/
```

Во-первых, могут быть различия в выполнении функций (обычно в пользу декларативных ограничений). Во-вторых, смешивание декларативных ограничений и триггеров может привести к неприятному перекрытию и проблемам предшествования. Обычно декларативные ограничения предшествуют выполнению триггеров и могут, например, остановить выполнение нужного триггера. Однако триггеры **«Вместо»** имеют преимущество над декларативными ограничениями.

В-третьих, триггеры могут быть реализованы без какого-либо учета понятий первичных и внешних ключей. Даже если они и учитываются, отношение типа представленного на рис. 20.9 вряд ли будет автоматически перепроектировано из кода БД, если отношение реализовано посредством триггера.

Последнее, но не самое незначительное, заключается в том, что как только БД загружена с триггерами на свое место, последующая попытка переключиться на декларативную целостность (типа запуска оператора изменения таблицы для существующих таблиц) не может быть легко выполнена. Это означает, что БД должна быть удалена, создана новая схема и вся БД перезагружена. Во многих практических ситуациях создание БД заново и ее перезагрузка — невыполнимое требование.

20.3. Безопасность и целостность в управлении электронной почтой

Реализация итерации 3 ЕМ выполняется на основе рассмотрения управления безопасностью и целостностью. Главные изменения *авторизации* касаются добавления авторизации поверх контролируемой авторизации пользователей, уже реализованной в предыдущих итерациях. Изменения в обеспечении *целостности* главным образом касаются добавления триггеров.

Реализация авторизации предприятия в итерации 3 включает добавление ролей и введение специальной таблицы для хранения прав доступа. Изменения в правах доступа можно выполнить через новое окно авторизации в пользовательском интерфейсе. Эти изменения не столь обширны, чтобы реализовать их как отдельную БД авторизации для «вычисления» полномочий пользователя при запуске приложения и для соответствующей настройки приложения.

Триггеры, введенные в итерации 3, разработаны так, чтобы иметь возможность взаимодействовать с недавно добавленными хранимыми процедурами и хранимыми функциями. Триггеры определены для двух таблиц: `OutMessage` (исходящее сообщение) и `DailyReport` (ежедневный отчет).

20.3.1. Безопасность в управлении электронной почтой

Степень глубины детализации в отношении авторизации, поддерживаемой в БД, ограничена в основном предоставлением операций `select` (выбор), `update` (корректировка) и `delete` (удаление) на каждую таблицу. Это невозможно для декларативного предоставления пользователю *выбора* на *подмножестве* таблицы. Такая ситуация является только одной из причин, которая вынуждает использовать вариант программной контролируемой реализации для авторизации в итерации 3.

Права доступа в итерации 3 ЕМ относятся только к таблице `OutMessage` (исходящее сообщение). Пользователи могут лишь просматривать и воздействовать на подмножество этой таблицы в зависимости от отдела, в котором они работают. Возможный граф авторизации для итерации 3 ЕМ показан на рис. 20.10. Граф относится к виртуальным подмножествам таблицы `OutMessage`. Например, `IND_outmessages` (исходящие сообщения неза-

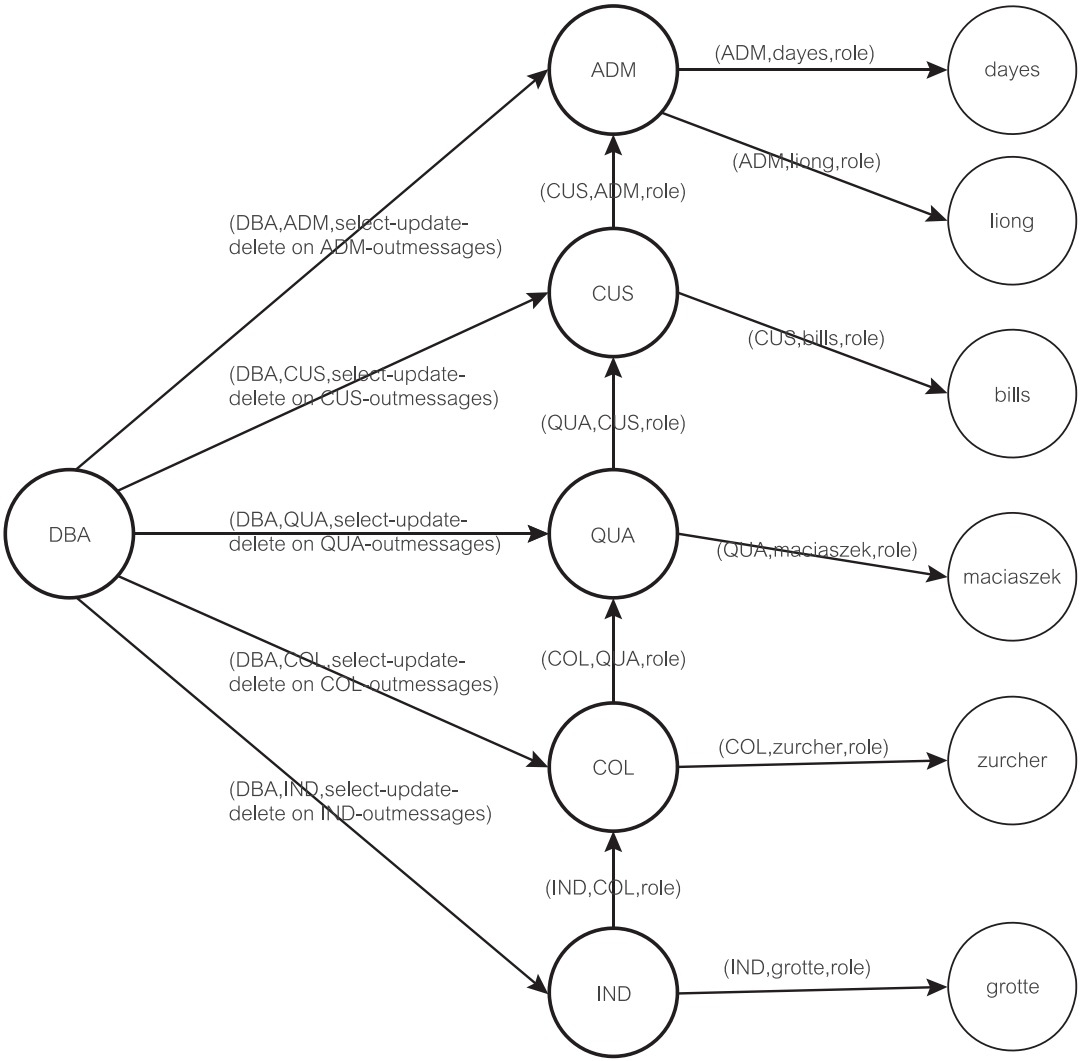


Рис. 20.10. Возможный граф авторизации итерации 3 EM

висимых служащих) представляют виртуальное подмножество таблицы `OutMessage`, доступное независимым служащим (Independent employees — `IND`). Подобный подход применяется и для других ролей авторизации.

Рис. 20.10 иллюстрирует, что иерархия полномочий по отношению к таблице `OutMessage` следующая — от `IND` к `COL`, далее к `QUA`, далее к `CUS` и, наконец, к `ADM` (расшифровка аббревиатур приведена в разделе 19.2.3). Наименьшее подмножество таблицы `OutMessage` относится к независимым служащим (`IND`). `Grotte` (Гротте) является независимым служащим и представляет `IND`-роль. Граф показывает, что элемент `IND` не может просматривать другие подмножества таблицы `OutMessage`, потому что `IND` не предоставлены никакие дополнительные полномочия.

Однако служащие отдела сбора данных (COL) могут управлять всеми исходящими сообщениями IND. Это происходит потому, что IND-роль предоставляется COL-роли. Кроме того, COL может просматривать большее подмножество таблицы OutMessage, как отражено в матрице авторизаций (раздел 19.2.3, таблица 19.1). Продвигаясь вверх по иерархии, очевидно, что наиболее мощная роль, которая имеет доступ и может изменять все исходящие сообщения, дана служащим отдела системного администрирования (ADM). Dayes (Дейес) и Liong (Лионг), как пользователи с предоставленной ADM-ролью, могут иметь доступ ко всему содержимому таблицы OutMessage.

SQL-операторы, соответствующие рис. 20.10, показаны в листинге 20.6. Лишь этот листинг мог бы использоваться, если бы имелись пять различных таблиц OutMessage. Деление существующей таблицы OutMessage на пять таблиц непрактично, особенно это неудобно, когда увеличение числа ролей потребовало бы даже большего количества таблиц. Кроме того, это не было бы приемлемо с точки зрения кода приложения. Заставить приложение выполнять многие операторы над подмножествами таблицы OutMessage, чтобы иметь доступ ко всем требуемым исходящим сообщениям, было бы кошмаром сопровождения и выполнения функций.

Листинг 20.6. SQL-операторы для реализации прав авторизации

SQL-операторы для реализации прав авторизации

```
create user Dayes identified by dayespsswd;
create user Liong identified by liongpsswd;
create user Bills identified by billspsswd;
create user Maciaszek identified by maciaszekpsswd;
create user Zurcher identified by zurcherpsswd;
create user Grotte identified by grottepsswd;
create role ADM;
create role CUS;
create role QUA;
create role COL;
create role IND;
grant create session to Dayes;
grant create session to Liong;
grant create session to Bills;
grant create session to Maciaszek;
grant create session to Zurcher;
grant create session to Grotte;
grant select on IND_OutMessage to IND;
grant select on COL_OutMessage to COL;
                                  --подобно и для update/delete
grant select on QUA_OutMessage to QUA;
                                  --подобно и для update/delete
grant select on CUS_OutMessage to CUS;
                                  --подобно и для update/delete
```

```
grant select on ADM_OutMessage to ADM;
                                    --подобно и для update/delete
grant IND to COL;
                --COL может теперь иметь доступ к IND_OutMessage
grant IND to QUA;  --QUA может иметь доступ к IND_OutMessage
grant IND to CUS;
grant IND to ADM;
grant COL to QUA;
                --QUA может теперь иметь доступ к COL_OutMessage
grant COL to CUS;
grant COL to ADM;
--и т.д., и т.д.
grant IND to Grotte ;
grant COL to Zurcher;
--и т.д., и т.д.
```

Чтобы решить эту проблему, итерация 3 использует программный подход к авторизации. Полномочия, предоставленные ролям, как показано на рис. 20.10, размещаются в таблице `Authorization` (авторизация). Эта таблица, используемая совместно со специально сформулированными представлениями, синонимами и хранимыми процедурами/функциями, может обеспечить желаемый уровень авторизации предприятия.

Обратите внимание, что в случае Oracle, помимо необходимости использования расширенных средств авторизации этого пакета, имеется дополнительная трудность задания каждой учетной записи на отдельной схеме. Это усложняет совместное использование глобальных ресурсов (глобальной схемы) в ситуациях, когда различные пользователи могут иметь доступ только к подмножеству этих ресурсов. Дублирование глобальных данных в локальных схемах не имеет смысла. Использование синонимов не защитило бы пользователей от доступа к запрещенным подмножествам данных. Представления являются запросами, ориентированными на множества, и в таком качестве требуют, чтобы полномочия на операцию `select` были предоставлены пользователю по отношению к исходной таблице. Это означает, что определенный пользователь был бы способен иметь доступ ко всему содержимому исходной таблицы.

Короче говоря, соответствующая обработка авторизации в EM вынуждает использовать программное решение, которое объединяет представления, синонимы, хранимые процедуры/функции и таблицу `Authorization`. Так как этот подход (представленный ниже) не является единственно возможным, другие возможные решения оставлены для упражнений.

Явно заданная таблица авторизации

Использование явно заданной таблицы `Authorization` упрощает процессы идентификации и авторизации. Пользователи идентифицируются в БД с помощью их локальной схемы. Глобальная схема выполняет определенную для приложения идентификацию через таблицу `Authentification` (опознание). Содержимое таблицы `Authorization` отражает матрицу авторизаций

(раздел 19.2.3, таблица 19.1). Каждая запись в таблице Authorization регистрирует роль (или отдел служащего) и одну (а возможно и несколько) *доступных* ролей (других отделов). Чтобы управлять доступом к исходящим сообщениям, необходимо реализовать все возможные доступы в хранимых процедурах/функциях. Хранимая процедура/функция может проверить, имеет ли текущий пользователь нужные полномочия. С этой целью таблица Authorization доступна для всех хранимых процедур/функций, реализованных в глобальной схеме. В качестве примера в листинге 20.7 таблица Authorization используется, чтобы сформировать подмножество таблицы OutMessage, которое может быть просмотрено текущим служащим (как известно, через его/ее empDept — отдел служащего).

Листинг 20.7. Использование таблицы Authorization

```
Использование таблицы Authorization
(чтобы сформировать подмножество OutMessage)

   Select *
   from "OutMessage"
   where "date_emailed" is null and "sched_dpt_code" in
      (select "target_dept_code"
      from "Authorization"
      where "from_dept_code" = empDept);
                                    --empDept ранее известен
```

Текущий служащий может просматривать только исходящие сообщения, адресованные ему/ей, а также те исходящие сообщения, которые позволяет ему/ей просматривать матрица авторизации (таблица 19.1). Используя этот подход, права доступа, показанные на рис. 20.10 и в листинге 20.6, можно реализовать без использования многих таблиц, чтобы хранить исходящие сообщения.

Таблица Authorization не обеспечивает механизм задания прав доступа к БД. Это просто записи ассоциаций между различными ролями, представленными отделами. Реальное осуществление прав доступа оставлено хранимым процедурам или SQL-операторам, подобным тем, которые представлены в листинге 20.7. Это рассмотрено более подробно в следующих двух разделах.

Использование индивидуальных схем, глобальной схемы и хранимых процедур

Пользователь БД на рис. 20.10 (каждый внутри тонкой окружности) имеет свою собственную **локальную схему**. Приложение клиента осуществляет доступ к БД через эту схему и поэтому знает только свои собственные данные. Он/она регистрируется в БД (или ЕМ-приложении в данном случае) через свои собственные пользовательское имя и пароль. Пользователь не знает, что БД фактически выполняет различные проверки его/ее полномочий.

Глобальная схема — это то, где расположены все совместно используемые данные. Ни одно из этих данных не скопировано или дублировано в локальных схемах. Пользователь не знает о существовании этой глобальной схемы. Глобальная схема содержит множество хранимых процедур, чтобы обеспечить своим *клиентам* (из локальных схем) доступ к своим данным. Эти хранимые процедуры/функции проверяют полномочия клиента. В зависимости от данного уровня авторизации клиента выполняются различные результаты и действия.

Пример этого подхода уже был приведен в листингах 20.2 и 20.3, которые показывают, как запрос к локальной схеме переадресован к хранимой функции в глобальной схеме. Переназначение включает скелет хранимой функции в локальной схеме. Глобальная схема выполняет проверку идентификации пользователя.

Другой ясный пример такой проверки показан в листинге 20.8. Идентификатор служащего проверяется по регистрационному имени пользователя, которое расположено в БД. Как только будет получен идентификатор служащего, извлекается код его отдела.

В приложении возникнет исключение, если пользователь введет неправильный параметр для хранимой функции и у него/нее не будут соответствующие регистрационное имя и код отдела. Уровень *безопасности* вышеупомянутого подхода увеличен по той причине, что приложение клиента не имеет непосредственного доступа к хранимой функции, показанной на рис. 20.8. Клиент обращается к глобальной хранимой функции через хранимую функцию его/ее собственной локальной схемы. Хранимая функция локальной схемы (типа показанной на рис. 20.4) гарантирует, что соответствующий параметр передан процедуре, приведенной на рис. 20.8.

В этом подходе индивидуальным пользователям не разрешается непосредственно выполнять операторы `select`, `update` и `delete` на глобальной схеме. Полномочия `select`, `update` и `delete` отменяются для всех пользователей, кроме пользователя глобальной схемы. Пользователям локальной схемы предоставляют только полномочия `execute` (выполнить), чтобы иметь возможность вызывать хранимые процедуры/функции глобальной схемы.

Использование индивидуальных схем, глобальной схемы, представлений и хранимых процедур

Разновидностью предыдущего подхода является дополнение локальной схемы **представлениями**. Несмотря на ограничения представлений (связанные с неспособностью корректировать данные через представления), они остаются удобным подходом для управления авторизацией.

Рассмотрим следующий сценарий. Приложение клиента соединяется с локальной схемой. Она выполняет оператор типа "`select * from uv_outmessages`". Приложение клиента не осознает, что `uv_outmessages` (представление исходящих сообщений) фактически является представлением. Представление `uv_outmessages` могло быть построено оператором в листинге 20.9. Первая часть оператора `select` выбирает все исходящие сообщения, которые были назначены текущему служаще-

му. Вторая часть оператора выбирает все исходящие сообщения, адресованные отделу служащего.

Листинг 20.8. Извлечение исходящего сообщения на основе авторизации служащего

Извлечение сообщения на основе авторизации служащего

```
create or replace function usp_retrieve_outmessage(usern in
                                                    varchar2)
return EMS.ref_cursor is
   c EMS.REF^CURSOR;
   inEmpID varchar2(100);
   empDept varchar2(100);
begin
   inEmpID := null;
   select "employee_id" into inEmpID
   from "Employee"
   where upper("login_name") = upper(usern);

   empDept := 'IND'; --независимый служащий по умолчанию
   select "department_code" into empDept
   from "Employee"
   where "employee_id" = inEmpID;

   open c for
      Select *
      from "OutMessage"
      where "date_emailed" is null and "sched_dpt_code" in
         (select "target_dept_code"
            from "Authorization"
            where "from_dept_code" = empDept);
      return c;
      EXCEPTION
         WHEN NO_DATA_FOUND THEN
            raise_application_error(-20009,
                'usp_retrieve_outmessage:
                unmatch username and loginname or you cannot
                view the message');
end;
```

Механизм авторизации, представленный в разделе 20.3.1, может использовать преимущества представлений. Представления могут использоваться, чтобы облегчить работу хранимых процедур/функций в отношении критериев выбора. Добавления, корректировки и удаления выполняются через хранимые процедуры/функции, игнорируя таким образом недостатки представлений.

Рассмотрим листинг 20.10, который показывает использование локальной схемы представления uv_outmessages. Представление используется, чтобы проверить, может ли служащий работать с сообщением, которое определяется идентификатором сообщения (msgID). Если исходящее сообщение имеется в представлении, то пользователь должен иметь доступ к нему и, следовательно, он/она должен иметь возможность удалить это исходящее сообщение. Если пользователь не имеет никаких полномочий удалить исходящее сообщение (то есть, select into dummy — выделить в макет — не дает результата), то в приложении клиента возникает исключение, чтобы указать на это.

Листинг 20.9. Представление uv_outmessages

```
Представление uv_outmessages

Create or replace view uv_outmessages as
select *
from PSE3."OutMessage"
where "date_emailed" is null and
("sender_emp_id" =
   (select "employee_id" from PSE3."Employee" emp, user_users u
      where upper(u.username) = upper(emp."login_name"))
or
   "sched_dpt_code" in
   (select "target_dept_code"
   from PSE3."Authorization"
   where "from_dept_code" = (select "department_code" from
PSE3."Employee" e, user_users u where upper(e."login_name")
= upper(u.username))
   )
)
```

Листинг 20.10. Удаление исходящего сообщения после проверки авторизации через представление

```
Удаление исходящего сообщения после проверки авторизации
через представление

create or replace procedure usp_delete_outmessage(msgID IN
                                  varchar2) is dummy integer;
begin
   select distinct 1 into dummy
   from uv_outmessages
   where msgID = "message_id";
```

```
    delete from PSE3."OutMessage" where "message_id" = msgID;
    --это удаление может быть заменено вызовом хранимой процедуры
    --PSE3.ups_delete_outmessage(msgID, usern);
    --где usern получен ранее, как показано в листинге 20.4.
    EXCEPTION
       WHEN NO_DATA_FOUND THEN
         raise_application_error(-20300, 'You are not allowed to
                                  delete the message');
end;
```

Удаление исходящего сообщения может быть просто выполнено так, как показано в операторе `delete` листинга 20.10. Лучший подход будет состоять в том, чтобы вызвать хранимую процедуру в глобальной схеме. Разрешение прямого выполнения операторов удаления требует, чтобы таблица `OutMessage` являлась выбираемой, обновляемой и удаляемой из локальной схемы. Фактически, это — слабость представления. Чтобы иметь возможность создать представление в локальной схеме, локальная схема должна иметь соответствующие полномочия (минимум полномочия `select`) по отношению к таблицам, включенным в представление. Это делает глобальную схему доступной для использования пользователям индивидуальной схемы.

Администрирование авторизации

Таблица `Authorization` позволяет выполнять динамическое формирование прав доступа (раздел 19.2.3, таблица 19.1) приложениями, когда они выполняются. Изменения прав доступа, когда-то зарегистрированных в таблице `Authorization`, могут автоматически формироваться запущенным приложением. Точно так же нет никакой необходимости корректировать локальную схему из-за изменений в правах доступа. Представления, как показано в листинге 20.9, также не подвергаются модификации.

Итерация 3 EM требует, чтобы пользовательский интерфейс позволял осуществлять модификацию прав доступа. Кроме того, только служащим отдела системного администрирования (`ADM`) разрешено модифицировать права. Это было реализовано с помощью набора хранимых процедур и GUI, показанных на рис. 20.11. Рис. 20.11 удовлетворяет требованию S9 (раздел 19.2.3, рис. 19.10). Вместо входов true/false рис. 20.11 использует выключатели.

Окно, изображенное на рис. 20.11, может быть редактируемо только `ADM`-пользователями. Это достигается разрешением/запрещением выключателей и разрешением/запрещением кнопки `Save` (сохранить). Если пользователю не позволено изменять матрицу авторизации, то выключатели не редактируемы и кнопка `Save` недоступна.

Хранимая процедура, выполняющая редактирование матрицы авторизации, показана в листинге 20.11. Процедура проверяет, является ли пользователь служащим `ADM`-отдела. Если это не так, возникает исключение, указывающее, что пользователю не позволено изменять права доступа (ошибки –20040 и –20041).

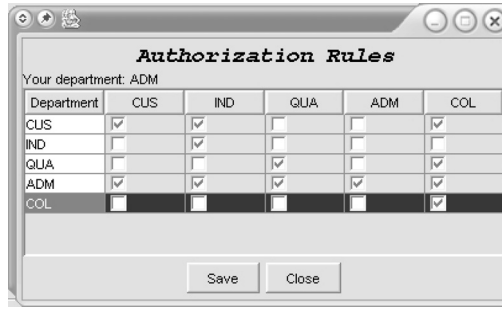


Рис. 20.11. GUI для модификации авторизации

Листинг 20.11. Хранимая процедура usp_upd_auth_rules

Хранимая процедура usp_upd_auth_rules

```
create or replace procedure usp_upd_auth_rules(frm in varchar2,
   target in varchar2, usern in varchar2) is
   inEmpID varchar2(100);

begin
   inEmpID := null;
   select "employee_id" into inEmpID
   from "Employee"
   where upper("login_name") = upper(usern) and
      "department_code" = 'ADM';
   if inEmpID is null then
      raise_application_error(-20040, 'usp_upd_auth_rules: '||
         'You are not allowed to modify authorization rules');
   end if;

  --это приведет к исключению, если вход уже существует. Поэтому
  --будьте уверены, что usp_clear_auth_rules вызывается первой
   insert into "Authorization"("from_dept_code",
                                        "target_dept_code")
      values(frm,target);
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
      raise_application_error(-20041, 'usp_upd_auth_rules: '||
         'You are not allowed to modify authorization rules');
      WHEN DUP_VAL_ON_INDEX THEN
         --вход уже существует; это означает, что
usp_clear_auth_rules
         --не была вызвана правильно
        raise_application_error(-20042, 'usp_upd_auth_rules:'||
            ' Duplicate entries found in authorization table');
end;
```

Как только пользователь будет проверен, параметры процедуры передаются оператору `insert`, чтобы добавить соответствующее правило авторизации. Добавление может вызывать исключение `DUP_VAL_ON_INDEX` (несколько строк с одним и тем же первичным ключом), если приложение клиента не выполнило предварительно очистку таблицы авторизации (если такая очистка требовалась).

20.3.2. Целостность управления электронной почтой

БД ЕМ поддерживает целостность своих данных через **декларативные ограничения целостности**, а также с помощью **триггеров**. Декларативные ограничения включают задание пустых указателей, доменов, правил проверки, ключей (первичных, внешних и уникальных), а также *ссылочные ограничения* от внешних ключей таблиц, соответствующих первичным ключам других таблиц. Бизнес-правила, которые не могут быть представлены декларативно, реализуются в триггерах.

Листинг 20.12 представляет декларативные ограничения целостности в отношении таблицы `DailyReport` (ежедневный отчет). Все столбцы объявлены как `not null` (не допускают пустых значений). Это гарантирует, что все данные записи (строки) должны быть заданы при добавлении в таблицу. Если необходимо, задается строгая длина столбца, как, например, для столбца `"contact_id"` (идентификатор делового партнера). Каждое значение в этом столбце должно иметь точно пять символов.

Листинг 20.12. Определение таблицы DailyReport

Определение таблицы DailyReport

```
 1:   create table "DailyReport" (
 2:      "date_of_report_day"          DATE       not null,
 3:      "contact_id"                  CHAR(5)    not null,
 4:      "num_emails_created"          INTEGER    not null,
 5:      "num_emails_sent"             INTEGER    not null,
 6:      "num_emails_outstanding"      INTEGER    not null,
 7:      "num_emails_past_outstanding" INTEGER    not null,
 8:      constraint PK_DAILYREPORT primary key
            ("date_of_report_day", "contact_id")
 9:   )
10:   alter table "DailyReport"
11:    add constraint FK_DAILYREP_CONTACT_R_CONTACT foreign key
12:          ("contact_id") references "Contact" ("contact_id")
```

Все таблицы в ЕМ имеют *первичный ключ*, чтобы гарантировать, что каждая конкретная запись в таблице будет уникальна относительно других записей. Дублированные вхождения не будут приняты. Таблица `DailyReport` имеет первичный ключ, состоящий из двух столбцов: `"date_of_report_day"` — дата дня отчета и `"contact_id"` — идентификатор делового партнера (строка 8).

Листинг 20.13. Правила проверки для num_emails_created

Правила проверки для num_emails_created

```
 1:   create table "DailyReport" (
 2:      "date_of_report_day"          DATE          not null,
 3:      "contact_id"                  CHAR(5)       not null,
 4:      "num_emails_created"          INTEGER       not null,
 5:      "num_emails_sent"             INTEGER       not null,
 6:      "num_emails_outstanding"      INTEGER       not null,
 7:      "num_emails_past_outstanding" INTEGER       not null,
 8:      constraint PK_DAILYREPORT primary key
            ("date_of_report_day", "contact_id"),
 9:      constraint num_created_greater0 CHECK
            ("num_emails_created" >= 0)
10:   )
```

Значение "contact_id" в каждой записи DailyReport должно быть равно одной из величин "contact_id" в таблице "Contact" (деловой партнер). Это определяет *декларативное ссылочное ограничение* между таблицами "DailyReport" и "Contact". Любая попытка ввести запись ежедневного отчета без надлежащей ссылки на *существующего* делового партнера будет отклонена в соответствии с этим правилом.

Декларативная целостность может гарантировать общие ссылочные ограничения. Она может также гарантировать, что значения столбца соответствуют определенному формату или попадают в пределы заданных диапазонов величин. Это может быть достигнуто с помощью *правил проверки*. Листинг 20.13 показывает, как правило проверки в таблице DailyReport определяет, что число созданных сообщений электронной почты не может быть отрицательным (строка 9). Это весьма очевидное правило, но оно может быть помещено для использования, потому что таблица не знает, как приложения собираются добавлять значения.

В EM имеются ограничения ссылочной целостности, которые не могут быть заданы с помощью использования декларативной целостности. Такие ограничения задаются *триггерами*. Ограничение в листинге 20.14 состоит из трех правил:

1. Менеджер служащего (в столбце manager_employee_id) может быть null, чтобы указать, что отдельный служащий может никем не управляться.
2. Менеджер должен принадлежать тому же самому отделу, что и служащий, если служащий имеет менеджера.
3. Служащий, который описан как независимый (IND), не ограничен предыдущим правилом (то есть он/она может управляться служащим из любого отдела) при условии, что он/она вообще имеет менеджера.

Листинг 20.14. Ограничение ссылочной целостности, заданное триггером

Ограничение ссылочной целостности, заданное триггером

```
 1:   create or replace trigger bins_upd_employee
 2:   before insert or update
 3:   on "Employee"
 4:   for each row
 5:   declare
 6:      dept_code varchar2(20);
 7:   begin
 8:      if :new."manager_employee_id" is not null then
 9:        if :new."department_code" <> 'IND' then
10:           select "department_code" into dept_code
11:               from "Employee"
12:           where "employee_id" = :new."manager_employee_id";
13:
14:              if :new."department_code" <> dept_code then
15:              raise_application_error(-20102,
                  'The manager cannot be from other department');
16:            end if;
17:         end if;
18:      end if;
19:   end;
```

Триггер, называемый `bins_upd_employee` (ввод корректировки информации о служащем), запускается в случае добавлений и корректировок таблицы `"Employee"` — служащий (строка 2). Конкретно, он запускается при *каждом* добавлении или корректировке строки таблицы (строка 4).

Строка 8 проверяет первое правило. Нет необходимости проверять добавление или корректировку входной информации таблицы, если новая строка (обозначенная как `:new` — новая) не имеет никакой информации о менеджере.

Строка 9 проверяет, является ли данный служащий независимым служащим (указанным принадлежащим к `IND`-отделу); в этом случае нет никакой необходимости далее проверять задание его/ее менеджера. Это удовлетворяет третьему правилу.

Оставшиеся строки 10–17 обеспечивают второе правило. `Department_code` (код отдела) менеджера извлекается и располагается в локальной переменной `dept_code` — код отдела (строки 10–12). Строка 14 проверяет, соответствует ли эта величина `dept_code` коду отдела служащего (`:new."department_code"`). Если она не соответствует отделу служащего, возникает ошибка приложения, указывающая на нарушение второго правила.

Когда возникает ошибка приложения, говорят, что триггер формирует событие (операцию) *нарушения* целостности, и поэтому основное событие должно быть отменено. Это означает, что операция добавления или корректи-

ровки вообще не будет выполняться (строка 2). Если триггер запускается *после* добавления или корректировки вместо выполнения *перед* этим (строка 2 указывает, что триггер запускается *перед* добавлением/корректировкой), то исключение, возникшее в триггере, будет завершено откатом такого добавления/корректировки.

Запуск триггера прежде или после операции существенно отличается, хотя различие может быть незаметным большую часть времени. Если добавление допустимо перед запуском триггера (**триггер «После»**), то могут быть другие триггеры, которые должны быть запущены перед текущим запущенным триггером. Рассмотрим случай, когда таблица имеет **триггер «Прежде»** наряду с триггером *«После»*. Добавление инициализирует триггер *«Прежде»*, после этого величина добавляется и запускается триггер *«После»*. Отказ в триггере *«После»* будет означать откат действия вставки так же, как и эффект любой работы, выполненной триггером *«Прежде»*. Не любая работа, выполняемая в БД, повторяема или восстанавливаема, и этому нужно уделять серьезное внимание. Было бы очень хорошо, если бы триггер *«Прежде»* выполнял операцию, которая может быть отменена, или отмена операции не приводила бы к несовместимости в БД.

Резюме

1. *Безопасность* является сугубо персональной (индивидуальной) характеристикой. *Целостность* имеет групповой характер (используется группой пользователей).
2. *Идентификация* является задачей подтверждения допустимости пользователю/приложению соединиться с БД.
3. *Контролируемая авторизация* основана на предоставлении *полномочий* доступа к объектам БД. Полномочия могут быть предоставлены/отменены пользователям и/или ролям. *Роль* обычно относится к рабочим функциям. *Пользователь* может быть назначен на одну или несколько ролей.
4. *Системные полномочия* — право выполнить конкретное действие на системе БД в целом или действие в отношении схемы БД. *Объектные полномочия* определяют, как пользователи/приложения могут работать с содержимым БД.
5. Полномочия могут быть переданы от пользователя/роли к другому пользователю/роли. Эффекты передачи полномочий могут быть изображены в виде *графа авторизации*.
6. Системные и объектные полномочия представляют SQL-*декларативные* пути реализации контролируемой авторизации. SQL обеспечивает также *программные* (процедурные) пути управления контролируемой авторизацией. Программная авторизация использует представления, синонимы и хранимые процедуры.
7. При *принудительной авторизации* пользователям/приложениям назначаются *уровни полномочий*, а объектам БД назначаются соответствующие *уровни безопасности*. Уровни полномочий используются при идентификации пользователя/приложения. Идентификация проверяет учетную за-

пись пользователя. Для более безопасных сценариев может использоваться *подход «запрос-ответ»*, использующий *шифрование*.

8. *Авторизация предприятия* требует, чтобы права доступа всех пользователей и приложений были размещены в БД *авторизации предприятия*, которую каждое приложение должно просматривать, чтобы определить, что текущий пользователь может или не может делать в отношении объектов БД и в отношении объектов приложения. В авторизации предприятия *прикладные роли* отличаются от *ролей пользователей*.
9. *Целостность* означает корректность и непротиворечивость данных, доступных программе. В контексте БД целостность определяется *ограничениями целостности*, заданными в отношении поведения БД.
10. *Триггеры* представляют процедурный путь обеспечения ограничений ссылочной целостности, они также реализуют другие бизнес-правила, которым должна удовлетворять БД.
11. Итерация 3 реализует *авторизацию предприятия*. Она включает добавление ролей и специальной таблицы, содержащей права доступа. Изменения в правах доступа могут быть выполнены через новое окно авторизации в пользовательском интерфейсе. Весь доступ к БД выполняется через хранимые процедуры/функции, которые также отвечают за обеспечение прав доступа.
12. Итерация 3 обеспечивает *целостность* БД различными средствами, включая использование триггеров. *Триггеры* разработаны так, чтобы взаимодействовать с вновь введенными хранимыми процедурами и хранимыми функциями.

Ключевые термины

Обзорные вопросы

1. Объясните различия между безопасностью и целостностью системы.
2. Объясните различия между идентификацией и авторизацией.
3. Представим пользователя приложения, которому разрешено только искать информацию в БД, но не делать никаких изменений в данных. Нужно/можно ли такому пользователю предоставить какие-либо системные полномочия? Если да, то какие это системные полномочия? Приведите примеры.
4. Могут ли быть роли идентифицированы? Каковы будут последствия?
5. Каково различие между отмененными полномочиями и запрещенными полномочиями? Поясните примером.
6. Почему представления обеспечивают только ограниченный механизм безопасности?
7. Как принудительная авторизация соотносится с контролируемой авторизацией? Являются они различными или перекрывающимися технологиями?
8. Объясните различие между прикладной ролью и ролью пользователя в авторизации предприятия. Как эти две концепции связаны с понятием авторизации БД?
9. Каковы декларативные и процедурные ограничения целостности? Перечислите и кратко определите их.
10. На рис. 20.7 уникальный ключ таблицы `actor` (актер) определен на столбцах `actor_name` (имя актера) и `address` (адрес). Однако столбец `address` допускает пустые значения. Допустимо это? Если это так, к чему приведет попытка добавить `actor_name` с пустым `address`, если такой `actor_name` с пустым адресом уже существует в таблице? Объясните.

11. Триггер — своего рода хранимая процедура. Может ли хранимая процедура вызвать триггер? Объясните.
12. Как подход Oracle к назначению локальной схемы каждой учетной записи влияет на проект авторизации в приложении? Полезен этот подход Oracle или нет? Объясните.
13. Приведите пример транзакции, когда действия триггеров или хранимых процедур нельзя откатить назад, несмотря на то, что транзакция потерпела неудачу.

Примеры задач

1. Обратимся к разделу 20.3.1, листинг 20.9. Представление, сформированное этим листингом, приведет к двойным входам. Почему? Проверьте это, экспериментируя в БД Oracle.
2. Обратимся к разделу 20.3.1 и рис. 20.10. Измените граф авторизации, показанный на рис. 20.10, со следующим дополнительным требованием: «QUA-служащие могут читать ADM-сообщения, но они не могут изменять их. QUA-служащие не имеют доступа к CUS-сообщениям.»
3. Реализуйте триггер `before` («Прежде») на таблице `ListedAs` (перечисление) БД `MovieActor` («Фильм-Актер») как замену для ее эквивалентного декларативного ограничения ссылочной целостности следующим образом:

```
ADD Constraint fk_const3 FOREIGN KEY (actor_code)
REFERENCES actor ON DELETE CASCADE
```

Глава

21 Транзакции и параллелизм

Транзакции были неоднократно неформально упомянуты в этой книге, но они не были рассмотрены как самостоятельная тема. Практически разработка транзакций в информационных системах предприятия — одна из больших проблем для инженера ПО. БД предлагают широкую поддержку для управления транзакциями, но только разработчики приложения могут определить границы транзакций и решить, какие механизмы транзакций должны использоваться для каждой из них.

Транзакция определяется как логическая единица работы, которая выполняет отдельную бизнес-задачу и гарантирует *целостность* БД после того, как задача завершается (глава 20). В этом определении термин «транзакция» является сокращением для термина «**бизнес-транзакция**». С точки зрения транзакции **целостность** означает, что данные остаются непротиворечивыми после того, как транзакция закончит свое выполнение.

Транзакция состоит из одного или большего количества операций манипуляции данными. Классический пример транзакции — перемещение денег между двумя счетами, типа оплаты счета кредитной карточкой со сберегательного счета. Оно включает извлечение денег со сберегательного счета и внесение их на счет кредитной карточки. Оба действия размещаются в одной транзакции.

Группировка действий, формирующих транзакцию, является бизнес-решением, а не техническим. Однако бизнес-операция может состоять из ряда более коротких **системных транзакций**. Действия транзакции выполняются в унисон — они или успешно завершаются и записывают (**фиксация**) все изменения данных в БД, или терпят неудачу и отменяют (**откат**) все частичные изменения в БД.

В бизнес-приложениях выполнение транзакций в пределах приложения должно быть настолько коротко, чтобы многие приложения и пользователи, работающие с той же самой БД, могли иметь параллельный доступ к совместно используемым данным (то есть транзакции не блокируют друг друга, захватывая общие ресурсы данных в течение длительного времени). **Короткие транзакции** облегчают **параллелизм**.

Имеются приложения, где короткие транзакции не могут составлять основные границы транзакций. Например, системы групповых вычислений, типа систем, облегчающих совместную работу служащих офиса, архитекторов, инженеров и т. д., требуют **длинных транзакций**. Служащему офиса,

пишущему документ или работающему с данными электронной таблицы (извлекаемыми из БД), нужно позволить выполнять эти задачи в течение более длительного времени и иметь возможность передавать изменения в БД без блокирования других пользователей и без нарушения целостности БД. Такие системы требуют различных механизмов параллелизма, которые большинство традиционных (реляционных) СУБД достаточно хорошо не поддерживают.

Проблема длинных транзакций — очевидная, но не единственная причина делать различия между бизнес-транзакциями и системными транзакциями. **Системная транзакция** (или **транзакция БД**) определяется механизмами поддержки, обеспечиваемыми представлением транзакций в СУБД или в другой системе управления транзакциями. Системная транзакция — техническая концепция, которая не имеет понятия о бизнес-причинах транзакции как единицы работы.

Бизнес-транзакция (или **транзакция приложения**) является бизнес-концепцией; она имеет деловой смысл для пользователя приложения. Клиент, использующий Web-приложение, чтобы купить книги по Интернету, определяет бизнес-транзакции. Действия помещения книг в корзину заказа через определение деталей оплаты и передачи (или отмены) заказа составляют единую бизнес-транзакцию. Из сказанного следует, что бизнес-транзакция, вероятно, охватит много системных транзакций. Это доставляет серьезную проблему инженеру ПО, который должен обратиться к некоторым хитрым проблемам параллелизма за пределами встроенных механизмов поддержки СУБД или другого монитора транзакций.

Кроме параллелизма управление транзакциями неотделимо от проблем **восстановления** БД. СУБД гарантирует, что любые отказы транзакций не оставят БД в противоречивом состоянии. Если транзакция терпит неудачу, БД восстанавливается к состоянию, в котором она была до начала транзакции. Механизмы восстановления расширяются на отказы, не связанные с транзакциями, типа физического повреждения диска, содержащего БД или данные транзакций.

Управление транзакциями для параллелизма — главная проблема для инженера ПО и разработчика. Управление транзакциями для восстановления главным образом является задачей администратора системы/БД. Следовательно, эта глава имеет дело с параллелизмом; ссылки на восстановление используются только в крайнем случае.

21.1. Параллелизм в системных транзакциях

Параллельное выполнение программ на совместно используемых источниках данных представляет одну из наиболее трудных проблем в программной инженерии. Это чрезвычайно сложная задача, в которой индивидуальные требования транзакций отдельного пользователя или приложения должны быть определены и реализованы в контексте всех других транзакций, которые могут одновременно выполняться в системе. К счастью, СУБД обеспечивают

простые (и не так уж простые) механизмы для освобождения инженеров ПО от необходимости обеспечивать целостность транзакций и от наиболее хитрых проблем реализации. *Системные транзакции* определяются в терминах этих примитивных механизмов.

Транзакция ограничена, явно или неявно, операторами, которые отмечают ее начало и конец. Эта операция установления границ транзакции называется **разграничением** или **заключением в скобки** транзакции. Разграничение системной транзакции известно как **разграничение на стороне сервера**, в то время как разграничение бизнес-транзакции называется **разграничением на стороне клиента**.

Начало системной транзакции часто задается неявно. Транзакция начинается с первым выполняемым SQL-оператором в группе SQL-операторов. Некоторый вид оператора `begin transaction` (начало транзакции) или `start transaction` (запуск транзакции) может использоваться для явного задания точки входа в транзакцию.

Конец транзакции может также быть неявным, но всегда лучше явно разграничить ее конец. Транзакция может заканчиваться оператором `commit` (фиксация) или оператором `rollback` (откат). *Фиксация* означает успешное завершение транзакции, которая приводит к записи в БД всех изменений транзакции. *Откат* обычно означает, что столкнулись с некоторой проблемой и транзакция прервана, а любые частичные изменения, сделанные в БД, будут удалены. В некоторых случаях откат может означать ожидаемое окончание транзакции (это случается, например, при тестировании программ БД и желании возвратить БД в ее первоначальное состояние после тестирования).

21.1.1. ACID-свойства

Тема транзакций и параллелизма имеет тесную связь с понятием целостности данных (раздел 20.2). Целостность данных определяет правила, обеспечивающие их корректность, точность и непротиворечивость. Для гарантии целостности данных в многопользовательской системе сервер БД должен реализовать механизмы параллелизма так, чтобы только один пользователь/приложение мог бы изменять конкретную часть данных одновременно, соблюдая при этом все правила целостности, которые предписаны данным.

Чтобы обеспечить целостность данных в присутствии параллельного доступа и отказов системы, системные транзакции должны удовлетворять четырем свойствам, известным в литературе как **ACID-свойства**. Аббревиатура обозначает atomicity (атомарность), consistency (непротиворечивость), isolation (изоляция) и durability (долговечность). Непротиворечивость и изоляция имеют главным образом отношение к параллелизму, в то время как атомарность и долговечность в большей мере связаны с восстановлением.

Атомарность задает требование, чтобы каждая транзакция представляла минимально возможную единицу, которая не может быть далее разделена на части. Атомарная транзакция гарантирует требование «все или ничего»; система должна или успешно закончить все ее действия, разграниченные транзакцией, или она должна откатить назад всю работу. Частичные завершения транзакций не допускаются. В примере транзакции, когда деньги передаются между двумя счетами, атомарность гарантирует, что деньги никогда не будут

взяты со сберегательного счета, если они не будут внесены на счет кредитной карточки. Любая такая неполная транзакция будет отменена (произойдет откат) и пользователю/приложению будет дан шанс повторить попытку.

Имеется много причин, почему транзакция может не закончиться. Это может быть ошибка сети или БД, конфликт (типа *тупиковой ситуации*), происходящий между одновременно выполняемыми транзакциями, или внезапный отказ прикладной программы. Имеется много частей программного/аппаратного обеспечения, которые участвуют в транзакции, от кода клиента, через Web-технологию и серверы приложения, до сервера БД. Любая из этих сторон может вызвать отказ. Поскольку никакая из этих сторон не знает то, что происходит в других частях, выполнение (или откат) транзакции использует схему голосования. Каждая сторона голосует за то, должна ли транзакция рассматриваться успешной, и если какой-либо голос отрицательный, вся транзакция должна откатиться назад. Две типичные схемы голосования называются *двухфазным завершением* и *трехфазным завершением*.

Долговечность дополняет свойство атомарности, утверждая, что успешная транзакция гарантирует ситуацию, когда выполненные изменения, помещенные в БД, являются постоянными (сохраняемыми). То есть сервер гарантирует, что любые отказы после того, как пользователь/приложение будут проинформированы, что произошли изменения, не приведут ни к какой потере данных. В действительности, сообщение пользователю, что совершены изменения, и само совершение изменений — два различных события, и отказ может даже произойти между этими двумя событиями. Во всех случаях сервер дает ту же самую гарантию пользователю.

Сервер гарантирует атомарность транзакции, **удаляя** дефектные транзакции, и он обеспечивает долговечность транзакции с помощью **повторного выполнения**, если это необходимо, успешных транзакций, которые в действительности не были сделаны сохраняемыми, в то время как приложение было информировано относительно успеха. Чтобы быть способным удалить и повторно выполнить транзакции, сервер БД обеспечивает регистрацию транзакций. *Регистрация* помнит (постоянно) все изменения, сделанные транзакциями в БД (все это записывается в БД). Сервер использует регистрацию, чтобы отменять и повторно выполнять транзакции, когда это требуется обстоятельствами.

Непротиворечивость представляет собой требование, чтобы каждая транзакция обеспечивала переход данных из одного непротиворечивого состояния в другое, сохраняя семантику данных и ссылочную целостность. Непротиворечивое состояние БД обеспечивается правилами целостности данных (раздел 20.2). Поскольку правила целостности являются бизнес-правилами, определяемыми пользователями, гарантия непротиворечивости транзакций, прежде всего, является обязанностью прикладных разработчиков и программистов. Поддержка непротиворечивости системы является гарантией, что транзакция будет терпеть неудачу, если она попытается нарушить указанные правила целостности.

Изоляция — требование, чтобы любые промежуточные изменения, сделанные в данных транзакцией, были невидимы другим параллельным транзакциям до операций фиксации. Выполнение транзакций изолировано так,

чтобы каждая транзакция формировала результаты в БД, которые не зависят каким-либо образом от других транзакций. Это называется **сериализуемым выполнением**, то есть, множество транзакций производит те же самые результаты независимо от того, выполняются ли транзакции одновременно или последовательно, одна за другой в произвольном порядке. Сервер БД гарантирует изоляцию, помещая **блокировки** на данные, в настоящее время используемые транзакцией. Когда одна транзакция содержит блокировку на данные, другие транзакции, заинтересованные в этих данных, должны ждать, пока эта блокировка не будет удалена.

21.1.2. Уровни изоляции

Сериализуемое выполнение транзакций в полной изоляции друг от друга требуется не всегда или является непрактичным. Например, одновременное выполнение транзакций, которые вычисляют некоторые статистические величины на большом объеме элементов данных, вряд ли будет заботиться относительно нестационарных изменений, сделанных другими транзакциями в одном или небольшом количестве этих элементов. Кроме того, полная изоляция транзакций в сериализуемом выполнении понижает возможный уровень параллелизма в системе и приводит к более низким производительности и качеству выполнения функций системы.

Контекст для компромиссов между корректностью выполнения, параллелизмом и выполнением функций обеспечивается четырьмя уровнями изоляции, определенными в SQL, Java и других стандартах. **Уровень изоляции** определяет, как параллельные транзакции изолированы друг от друга.

Самый высокий уровень изоляции, с самым плохим параллелизмом и выполнением функций, но со строгой (изолированной) корректностью результатов транзакций, является **сериализуемым** (serializable) уровнем. Выполнение транзакции на сериализуемом уровне защищено от чтения незавершенных изменений другими транзакциями. Этот уровень гарантирует также, что многократное чтение одних и тех же данных в пределах одной и той же транзакции будет всегда возвращать те же самые значения, даже если другая транзакция изменила эти данные (*непротиворечивость чтения*). Наконец, на сериализуемом уровне, при многократном выполнении одного и того же запроса, будут всегда получаться одни и те же результаты, даже если другая транзакция добавит новые записи данных, которые удовлетворяют условию запроса (*непротиворечивость запроса*). Сериализуемое выполнение достигается системой, помещающей блокировки на множество данных транзакции так, чтобы другие транзакции не могли корректировать или добавлять в это множество новые данные до завершения транзакции.

Следующий уровень за сериализуемым называется **повторяемым чтением** (repeatable read). Этот уровень гарантирует непротиворечивость чтения, но не гарантирует непротиворечивость запроса. Отсутствие гарантии непротиворечивости запроса означает, что повторяемое чтение разрешает возникновение фантомов. **Фантом** возникает, когда транзакция выполняет тот же самый запрос дважды и второе выполнение показывает дополнительные записи данных (фантомы), помещенные тем временем другими транзакциями. Повторя-

емое чтение достигается механизмом блокировки, который запрещает нестационарные корректировки, но позволяет нестационарные добавления.

Третий по очереди уровень, следующий за повторяемым, — это так называемое **чтение зафиксированных данных** (read committed). Этот уровень гарантирует один из двух аспектов непротиворечивости чтения. Он гарантирует невидимость любых элементов данных, измененных другой транзакцией при условии, что модификация не была еще зафиксирована. Однако он позволяет видеть зафиксированные изменения, которые могут привести к **неповторяемым чтениям**. Чтение фиксированных данных достигается помещением так называемых *совместно используемых блокировок* на данные, которые читаются транзакциями, обеспечивая так называемые *блокировки взаимоисключающего доступа*, установленные на данных другими транзакциями, чтобы модифицировать эти данные.

Низший уровень изоляции — **чтение незафиксированных данных** (read uncommitted). Этот уровень не гарантирует никакого аспекта непротиворечивости чтения. На этом уровне возможно **«грязное» чтение**, которое читает данные, измененные другой транзакцией, но фактически все еще не зафиксированные. На этом уровне не используются никакие совместно используемые блокировки и не задаются никакие блокировки взаимоисключающего доступа.

Таблица 21.1 суммирует вышеупомянутое обсуждение, показывая, какая несовместимость (запроса или чтения) может возникнуть на различных уровнях изоляции. Программист может устанавливать различные уровни изоляции для различных транзакций, выполняющихся одновременно, но система обычно отвергает изменение уровня в середине выполнения транзакции. Уровнем изоляции по умолчанию в большинстве систем является чтение зафиксированных данных.

Таблица 21.1. Поведение уровней изоляции

Уровень изоляции:	«Грязное» чтение	Неповторяемое чтение	Фантом
Сериализуемый	невозможно	невозможно	невозможно
Повторяемое чтение	невозможно	невозможно	возможно
Чтение зафиксированных данных	невозможно	возможно	возможно
Чтение незафиксированных данных	возможно	возможно	возможно

21.1.3. Способы блокировки и уровни блокировки

Уровни изоляции реализуются с помощью блокировки ресурсов данных. Лишь уровень чтения незафиксированных данных не использует никаких блокировок. Блокировки могут быть помещены на ресурсы данных и процессов. Системы БД предлагают разнообразные варианты способов блокировки и уровней блокировки, которые существенно различаются между собой.

Основные **способы блокировки** следующие [42, 61]:

- блокировки взаимно исключающего доступа (блокировки записи);
- блокировки корректировки (блокировки попыток записи);
- совместно используемые блокировки (блокировки чтения).

Блокировка взаимно исключающего доступа, помещенная в ресурс транзакцией, означает, что только эта транзакция может иметь доступ к ресурсу, как для записи, так и для чтения. Это наиболее строгий способ в смысле ограничения доступа к ресурсу. Пока транзакция, задавшая блокировку, не снимет ее, никакие другие блокировки не могут быть установлены на ресурсе. Блокировка взаимно исключающего доступа всегда автоматически помещается в ресурс системой на время фактического обновления ресурса.

Одна проблема с одновременно задаваемыми блокировками состоит в том, что транзакция может потребовать блокировок, выполненных другой транзакцией, и наоборот — вторая транзакция может потребовать блокировок, выполненных первой транзакцией. Это создает **тупиковую ситуацию**. Тупиковая ситуация может быть разрешена только путем отката назад одной из этих двух транзакций. Так как предотвращение лучше, чем обнаружение, используются блокировки корректировки, чтобы предотвратить многие тупиковые ситуации.

Блокировка корректировки, полученная ресурсом от транзакции, гарантирует, что транзакция будет способна корректировать блокировку в способ взаимно исключающего доступа, как только потребуется такая корректировка. Только одна транзакция может одновременно выполнять блокировку корректировки на ресурсе.

Совместно используемая блокировка, помещенная в ресурс транзакцией, не ограничивает другие транзакции в чтении того же самого ресурса или в получении блокировки корректировки на этом ресурсе. Для повторяемого чтения или сериализуемой изоляции совместно используемые блокировки поддерживаются до конца транзакции. Для уровня изоляции, обеспечивающего чтение зафиксированных данных, совместно используемые блокировки поддерживаются только на время чтения данных сервером (именно поэтому неповторяемые чтения могут возникнуть на этом уровне изоляции).

Блокировки могут быть выполнены на ресурсах различной степени детализации. Это называется **уровнем блокировки**. Три наиболее интересных уровня следующие:

- строка;
- страница;
- таблица.

Блокировка строки — самая низкая возможная блокировка в реляционных БД, где минимальный доступный объект — строка (запись) таблицы. **Блокировка страницы** охватывает все строки, которые размещаются на единой странице. Страница — физическая часть размещения на диске БД, обычно размером между 8 К и 128 К. **Блокировка таблицы** охватывает все строки таблицы БД.

Уровень блокировки — компромисс между параллелизмом и выполнением функций. Серверы БД имеют способность **повышения уровня блокировки**,

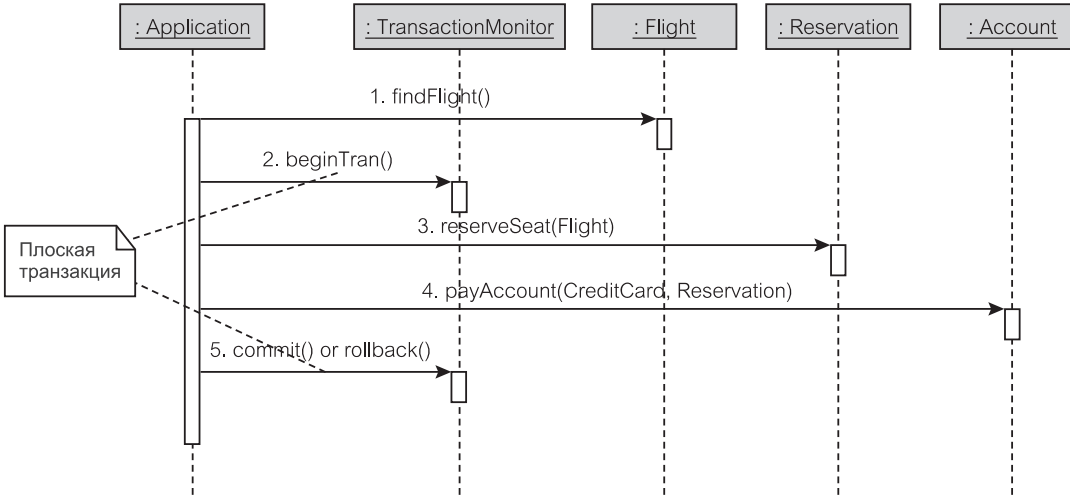


Рис. 21.1. Плоская транзакция

то есть автоматического увеличения уровня блокировки для транзакций, которые имеют доступ ко многим записям одной и той же страницы или таблицы.

21.1.4. Модели транзакций

Предыдущее обсуждение относилось к самой простой модели транзакций — плоской модели транзакций. **Плоская транзакция** — отдельная единица работы, которая заканчивается единственной фиксацией или откатом. Такая модель транзакций недостаточна для некоторых более сложных приложений, включая многие Web-приложения [89].

Рис. 21.1 представляет диаграмму последовательности действий, которая изображает плоскую модель транзакций. Пример относится к приложению, которое позволяет находить подходящий рейс самолета, резервируя место и оплачивая билет. Резервирование места невозможно, если не сделана оплата. Поэтому операции `reserveSeat()` (резервировать место) и `payAccount()` (оплатить счет) ограничены одной плоской транзакцией. Поиск рейса посредством `findFlight()` (поиск рейса) находится вне транзакции.

Если в плоской транзакции на рис 21.1 операция `payAccount()` терпит неудачу, то вся транзакция откатывается, и клиент теряет место. Формирование цепочки двух операций посредством точки сохранения (savepoint) транзакции может исправить это. **Точка сохранения** — именованный маркер в программе, который делит более длинную транзакцию на меньшие части. Она постоянно сохраняет часть работы, выполненной транзакцией до нее. Это позволяет выполнять частичные откаты к точке сохранения.

В **цепочечной транзакции** на рис. 21.2 точка сохранения по имени `Payment` (оплата) создана до выполнения операции `payAccount()`. Точка сохранения запоминает резервирование, выполненное операцией `reserveSeat()`. Если операция `payAccount()` потерпит неудачу, программа может откатить транзакцию назад к точке сохранения, таким образом,

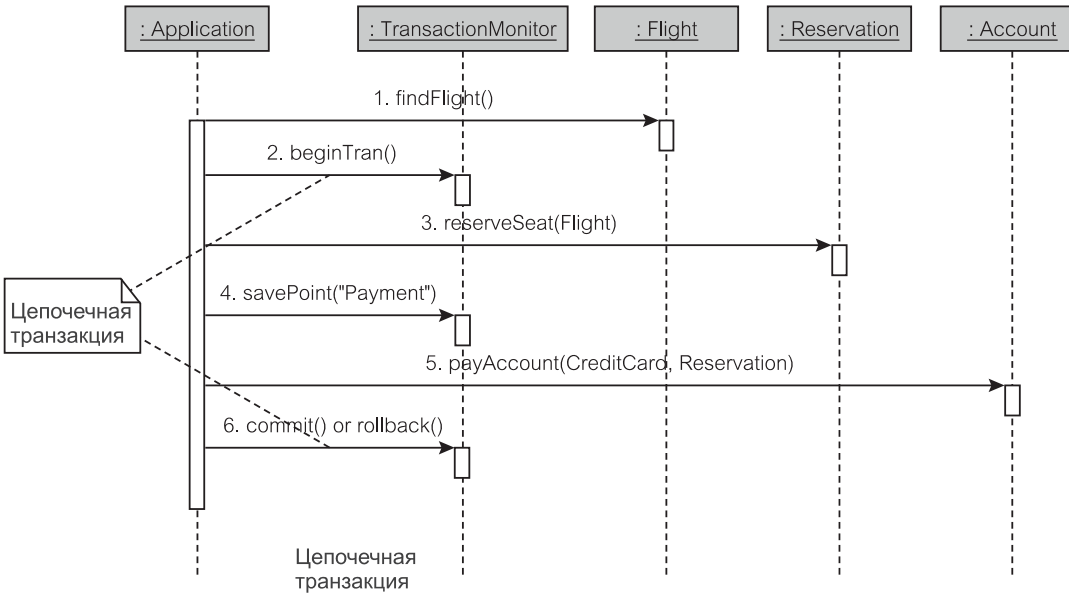


Рис. 21.2. Цепочечная транзакция

позволяя пользователю попробовать оплатить счет снова. Однако, в конечном счете, цепочечная транзакция должна или все зафиксировать, или потерпеть неудачу в унисон. Невозможно передать частичные изменения, зафиксированные точкой сохранения.

Иногда желательно делить более длинную транзакцию на меньшие части так, что эти меньшие части будут относительно независимыми и смогут быть зафиксированы или откатиться назад независимо, если это будет желательно. Такая модель известна как вложенная транзакция. **Вложенная транзакция** состоит из подтранзакций (автономных транзакций). Фиксация или откат подтранзакции не затрагивают состояние *вызывающей транзакции*, которая может продолжать свою работу независимо. (Термин «вложенная транзакция» является неудачным, поскольку на самом деле это вызывающая транзакция, которая включает другие транзакции. Возможно, более точным названием было бы «транзакция вложения».)

Рис. 21.3 является примером вложенной транзакции для резервирования различных товаров и услуг, необходимых в поездке. Резервирование поездки включает резервирование самолета и затем резервирование автомобиля и гостиницы. Идеально пользователь хотел бы резервировать все элементы поездки в единственной транзакции. Однако резервирование автомобиля и гостиницы выполнены в отдельных автономных подтранзакциях. Это позволяет делать индивидуальные резервирования, если полная поездка не может быть зарезервирована в единственном сеансе. Так, например, рейс может быть сохранен без заказа автомобиля или, наоборот, автомобиль может быть заказан, даже если рейс не сохранен.

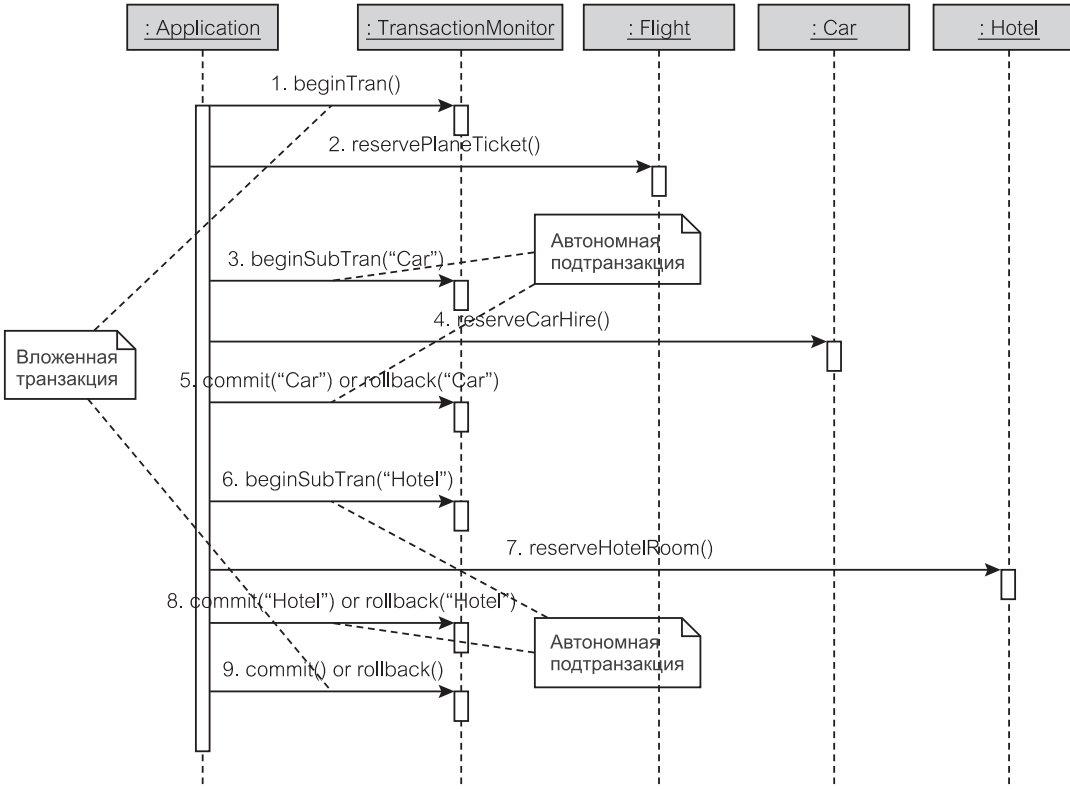


Рис. 21.3. Вложенная транзакция

21.1.5. Схемы управления параллелизмом

Управление параллелизмом, рассматриваемое до сих пор, основанное на блокировках, известно как **пессимистическое управление параллелизмом**. Пессимистическое управление параллелизмом основано на предположении, что конфликты между транзакциями являются вероятными, то есть, что параллельные транзакции, вероятно, будут нуждаться в доступе к одним и тем же ресурсам данных и обработки. Пессимистический параллелизм — схема предотвращения конфликта, посредством чего транзакция, которая использует ресурсы, запирает их от других транзакций, пока не закончит свою работу. Такая схема наиболее подходит для обычных информационных систем предприятия и является стандартной схемой в реляционных БД.

Оптимистическое управление параллелизмом использует противоположное представление, основанное на том, что конфликты между транзакциями нечасты. Это действительно может иметь место в некоторых приложениях, например в университетской системе оценок, где оценки проставляются студентам классов преподавателем, который преподает в данном классе. Оптимистический параллелизм не использует блокировки и, следовательно, избавлен от издержек на управление блокировками.

Работа при оптимистическом параллелизме выполняется в три стадии [84]:

1. Транзакция читает данные из БД без каких-либо ограничений.
2. Когда транзакция решает зафиксировать изменения в БД, система проверяет, не будут ли изменения переписывать изменения, сделанные тем временем другой транзакцией (то есть, с начала транзакции).
3. Если никакой конфликт не обнаружен, транзакции будет позволено зафиксировать изменения; иначе, транзакцию откатят назад, и пользователю нужно будет повторно начать ее.

Управление многовариантным параллелизмом (Multiversion concurrency control — MVCC) иногда рассматривается как вариант оптимистического управления. В многовариантной схеме транзакция никогда не ожидает момента, когда сможет прочитать объект данных, потому что она получает объект с новым номером версии. Чтение объекта в область транзакции (собственное рабочее пространство пользователя/приложения) называется операцией **импорта из БД** (check-out). Когда транзакция заканчивает свою работу, она помещает объекты данных назад в БД сервера в операции **экспорта в БД** (check-in). Экспорт в БД всегда выполняется успешно, потому что объект имеет свой собственный уникальный номер версии. Однако система организует проверку для версий в БД, нет ли у них конфликтов, и инициирует процесс разрешения конфликтов, который включает человеческое вмешательство.

Многовариантные схемы широко использовались как привилегированный принцип работы в **объектных БД**, типа Versant или Objectivity/DB. Объектные БД являются средствами приложений типа групповой вычислительной обработки и систем мультимедиа. В таких приложениях доминируют длинные транзакции. БД в групповой вычислительной обработке служат как хранилища таких продуктов, как архитектурные и технические рисунки, модели программной инженерии, географические карты и т. д. Пользователи этих приложений — архитекторы, проектировщики и т. д., которые импортируют из БД объекты данных в свои личные рабочие пространства и работают с ними в течение продолжительного времени — часы или дни перед экспортированием результатов их работы обратно в БД.

Объектные БД предлагают в одной системе наиболее гибкую комбинацию пессимистических, оптимистических схем и схем управления многовариантным параллелизмом. Такое объединенное использование результатов различных схем приводит к схеме, называемой **совместным управлением параллелизмом** (collaborative concurrency control). Объектные БД предназначены для обеспечения многопользовательского доступа, предоставляя совместный доступ к данным в длинных транзакциях. Пользователи работают на своих собственных рабочих станциях с клиентской частью ПО БД, запущенного на их машинах, и используют персональные БД, импортированные (скопированные) из групповой БД. Вместо изоляции пользователей объектные БД обеспечивают совместную среду, в которой те могут познакомиться с работой, сделанной другими на совместно используемых элементах данных.

Параллелизм во время **длинной транзакции** управляется **постоянной (длинной) блокировкой**. Такая блокировка имеет обычные особенности ко-

роткой блокировки, но (будучи постоянной) имеет возможность охватить **короткие транзакции** и **сеансы** БД. Чтобы иметь возможность охватить короткие транзакции в БД (**рабочем пространстве группы**), длинная транзакция запускается в личном рабочем пространстве пользователя. Длинной транзакции могут быть назначены опции организации очереди (позволяющие пользователю ожидать объект) и уведомления о блокировке (информирование пользователя других транзакций, которые используют объект). В целом основные показатели длинных транзакций следующие:

- минимизировать откаты и тупиковые ситуации;
- организовать обмен информацией (даже если он противоречивый) между сотрудничающими пользователями;
- позволить параллельные корректировки на одних и тех же объектах (в частных БД);
- обнаружить несовместимость данных и обеспечить ее разрешение.

Чтобы гарантировать ACID-свойства во время операций экспорта в БД и импорта из БД, длинные транзакции активизируют **короткие блокировки** (эти блокировки освобождаются немедленно после завершения экспорта/импорта). Длинные транзакции также активизируют *постоянные блокировки*, помещенные в объекты рабочего пространства группы. Наличие выполнения длинной транзакции в личном рабочем пространстве и коротких транзакций, запущенных в рабочих пространствах группы и личных, позволяет реализовать *вложенные транзакции*. Корректировки в БД рабочей группы могут быть зафиксированы или прерваны без фиксации либо могут прервать транзакцию окружения в личном рабочем пространстве.

Кроме типичных коротких блокировок, типа совместно используемых или взаимно исключающего доступа, объектные БД поддерживают **блокировки копированием**, позволяющие реализовать «грязное» чтение. Объект, импортируемый из БД с помощью блокировки копированием, оказывается в личном рабочем пространстве независимо от его статуса блокировки в БД группы. Если объект изменяется в личном рабочем пространстве, он экспортируется в БД как новый (другой) объект. На самом деле это является вариантом оптимистического управления параллелизмом.

Рис. 21.4 иллюстрирует, как совместное управление параллелизмом используется объектной БД, поддерживающей систему групповой вычислительной обработки [36]. Ресурсы объектов в БД группы могут не обладать или обладать версиями (раздел 5.4.2). **Идентификаторы объектов** (Object identifier — OID) включают идентификацию БД и — в случае объекта, имеющего версии — назначенный номер версии. Объекты, не обладающие версиями, имеют блокировки, наложенные согласно способу, которым они в настоящее время используются транзакциями.

Рис. 21.4 демонстрирует, что импорт из БД объекта `ob1` (1-й объект) с блокировкой чтения создает копию того же самого объекта (с неизменной идентификацией) в частной БД и это блокирует изменения `ob1` в БД группы в течение блокировки чтения (то есть, пока `ob1` не будет экспортирован в БД). Импорт из БД объекта `ob2` (2-й объект) с блокировкой записи ведет себя аналогично за исключением того, что блокируется даже доступ для чте-

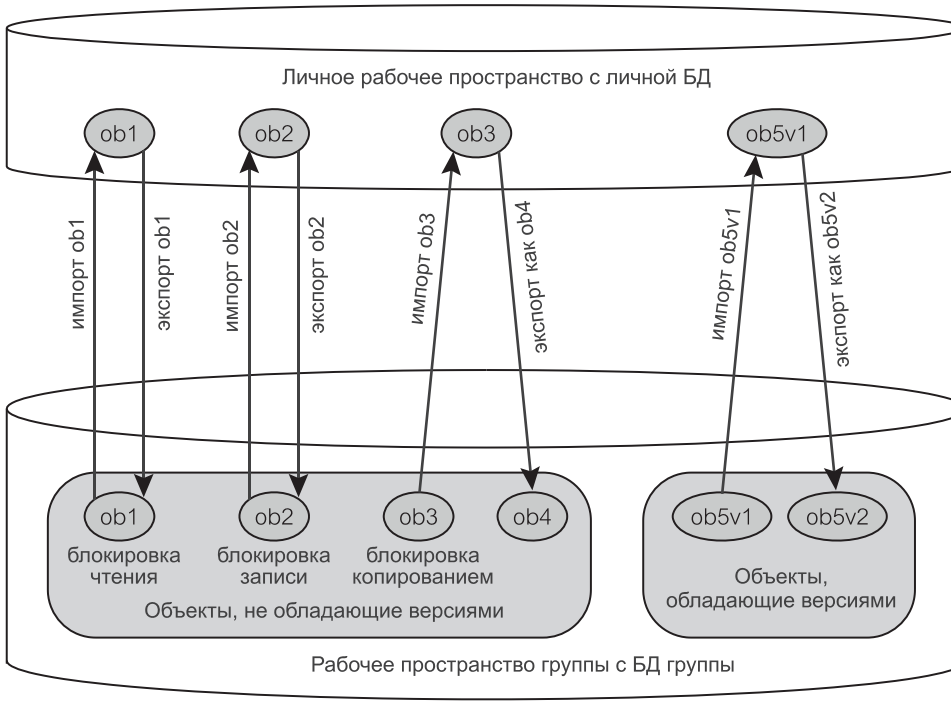


Рис. 21.4. Совместное управление параллелизмом

ния другими транзакциями в БД группы. Импорт из БД объекта `ob3` (3-й объект) с блокировкой копированием создает идентичную копию объекта в личной БД, но это действие не затрагивает (не блокирует) другие транзакции — любой последующий экспорт в БД этого объекта создает новый объект в БД группы (`ob4` — 4-й объект).

К объекту в личной БД могут обращаться более чем одна длинная транзакция, которые пользователь может подключить. Это достигается путем завершения первоначального *сеанса* без завершения длинной транзакции (так называемый *непрерывный* или *отдельный* режим) и начала нового сеанса с другой транзакцией. Конечно, запуск нескольких длинных транзакций на одном и том же объекте не изменяет ассоциацию объекта с первоначальной длинной транзакцией, используемой для проверки импорта.

Имеется несметное число вариантов реализации схем управления параллелизмом. Например, Oracle реализует интересный способ управления многовариантным параллелизмом, заменяя блокировки чтения чтением совместимых копий БД. **Чтение совместимой копии** (не имеет ничего общего с блокировкой копированием на рис. 21.4) является представлением БД так, как она выглядела в некоторый момент времени, и которое отражает только завершенные к этому моменту изменения. Это означает, что операции чтения никогда не ждут и никогда не блокируют операции записи. Oracle гарантирует непротиворечивость чтения, проверяя (оптимистично), изменился ли объект до того времени, когда транзакция будет зафиксирована. В случае запросов типа

создания отчетов для БД Oracle может использовать данные копий и не заботиться относительно изменений, которым подвергается БД. Следовательно, Oracle не нуждается в поддержке более низких уровней изоляции. Она поддерживает только чтение фиксированных данных и сериализуемых уровней.

21.2. Параллелизм в бизнес-транзакциях

Транзакция — разграниченная последовательность действий. В **системной транзакции** разграничение проходит между приложением и БД. Системная транзакция выполняет единственный запрос пользователя, типа «изменить адрес клиента», «оплатить» и т. д. В некоторых вполне определенных случаях приложений групповой вычислительной обработки, поддержанных длинными транзакциями в рамках совместного управления параллелизмом, системная транзакция может охватывать целые сеансы (раздел 21.1.5). В типичных сценариях системные транзакции являются короткими и не охватывают сеансы или запросы.

В **бизнес-транзакции** разграничение осуществляется между пользователем и приложением [31]. Многочисленные пользователи групп бизнес-транзакций запрашивают разграниченную последовательность операций. Поскольку каждый запрос пользователя соответствует единственной транзакции, бизнес-транзакция охватывает ряд системных транзакций. Шаги, необходимые для перемещения от одной системной транзакции к другой, не получают поддержку со стороны сервера, подобную механизмам управления параллелизмом, рассмотренным в разделе 21.1. Ответственность за ACID-свойства бизнес-транзакций перемещена от сервера БД к клиенту приложения и пользователю.

В отличие от **оперативного параллелизма** в системных транзакциях связующее звено, которое требуется бизнес-транзакции для объединения ряда системных транзакций так, чтобы поддерживались ACID-свойства, называется **автономным параллелизмом** [31]. J2EE-платформа обеспечивает поддержку для сервисов транзакции, выполняющих задачу, но в большинстве случаев изобретательность и мастерство разработчика — существенные факторы в обеспечении автономного параллелизма (иначе это не называлось бы «автономным»).

21.2.1. Контексты выполнения бизнес-транзакций

Выполнение программы происходит в *контекстах* [31]. В случае обработки, основанной на системных транзакциях, контексты — это сеанс и сама транзакция. **Сеанс** определяется взаимодействиями пользователя/компьютера между временем, когда пользователь зарегистрирован в системе, и временем, когда завершается его регистрация. Вероятно, сеанс будет активизировать многие системные транзакции. Длинная транзакция при совместном управлении параллелизмом может охватывать многие сеансы (раздел 21.1.5).

Системная транзакция соответствует единственному запросу, который является другим контекстом выполнения. «**Запрос** соответствует единственному вызову из внешнего мира, над которым работает ПО и на который оно необязательно возвращает ответ» [31]. Бизнес-транзакция может охватывать не-

сколько запросов типа заказа товаров и затем его оплаты. Транзакции, охватывающие несколько запросов, используют автономный параллелизм.

Сеанс возникает между двумя соседними уровнями ПО. Следовательно, может быть несколько сеансов для единственного запроса. Например, запрос от Web-клиента может вовлекать HTTP-сеанс (к Web-серверу), сеанс приложения (к серверу приложения) и сеанс БД (к серверу БД).

С точки зрения операционной системы транзакция выполняется внутри процесса или внутри потока. **Процесс** представляет временной интервал, в течение которого выполняется программа. Программа может выполняться и более чем в одном процессе. Выполнение такой программы использует взаимодействие между процессами. Контекст выполнения процесса обеспечивает значительную изоляцию для ресурсов данных, которые он использует.

Процессы тяжеловесны в том смысле, что они связывают много вычислительных ресурсов. **Поток** — легкая часть процесса, которая использует преимущество ресурсов, размещенных в процессе, хотя он все еще требует своих собственных небольших ресурсов. Привлекательность потоков для транзакций состоит в том, что многие потоки процесса могут запускаться одновременно и выполнять различные задачи. Это позволяет запускать транзакции в потоке.

21.2.2. Бизнес-транзакции и технология компонентов

Компонент — объект ПО многократного использования. Технология компонентов облегчает конструирование многозвенных промышленных приложений. J2EE-платформа поддерживает широкий диапазон компонентов, включая апплеты, клиенты приложения, JavaBeans, Enterprise JavaBeans (EJB), сервлеты и Java Server Pages (JSP) [94]. Технология компонентов порождает бизнес-транзакции.

Рис. 21.5 иллюстрирует, как PCMEF-структура соответствует уровням системы, на которых размещаются компоненты. Бизнес-транзакции проходят по всем уровням, но по большей части размещаются на уровне БД и среднем уровне. Средний уровень обычно состоит из сервера приложения и/или Web-сервера. Диаграмма показывает, что PCMEF-пакеты presentation (представление) и control (управление) часто развертываются на Web-сервере, а оставшиеся три пакета — на сервере приложения.

Системные транзакции, по крайней мере, способ, которым здесь с ними обращаются, являются синонимичными транзакциям БД. Бизнес-транзакции поднимают тему сервисов транзакций на уровнях, выше сервера БД, который находится на сервере приложения, Web-сервере и даже в апплете или клиенте приложения.

21.2.3. Распределение по уровням сервисов транзакции

J2EE-платформа определяет компоненты для сервисов транзакции в пределах **Java Transaction API** (JTA) и **Java Transaction Service** (JTS) [94]. Серверы на основе J2EE могут использовать эти сервисы. JTA определяет стандарт интерфейсов Java между различными прикладными уровнями и ресурсами, с одной стороны, и администратором транзакций, с другой [47]. JTS определяет

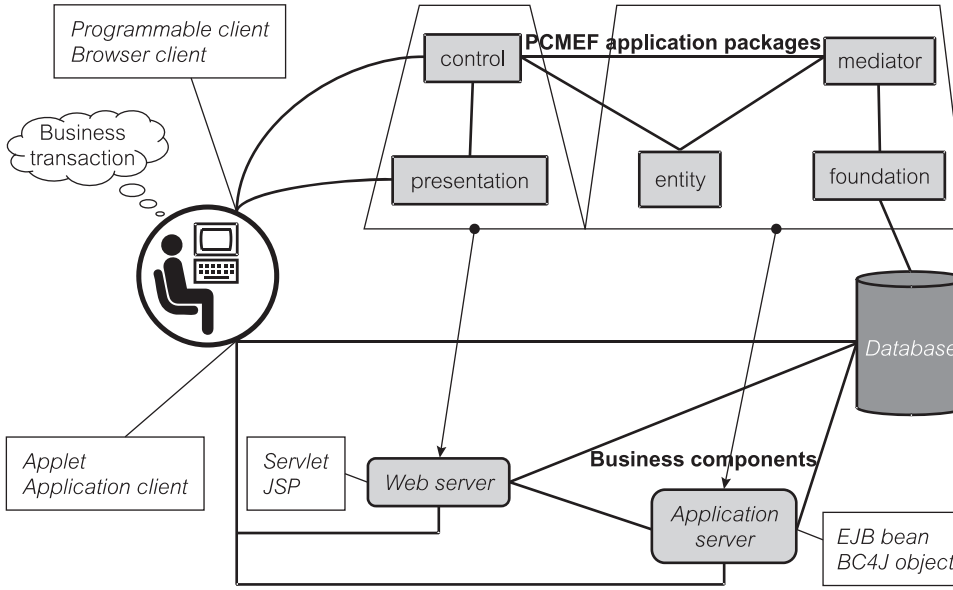


Рис. 21.5. PCMEF и компоненты

реализацию администратора транзакций, который поддерживает JTA [48]. Реализация такого администратора транзакций доступна внутри J2EE-платформы.

JTA-транзакция, управляемая J2EE-администратором транзакций, может быть распределенной транзакцией, охватывающей многие БД. Поскольку эти БД могут быть гетерогенными, J2EE-администратор транзакций имеет преимущество над менеджером транзакций, обеспечиваемым базой данных, если приложение должно использовать БД различных производителей. Как отрицательный момент, J2EE-администратор транзакций поддерживает только плоские транзакции; он не поддерживает цепочечные и вложенные транзакции (раздел 21.1.4).

J2EE-платформа предлагает два вида разграничения транзакций: декларативный и программный. Bean-компоненты предприятия в сервере приложения поддерживают декларативное разграничение. **Декларативное разграничение** использует информацию о конфигурации развертывания, чтобы выбрать путь автоматического начала и завершения транзакций. **Программное разграничение** требует прямого кодирования разграничения, используя JTA. Как только разграничение будет определено, все другие типичные функциональные возможности управления транзакциями обеспечиваются J2EE-платформой.

В большинстве случаев бизнес-транзакции инициализируются в *апплетах* и *клиентах приложения*, но реальная работа транзакций выполняется на более низких уровнях системы. Задача разграничения запросов транзакций, полученных от пользователя, обычно делегируется Web-серверу или серверу приложения (возможно, через Web-сервер). В PCMEF-структуре это соответствует делегированию работы с уровня `presentation` на уровень `control` или `domain`.

Web-уровень

JTA-транзакции могут инициализироваться в сервлете или JSP-странице **уровня Web-сервера**. Таким образом, JTA-транзакция может содержать сервлет или JSP-страницу, обращающиеся к одному или нескольким Bean-компонентам предприятия на сервере приложения, который в свою очередь может иметь доступ к одной или нескольким БД. Для сервлетов и JSP-страниц поддерживается только программное разграничение. Сервлет или JSP-компонент могут явно инициализировать JTA-транзакцию, используя интерфейс `javax.Transaction.UserTransaction` в методе `service` (сервис), как проиллюстрировано в листинге 21.1.

Листинг 21.1. Бизнес-транзакция, разграниченная в сервлете

Бизнес-транзакция, разграниченная в сервлете

```
 1:  Context ic = new InitialContext();
 2:  UserTransaction ut = (UserTransaction) ic.lookup
                           ("java:comp/env/UserTransaction");
 3:  ut.begin();
 4:  try{
 5:     insertMovieToDB(movie);
 6:     Iterator it = actors.iterator();
 7:     while(it.hasNext())
 8:        insertActorToDB((Actor)it.next());
 9:     it = listedas.iterator();
10:     while(it.hasNext())
11:        insertListedAsToDB((ListedAs)it.next());
12:     ut.commit();
13:  }catch(Exception exc){
14:     ut.rollback();
15:  }
```

Активизация `begin` (начать) объекта `UserTransaction` (транзакция пользователя) обеспечивает вызов потока для транзакции. Транзакция использует этот поток, чтобы выполнить последовательные действия транзакций, типа создания JDBC-соединения с БД. Она должна быть зафиксирована в пределах метода сервиса; это означает, что J2EE-транзакция не может охватывать несколько запросов от Web-пользовательского интерфейса (user interface — UI).

Уровень приложения

Основная обязанность уровня Web-сервера — обрабатывать представления и взаимодействия пользовательского интерфейса, не обрабатывая бизнес-логики. *Бизнес-транзакции* связаны с бизнес-логикой и, коль это так, они должны управляться **уровнем сервера приложения** всякий раз, когда это возможно (если, конечно, они не управляются уровнем сервера БД). Для этой задачи подходят Bean-компоненты предприятия в транзакции.

Bean-компоненты предприятия обеспечивают и программное (управляемое Bean-компонентом), и декларативное (управляемое контейнером) разграничение транзакций. Как показано в листинге 21.2, управляемое разграничение использует интерфейс объекта `UserTransaction` подобно примеру Web-уровня в листинге 21.1. В управляемом Bean-компонентом разграничении ссылка на `userInterface` (интерфейс пользователя) получается путем вызова `ejbContext.getUserTransaction()` (метод «получить транзакцию пользователя» объекта «контекст EJB»).

Листинг 21.2. Бизнес-транзакция, программно разграниченная в Bean-компоненте предприятия

Бизнес-транзакция, программно разграниченная в Bean-компоненте предприятия

```
 1:  MovieHome = (MovieHome) PortableRemoteObject.narrow(
                  ctx.lookup("MovieHome"), MovieHome.class);
 2:
 3:  UserTransaction ut = ejbContext.getUserTransaction();
 4:  ut.begin();
 5:  try{
 6:     home.create("Neil Jordan",
                    "Interview with the Vampire",
                    "The vampire bla bla bla");
 7:     ut.commit();
 8:  }catch(Exception exc){
 9:     ut.rollback();
10:  }
```

Листинг 21.2 пытается создать (поместить) запись в таблицу `movie` (кинофильм), вызывая метод `create()` (создать) интерфейса `MovieHome` (место кинофильма), описанного в главе 22. Успешное создание записи кинофильма приводит к выполнению метода `commit()` (фиксация), в противном случае вызывается метод `rollback()` (откат) транзакции.

J2EE-платформа поощряет использование **сервисов транзакций, управляемых контейнером**. В этом подходе методам бизнес-компонента (Bean-компонент предприятия) может быть назначен один из шести атрибутов транзакции, которые определяют поведение транзакции компонента. Как правило, всем методам Bean-компонента предприятия дают один и тот же атрибут транзакции (кроме, возможно, методов, которые не требуют никакой поддержки транзакции).

Вне определенного атрибутом поведения Bean-компонент должен ограничить влияние только на уже начатую транзакцию. Единственную вещь, которую он может делать, это откатить транзакцию с помощью метода `setRollbackOnly()` (задать только откат). Шесть атрибутов следующие [94]:

- Required (требуется);
- RequiresNew (требуется для новых);

- NotSupported (не поддерживается);
- Supports (поддерживается);
- Mandatory (принудительно);
- Never (никогда).

Метод Bean-компонента с атрибутом `Required` выполняется всегда в пределах контекста JTA-транзакции. Этот контекст может быть получен из вызывающего клиента или, если клиент не связан с JTA-транзакцией, контейнер начнет новую транзакцию для метода. Если `setRollbackOnly()` не будет использован, транзакция будет зафиксирована, когда метод завершит работу. Методы с атрибутами `Required` в различных Bean-компонентах могут быть удачно объединены, чтобы сформировать в JTA-транзакции единственный мульти-метод более высокого уровня.

Метод Bean-компонента с атрибутом `RequiresNew` начинает только новую JTA-транзакцию. Любой контекст транзакции, связанный с вызывающим клиентом, приостанавливается, и текущий поток используется для нового контекста транзакции. Приостановленная транзакция будет возобновлена на том же самом потоке, когда новая транзакция будет завершена.

Метод Bean-компонента с атрибутом `NotSupported` приостанавливает любой поступающий контекст транзакции, а сам метод выполняется без какого-либо разграничения транзакции. Когда метод будет закончен, контейнер возобновляет приостановленную транзакцию.

Метод Bean-компонента с атрибутом `Supports` обеспечивает промежуточное решение между случаями `Required` и `NotSupported`. Если метод вызывается с контекстом транзакции, он ведет себя как случай `Required`. Если он вызывается без контекста транзакции, он ведет себя как случай `NotSupported`. Это опасный атрибут, поскольку его поведение «раздвоения личности» может нарушить ACID-свойства.

Метод Bean-компонента с атрибутом `Mandatory` добавляет дальнейшее изменение к случаю `Required`. Если метод вызывается с контекстом транзакции, он ведет себя как случай `Required`. Однако когда он вызывается без контекста транзакции, контейнер формирует исключение.

Наконец, метод Bean-компонента с атрибутом `Never` запрещает использование контекста транзакции для этого метода. Метод всегда вызывается без разграничения транзакции. Если он вызывается с контекстом транзакции, контейнер формирует исключение.

Уровень БД

Системные транзакции разграничены в пределах **уровня БД**. Это маленькие транзакции, соответствующие элементарному запросу пользователя, включающему элементарное добавление, корректировку или удаление в таблице БД. Разграниченная в БД транзакция обычно обрабатывается в пределах хранимых процедур/функций, вызываемых в результате запроса пользователя. В J2EE-спецификациях такие транзакции называются **локальными транзакциями** менеджера ресурса (например, менеджера БД).

Листинг 21.3. Бизнес-транзакция, охватывающая соединения БД

Бизнес-транзакция, охватывающая соединения БД

```
 1:  Collection movies =   ...; //набор фильмов для размещения
10:  InitialContext ic = new InitialContext(System.
                                    getProperties());
11:  DataSource db1 = (DataSource)
                              ic.lookup("MovieActorDB");//JDBC
12:  ConnectionFactory db2 = (ConnectionFactory) ic.
                         lookup("VideoStore"); // Коннектор CCI
13:  java.sql.Connection con1 = db1.getConnection();
14:  javax.resource.cci.Connection con2 = db2.getConnection();
15:  UserTransaction ut = ejbContext.getUserTransaction();
16:  ut.begin();
17:  try{
18:     //выполнение корректировок для MovieDB,
                                    используя соединение con1
19:     insertMovies(con1, movies);
20:     //выполнение корректировок для VideoStore,
                                    используя соединение con2
21:     updateVideoStore(con2, movies);
22:     ut.commit();
23:  }catch(Exception exc){
24:     ut.rollback();
25:  }
```

Рекомендуемая практика J2EE-платформы (с точки зрения бизнес-транзакций) — разграничивать транзакции выше уровня БД. Другими словами, к БД нужно обращаться в пределах разграничения JTA-транзакций, происходящего в Bean-компонентах предприятия уровня приложения, или в сервлетах/JSP-страницах Web-уровня. Разграничение выше уровня БД имеет много преимуществ, включая возможность объединения коротких транзакций в более длинные элементарные единицы под управлением логики приложения. Эти транзакции могут охватывать более одного соединения БД.

Листинг 21.3 является примером бизнес-транзакции JTA, работающей на двух соединениях БД и охватывающей системные транзакции на этих двух соединениях.

Имеются случаи, когда полное разграничение бизнес-транзакции технически невозможно. Это происходит, когда сервер БД не поддерживает JTA-транзакции посредством адаптера ресурса XATransaction или подобного устройства. Такая ситуация возможна также тогда, когда конкретная форма доступа к БД не поддерживается J2EE-платформой. Во время записи J2EE-платформа поддерживает JTA-транзакции для JDBC- и для JMS-доступа. JDBC-поддержка транзакции подходит для сервлетов, JSP-страниц и Bean-компонентов предприятия.

Проблема с транзакциями, разграниченными на клиенте, состоит в том, что каждая транзакция на сервере завершается явно заданной фиксацией или откатом. Изменения, совершенные в БД, не могут быть просто незафиксированной («откат назад») JTA-транзакцией. Чтобы исправить нежелательную фиксацию, должна быть запрограммирована в прикладной логике отдельная **транзакция компенсации** [61, 94]. В зависимости от бизнес-прерогатив это трудная, а иногда и невозможная задача. В некоторых случаях может потребоваться человеческое вмешательство в БД вне области действия приложения, чтобы исправить эту проблему. Листинг 21.4 иллюстрирует принципы, стоящие за транзакциями компенсации.

Листинг 21.4. Транзакция компенсации

Транзакция компенсации

```
1:   updateRemoteSystem();
2:   try {
3:      UserTransaction.begin();
4:      updateJDBCDatabase();
5:      UserTransaction.commit();
6:   }catch (RollbackException ex) {
7:      undoUpdateRemoteSystem();
8:   }
```

21.2.4. Паттерны автономного параллелизма

Параллелизм лучше оставить управлению ПО транзакций. Однако сама концепция транзакции — бизнес-понятие. Бизнес-правила полны специальными случаями, которые иногда не могут быть разграничены некоторым простым в реализации понятием бизнес-транзакции. В других случаях доступное ПО транзакций не совсем подходит для задачи.

Одна из наиболее важных проблем, стоящих перед разработчиками, это конфликты между параллельными запросами пользователей и, следовательно, между параллельными бизнес-транзакциями. Этими проблемами занимаются **паттерны автономного параллелизма** [31]. Два из этих паттернов, Оптимистическая автономная блокировка (Optimistic Offline Lock) и Пессимистическая автономная блокировка (Pessimistic Offline Lock), обсуждаются ниже. Паттерны автономного параллелизма предполагают, что приложение реализует некоторый вид паттерна Единица работы, представленный ранее в разделе 15.3.5.

Единица работы

Паттерн **Единица работы** «поддерживает список объектов, на которые воздействует бизнес-транзакция и который координирует запись изменений и разрешение проблем параллелизма» [31]. «Паттерн Единица работы отслеживает все, что вы делаете в течение бизнес-транзакции, и что может затрагивать БД. Когда вы завершаете работу, он вычисляет все, что требуется для изменения БД в результате вашей работы» [31].

Вышеупомянутые описания Единицы работы дают несколько более гибкое и более широкое представление, чем бизнес-транзакция. Ранее в этой главе, а также в книге Фаулера [31], понятие бизнес-транзакции было ограничено действиями, выполняемыми приложениями, но воздействующими только на сохраняемые данные. Паттерн Единица работы предписывает также (если не прежде всего), какие действия должны быть предприняты над переходными объектами, находящимися в памяти, *до* открытия и разграничения бизнес-транзакции, предназначенной для фиксации изменений в БД.

Учитывая это, паттерн Единица работы представляет класс уровня `domain` (предметная область). Класс может называться `MUnitOfWork` — класс «единица работы» пакета `mediator` (предполагая, что он находится в пакете `mediator` — посредник). Он отслеживает изменения, сделанные в памяти у объектов сущности. Возможная поведенческая структура класса такова, как изображено на рис. 21.6 [31].

`MUnitOfWork` отслеживает изменения объектов сущности с помощью бизнес-транзакции. Когда приложение решит, что изменения следует сделать сохраняемыми в БД, `MUnitOfWork` начинает бизнес-транзакцию. Изменения сохраняются в списках трех массивов: `newObjects` (новые объекты), `dirtyObjects` (измененные объекты) и `removedObjects` (удаленные объекты). Списки поддерживаются соответствующими методами: `registerNew()` (регистрировать новый), `registerDirty()` (регистрировать измененный) и `registerRemoved()` (регистрировать удаленный). Метод `registeredClean()` (регистрировать очистку) не подходит для бизнес-транзакции, поскольку очищенные объекты не становятся скорректированными в БД (этот метод может использоваться, чтобы поместить зарегистрированные объекты в Коллекцию идентичности объектов — раздел 15.3.1).

Рис. 21.6. Единица работы

Метод `commit()` (фиксация) содержит запросы к трем частным методам: `insertNew()` (поместить новый), `updateDirty()` (скорректировать измененный) и `deleteRemoved()` (удалить устраненный). Вызов `commit()` в текущем объекте `MUnitOfWork` приводит к инициализации бизнес-транзакции, цель которой состоит в том, чтобы синхронизировать состояние объектов сущности, находящихся в памяти программы, с состоянием БД. В зависимости от частоты, с которой вызывается `commit()`, бизнес-транзакция будет короче или длиннее, и она может включать многие системные транзакции в БД.

Рис. 21.7 демонстрирует конкретный упрощенный сценарий с единственным изменением в объекте `EContact` (класс «деловой партнер» пакета `entity`), вызывающим `commit()`. Чтобы начать, объект `MUnitOfWork`, нацеленный на обслуживание конкретной транзакции, должен быть инициализирован. Как было сказано мимоходом, транзакция приложения может ра-

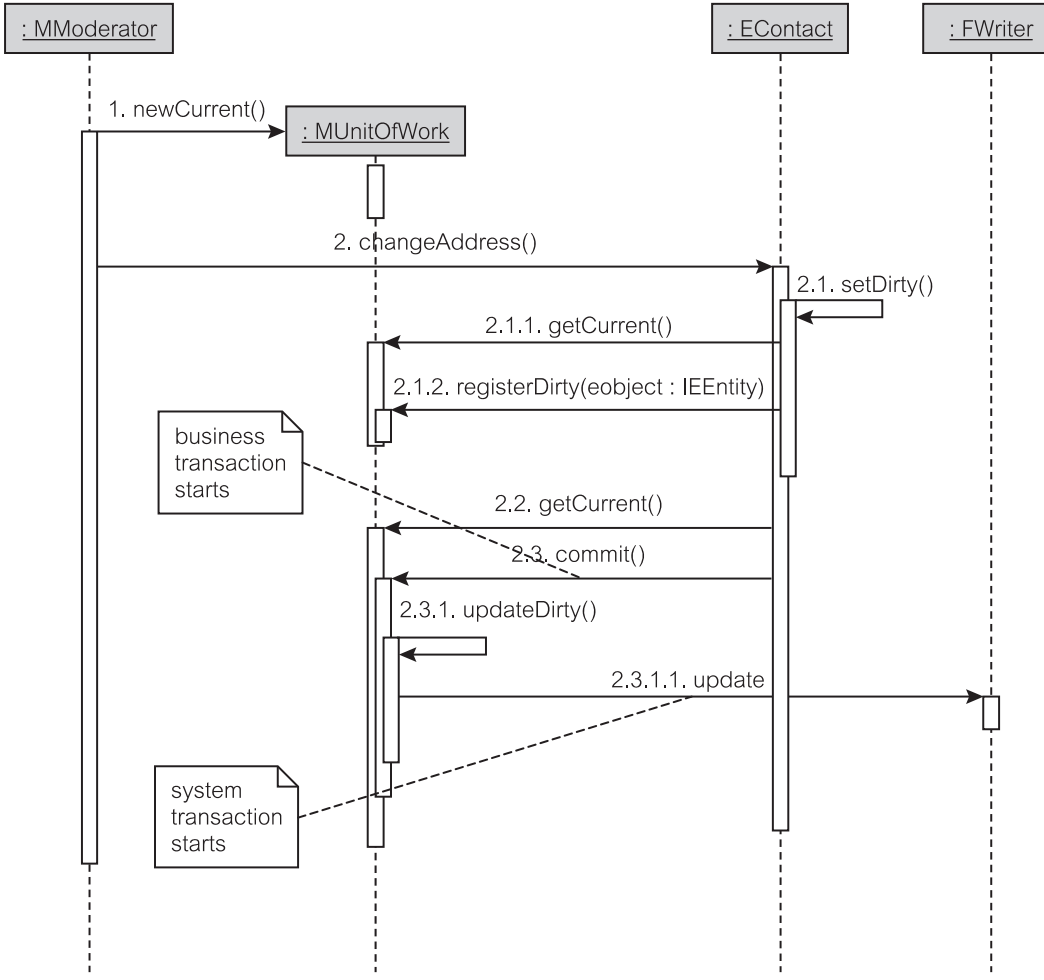


Рис. 21.7. Элемент работы, активизирующий бизнес-транзакцию

ботать на текущем потоке, используя класс `java.lang.ThreadLocal` [31]. Метод `newCurrent()` (новый текущий) инициализирует новый объект `MUnitOfWork` и размещает его в потоке текущего сеанса.

Когда `EContact` изменяет свой адрес, это вызывает `setDirty()` (объявить измененным), что в свою очередь регистрирует его статус измененного в потоке объекта `MUnitOfWork`. Когда принимается решение внести изменения в БД, `EContact` вызовет `commit()` объекта `MUnitOfWork`. Этот метод начинает бизнес-транзакцию, включающую в данном случае только метод `updateDirty()`. Метод `updateDirty()` использует `FWriter` (класс «запись» пакета `foundation`), чтобы породить системную транзакцию, которая выполняет SQL-корректировку в БД.

Оптимистическая автономная блокировка

Единица работы представляет собой решение с единственным запросом, единственным сеансом, единственной обработкой, единственным потоком и единственной бизнес-транзакцией. Но в многопользовательской системе с несколькими пользователями/приложениями, одновременно обращающимися к БД, проблемы параллелизма транзакций становятся намного более сложными. Сервисы системной транзакции решают многие из этих проблем параллелизма автоматически (раздел 21.1). Некоторые решаются сервисами бизнес-транзакции (раздел 21.2.3). Однако проблемы, возникающие благодаря конфликтам транзакций в параллельных сеансах при многих пользователях, требуют отдельного внимания при программировании.

Паттерн **Оптимистическая автономная блокировка** «предотвращает конфликты между параллельными бизнес-транзакциями, определяя конфликты и откатывая транзакции назад» [31]. В Оптимистической автономной блокировке предполагается, что конфликты являются редкими. Паттерн проверяет во время фиксации, не были ли объекты данных, записываемые в постоянный склад, изменены другой бизнес-транзакцией с тех пор, когда он были прочитаны из склада. Если они были изменены, бизнес-транзакция откатывается назад.

На практике этот паттерн эквивалентен реализованным в приложении методам непессимистических схем управления параллелизмом (раздел 21.1.5). В идеальной ситуации разработчик приложения получает помощь от схемы параллелизма, использующей различные версии, которую поддерживает администратор транзакций. Если это не так, в типичной реализации Оптимистической автономной блокировки назначается **номер версии** на каждый постоянный объект данных в системе (кое-что из того, что из управления многовариантным параллелизмом можно получить «бесплатно»).

Имея номер версии, каждая бизнес-транзакция должна:

- получить исключительный доступ к сохраняемому объекту (записи БД), чтобы скорректировать его транзакцией;
- использовать копию сеанса для номера версии в выражении `where` SQL-оператора корректировки или удаления, чтобы проверить, осталась ли версия записи неизменной;
- если версия изменилась, тогда корректировка/удаление отменяется, и бизнес-транзакция откатывается назад.

Для разрешения конфликтов «автономно», Фаулер [31] рекомендует, чтобы кроме номера версии сохраняемые объекты были еще дополнены информацией о том, когда и кто последний раз изменял запись. Это позволит текущему пользователю участвовать в «обсуждениях по согласованию» с пользователями, которые произвели недавние изменения. Фактически это является стандартной практикой в совместных схемах управления параллелизмом (раздел 21.1.5), которые обеспечивают автоматическое уведомление (например, по электронной почте), чтобы инициализировать «обсуждения по согласованию».

Пессимистическая автономная блокировка

Паттерн **Пессимистическая автономная блокировка** «предотвращает конфликты между параллельными бизнес-транзакциями, обеспечивая доступ к данным только одной бизнес-транзакции одновременно» [31]. Он эквивалентен управлению многими бизнес-транзакциями пессимистическими схемами управления параллелизмом. Пессимистическая автономная блокировка требует, чтобы бизнес-транзакция получила надлежащие блокировки данных прежде, чем она может воздействовать на эти данные. Это — точно то же самое, что делают пессимистические схемы параллелизма для системных транзакций. Бизнес-транзакции должны использовать такой паттерн, как Пессимистическую автономную блокировку.

Пессимистическая автономная блокировка рекомендует три типа блокировок данных сеанса, используемых бизнес-транзакцией [31]:

- исключительная блокировка записи;
- исключительная блокировка чтения;
- блокировка чтения/записи.

Исключительная блокировка записи гарантирует, что только одна транзакция одновременно может делать изменения в наборе данных, обрабатываемых транзакцией, которая сумела получить такую блокировку. Другие транзакции могут читать тот же набор данных, но, конечно, можно подразумевать, что они обрабатывают противоречивый набор, который может измениться ко времени, когда они закончат свою работу.

Исключительная блокировка чтения удаляет лазейку исключительной блокировки записи. При исключительной блокировке чтения бизнес-транзакция, которая получает блокировку на наборе данных, исключает другие транзакции даже от чтения этого набора данных. Другие транзакции должны ждать удаления блокировки. Ясно, что это ослабляет весь параллелизм системы.

Блокировка чтения/записи разрешает задавать многократные параллельные блокировки чтения при условии, что не установлена никакая блокировка записи. Это делает блокировки чтения и записи взаимно исключающими. Бизнес-транзакция не может получить блокировку записи, если какая-либо другая бизнес-транзакция установила блокировку чтения на том же самом наборе данных.

Реализация *менеджера блокировки*, поддерживающего Пессимистическую автономную блокировку, включает потребность в общей постоянной таблице

БД, в которой размещаются блокировки для бизнес-транзакций. Такая таблица иногда называется **таблицей-семафором**. Таблица-семафор хранит информацию о том, что в настоящее время заблокировано, какая бизнес-транзакция блокирует это, и какая применяется блокировка. Бизнес-транзакции обязаны запрашивать таблицу-семафор, прежде чем они смогут выполнять свои действия.

Последний момент поднимает проблему *протокола блокировки* для Пессимистической автономной блокировки, который диктует, когда и что блокировать, когда блокировка должна быть устранена, и что делать, если блокировка не может быть получена. Неудивительно, что большинство принципов пессимистического управления параллелизмом, используемого для системных транзакций, применяется также и для Пессимистической автономной блокировки. Блокировки должны быть получены, прежде чем данные будут загружены. Они должны быть установлены на идентификаторы (первичные ключи) записей БД и должны быть освобождены после того, как завершится транзакция. Бизнес-транзакция должна прерваться, если блокировки не могут быть получены в течение некоторого времени.

Наиболее неприятные проблемы с Пессимистической автономной блокировкой — определение и степень детализации множества данных, необходимых бизнес-транзакции (и, прежде всего, идентификация самой бизнес-транзакции), и потенциально узкое место, созданное таблицей-семафором. Упрощенное решение рассматривать бизнес-транзакции в паре с сеансом может уменьшить параллелизм до неприемлемо низкого уровня. Таблица-семафор, вероятно, станет главной точкой непосредственной установки блокировки, и, в некоторых случаях, бизнес-транзакция может испытывать трудности в получении блокировок, потому что они закрыты в таблице-семафоре.

21.3. Транзакции и параллелизм в управлении электронной почтой

ЕМ-приложение для хранения БД использует Oracle. Это дает ЕМ преимущество мощного, но сложного администратора тразакций Oracle, который использует механизм **многовариантного параллелизма** с **чтением совместимой копии** (раздел 21.1.5). Скорость запросов обеспечивается чтением копии без необходимости делать паузы в текущих корректировках БД. Однако этот подход не гарантирует, что запросами читаются современные данные. Однако данные современны относительно копии. Oracle оптимистично обнаруживает конфликты, что улучшает параллелизм (предполагается, что конфликты возникают не часто).

Использование оптимистической автономной блокировки Oracle требует, чтобы приложение клиента выполнило необходимые шаги, когда совершающееся действие могло бы вызывать исключения. Комбинация оптимистической блокировки и чтения совместимой копии не дает никаких указаний приложению клиента, если выполнение изменений в БД может завершиться конфликтом. Это существенно отличается от пессимистической блокировки, в которой имеется только одна транзакция единовременно, которая может изменять данные, и поэтому другие транзакции должны ожидать снятия блоки-

ровки. Оптимистические блокировки позволяют нескольким транзакциям выполняться одновременно и формировать исключение, только когда они обнаруживают конфликт.

Итерация 3 EM использует ряд моделей транзакций. Плоская модель транзакций используется для простых, прямых по своей сути транзакций. Цепочечные транзакции используются, чтобы гарантировать, что пользователю дают второй шанс попробовать выполнить неудавшуюся операцию. Цепочечные транзакции используют хранимые процедуры/функции, а также триггеры. Для учебного примера EM нет настоятельной необходимости использовать вложенные транзакции и автономные транзакции.

21.3.1. Модель плоской транзакции

Плоская транзакция является самой простой моделью транзакции, гарантирующей, что единственный запрос на корректировку БД или будет зафиксирован, или откатится назад. Рис. 21.8 иллюстрирует, как плоская модель транзакции используется, чтобы инкапсулировать удаление исходящего сообщения. Приложение клиента запрашивает `CAdmin` (класс «администратор» пакета `control`), чтобы удалить исходящее сообщение. Удаление выполняется объектом `MModerator` (класс «координатор» пакета `mediator`), который создает новый объект `MUnitOfWork` (класс «единица работы» пакета `mediator`) и регистрирует в нем, что исходящее сообщение должно быть удалено. Наконец, `MModerator` просит, чтобы `MUnitOfWork` зафиксировал транзакцию.

21.3.2. Единица работы и поддержка транзакций

Единица работы используется не только для того, чтобы инкапсулировать простые транзакции, как показано на рис. 21.8, но она может использоваться, чтобы инкапсулировать намного более сложные транзакции. Сложной тран-

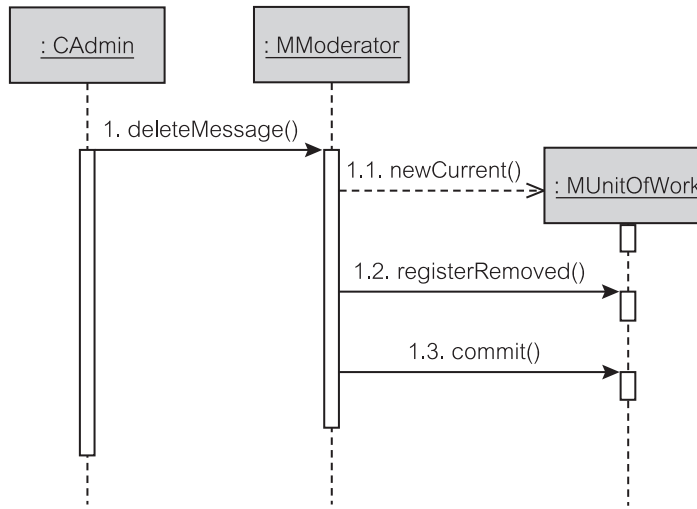


Рис. 21.8. Простая плоская транзакция в EM

закцией может быть произвольная последовательность добавлений, корректировок и удалений в БД. Хотя учебный пример EM в настоящее время не имеет сложных транзакций, `MUnitOfWork` (класс «единица работы» пакета `mediator`), реализованный в итерации 3, имеет возможность работы со сложными транзакциями.

Рис. 21.9 дает пример, в котором транзакция является последовательностью трех операций над таблицей `OutMessage` (исходящее сообщение): `insert` (добавление), `update` (корректировка) и `delete` (удаление). Объект `MUnitOfWork` можно получить с помощью его статического метода `getCurrent()` (получить текущий объект) или метода `newCurrent()` (новый текущий объект). Использование статического метода гарантирует, что будет возвращен нужный экземпляр `MUnitOfWork`. Как только единица работы будет *инициализирована* своим методом `getCurrent()` или `newCurrent()`, объекты сущности могут в нем регистрироваться, вызывая соответствующие методы типа `registerDirty()` (регистрировать измененный), `registerRemoved()` (регистрировать удаленный) или `registerNew()` (регистрировать новый) — сообщения 2.2 и 3.2.

Когда в `MUnitOfWork` будет выполнен метод `commit()` (фиксация) — сообщение 4, вся последовательность изменений будет выполняться в виде транзакции. Это приводит к созданию точки сохранения в текущем соединении с БД (сообщение 4.1). Как только точка сохранения будет задана, добавления, корректировки и удаления выполняются согласно последовательности их регистрации (сообщение 4.2). Наконец, вся транзакция (защищенная точкой сохранения) или фиксируется, или откатывается назад (сообщения 4.3 и 4.4).

Рис. 21.9 показывает, что EM вызывает `MModerator` (класс «координатор» пакета `mediator`), чтобы управлять началом и концом транзакции. Если транзакция начата с помощью `MModerator`, то и откатываться или фиксироваться она будет также с помощью `MModerator`. Если транзакция затем инкапсулируется в `MUnitOfWork`, то единица работы должна гарантировать, что с последовательностью действий обращаются как с единственной транзакцией, и поэтому она должна возвратить или зафиксированное состояние, или состояние отката.

Резюме

1. *Бизнес-транзакция* — логическая единица работы, которая выполняет законченную бизнес-задачу и гарантирует целостность БД после того, как задача будет закончена. Бизнес-транзакция состоит из одной или нескольких *системных транзакций*.
2. Управление транзакциями имеет две главные функции: параллелизм системы и восстановление системы. Эта глава имеет дело в основном с *параллелизмом системы* (параллельное выполнение программ на совместно используемых источниках данных).
3. Транзакция выделяется (разграничивается), явно или неявно, операторами, которые отмечают ее начало и ее конец. Разграничение системной транзакции известно как *разграничение на стороне сервера*, в то время

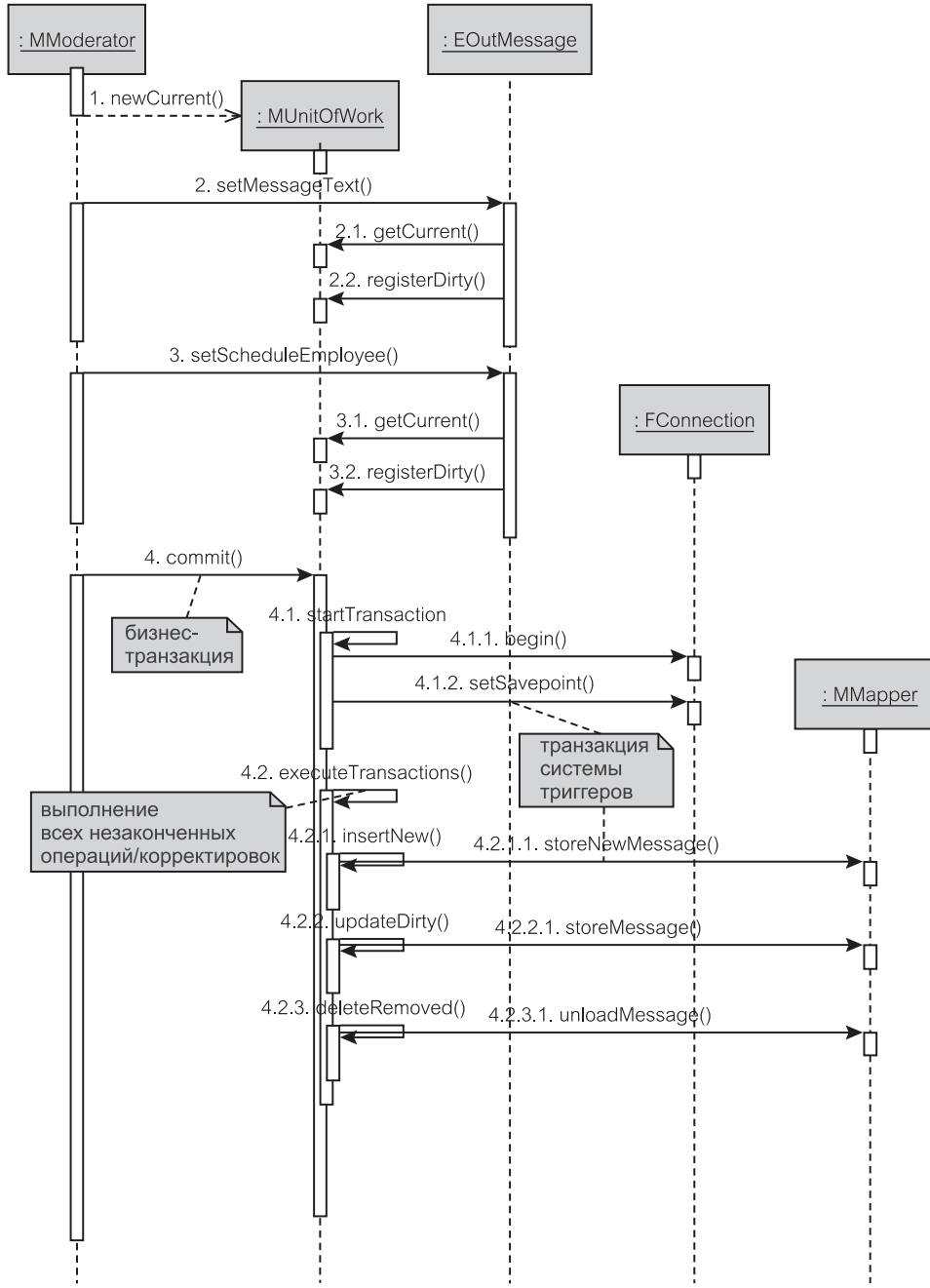


Рис. 21.9. Единица работы в действии

как разграничение бизнес-транзакции называется *разграничением на стороне клиента*. Транзакции завершаются *фиксацией* или *откатом*.

4. Чтобы обеспечить целостность данных в присутствии параллельного доступа и отказов системы, системные транзакции должны удовлетворять *ACID-свойствам* (атомарность, непротиворечивость, изоляция и долговечность).
5. *Уровень изоляции* определяет, как параллельные транзакции изолированы друг от друга. SQL-стандарт определяет четыре уровня изоляции: сериализуемый, повторяемое чтение, чтение фиксированных данных и чтение незафиксированных данных.
6. Уровни изоляции реализуются *блокировкой* ресурсов данных. Главные *способы блокировки*: блокировки взаимно исключающего доступа, корректируемые блокировки и совместно используемые блокировки. Блокировки могут быть выполнены на ресурсах различной степени детализации — они могут быть выполнены на строке, странице или на всей таблице.
7. Имеются различные модели транзакций. *Плоская транзакция* — отдельная единица работы, которая заканчивается единственной фиксацией или откатом. Части плоской транзакции могут быть реализованы в виде цепочки с использованием *точки сохранения*. Это приводит к понятию *цепочечной транзакции*. *Вложенная транзакция* состоит из подтранзакций (автономные транзакции), которые могут быть зафиксированы или откатиться назад независимо.
8. *Пессимистическое управление параллелизмом* основано на предположении, что конфликты между транзакциями являются возможными. *Оптимистическое управление параллелизмом* использует противоположное представление, что конфликты между транзакциями нечасты. Вариант оптимистического управления, используемого обычно в объектных БД, известен как *управление многовариантным параллелизмом*. Объединенное использование пессимистического, оптимистического управления и схем управления многовариантным параллелизмом называется *совместным управлением параллелизмом*.
9. В *системной транзакции* разграничение устанавливается между приложением и БД. В *бизнес-транзакции* разграничение осуществляется между пользователем и приложением. Бизнес-транзакция может охватывать многие *запросы* и *сеансы*.
10. В отличие от *оперативного параллелизма* в системных транзакциях, задача разработчика, объединяющего системные транзакции в бизнес-транзакцию и удовлетворяющего ACID-свойствам в обработке, называется *автономным параллелизмом*.
11. *Компонент* — объект ПО многократного использования. Технология компонентов облегчает конструирование многозвенных промышленных приложений. J2EE-платформа определяет компоненты для сервисов транзакций в пределах *Java Transaction API* (JTA) и *Java Transaction Service* (JTS). J2EE-платформа предлагает два вида разграничения транзакций: *декларативный* и *программный*.

12. *Бизнес-транзакции* связаны с бизнес-логикой, и в таком качестве они должны управляться *уровнем сервера приложения*. *Системные транзакции* разграничены в пределах *уровня БД*. Разграниченная в БД транзакция обычно обрабатывается в пределах хранимых процедур/функций, вызываемых в результате запроса пользователя. В J2EE-спецификациях такие транзакции называются *локальными транзакциями* менеджера ресурса (например, менеджера БД).
13. *Паттерны* для программирования *автономного параллелизма* включают Единицу работы, Оптимистическую автономную блокировку и Пессимистическую автономную блокировку.
14. Итерация 3 EM использует модели транзакций в пределах от простых плоских транзакций до цепочечных транзакций, включающих хранимые процедуры/функции, а также триггеры. Учебный пример EM не использует вложенные и автономные транзакции или другие более сложные их модели.

Ключевые термины

Обзорные вопросы

1. Объясните различие между понятиями бизнес-транзакции и системной транзакции. Как эти два термина относятся к концепциям разграничения на стороне клиента и на стороне сервера? Общее использование термина «транзакция» означает бизнес-транзакцию или системную транзакцию?
2. Какая из следующих двух пар концепций больше переплетается — непротиворечивость и долговечность или непротиворечивость и изоляция? Объясните.
3. Какая стратегия блокировки требуется для обеспечения уровня изоляции чтения фиксированных данных? Объясните и проиллюстрируйте.
4. Приведите пару примеров приложений, в которых должна использоваться вложенная модель транзакций. Объясните. Действительно ли точки сохранения необходимы для реализации вложенных транзакций? Снова дайте объяснения.
5. Могут ли длинные транзакции использовать короткие блокировки? Объясните.
6. Как блокировка копированием относится к чтению совместимой копии? Объясните.
7. Может ли запрос охватывать несколько сеансов? Может ли сеанс охватывать несколько запросов? Объясните.
8. Объясните опции развертывания для PCMEF-пакетов.

9. Объясните все «за» и «против» декларативного и процедурного разграничения. Которое из них в большей мере поддерживается J2EE-платформой? Почему?
10. Является ли транзакция компенсации системной или бизнес-транзакцией? Когда используются транзакции компенсации?
11. Является ли паттерн Единица работы также Одноэлементным паттерном? Действительно ли паттерн Единица работы является решением параллелизма? Объясните.
12. Как таблица-семафор используется в управлении параллелизмом?

Примеры задач

1. Предположим, что вы собираетесь разработать приложение со следующими требованиями: 1) приложение требует чтения данных из БД, 2) данные в приложении могут быть изменены, 3) может быть продолжительная задержка времени между чтением данных и их модификацией (то есть пользователи, возможно, анализировали данные перед их изменением) и 4) извлекаемые данные должны быть самыми последними данными, после которых никакие дополнительные данные не могут быть вставлены в БД. Какой уровень изоляции является наиболее подходящим для этого приложения? Каково влияние выбранного уровня изоляции на оперативность приложения?
2. Предположим, что вы собираетесь разработать приложение со следующими требованиями: 1) приложение требует чтения данных из БД, 2) данные в приложении могут быть изменены, 3) может быть продолжительная задержка времени между чтением данных и их модификацией (то есть пользователи, возможно, анализировали данные перед их изменением), 4) извлекаемые данные должны быть самыми последними данными и 5) приложение часто опрашивает данные в БД, чтобы учесть изменения в БД через быстрый собственный алгоритм «кэшировать и сравнивать». Какой уровень изоляции является наиболее подходящим для этого приложения? Каково воздействие выбранного уровня изоляции на оперативность приложения?
3. ACID-свойства используются не только в операциях с БД. Некоторые или все из этих свойств используются в любой операции программирования, которая размещает и совместно использует данные. Следующий оператор, если он не защищен синхронизацией, мог бы привести к неатомарному поведению. Объясните, почему. Вы можете считать, что операционная система использует квантование времени как свой алгоритм планирования. [Намек: организация многопоточной обработки]

```
value = value + newValue;
```

4. Ниже представлена реализация извлечения отчета в итерации 3 (рис. 21.10). Имеется кое-что не совсем правильное с последовательностью сообщений. Определите проблему и предложите подход для фиксации этого.

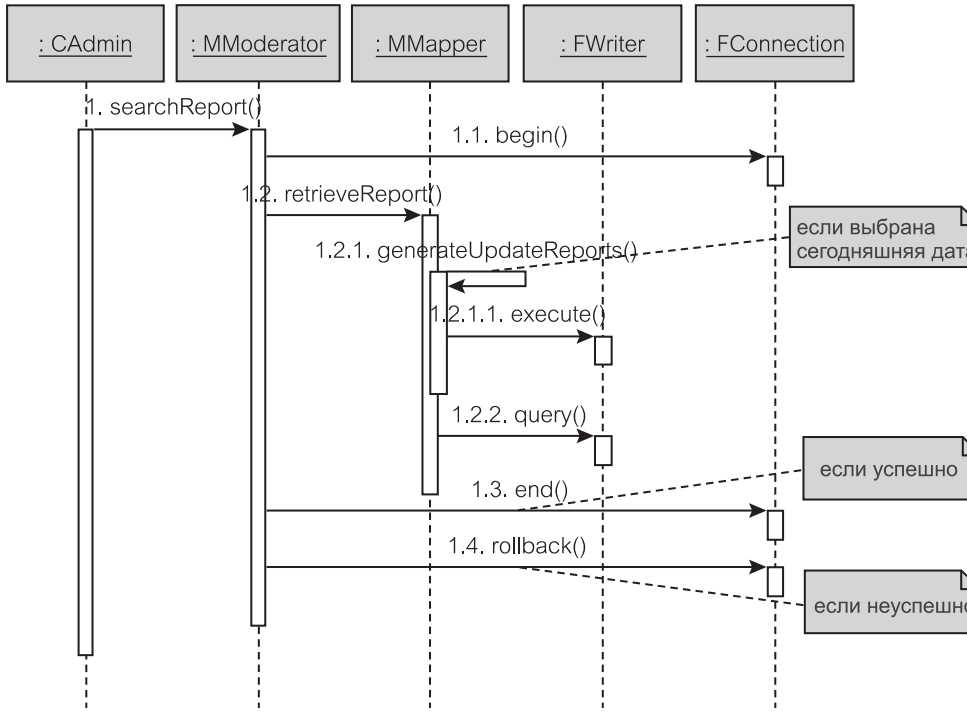


Рис. 21.10. Диаграмма последовательности для извлечения отчета

Глава

22

Бизнес-компоненты

Технология компонентов часто обсуждалась на протяжении этой книги. Эта глава, как предполагается, объединит различные «частицы и кусочки» и объяснит более подробно фундаментальные технологии Java и БД, которые помогают в конструировании систем из бизнес-компонентов многократного использования. Эта глава обсуждает две типичные технологии: 1) **Enterprise JavaBeans** (EJB), представляющую мир Java, и 2) **Business Components for Java** (BC4J), представляющую мир БД. Другие примеры технологий компонентов включают серверные объекты CORBA и ODMG-compliant Object Storage API.

Бизнес-компонент — объект многократного использования, который занимает средний уровень между пользовательским интерфейсом и БД. Бизнес-компоненты представляют сохраняемые объекты БД, реализуют бизнес-логику и бизнес-правила, знают, как получить доступ к БД, и могут выполнять бизнес-транзакции. Физическое развертывание бизнес-компонентов обычно осуществляется на сервере приложения, но некоторые компоненты могут быть размещены также и в БД или на Web-сервере или даже локально в пределах клиента Java. Имеется строгое упорядочение бизнес-компонентов по пакетам `entity` (сущность), `mediator` (посредник) и `foundation` (основание) PCMEF-шаблона (раздел 21.2.2, рис. 21.5).

Технологии бизнес-компонентов нацелены на облегчение быстрой разработки приложений стороны сервера, соединяя компоненты многократного использования через свои опубликованные интерфейсы. Компоненты могут быть свободно адаптированы и настроены.

Главное философское различие между EJB и BC4J заключается в том, что последний получает бизнес-компоненты из основного проекта БД. EJB-компоненты не получают бизнес-логику из БД; логика должна быть обеспечена EJB-компонентам разработчиком. В EJB компоненты называются **бинами** (**Bean-компонентами**). EJB предлагает *Bean-компоненты сущностей*, чтобы размещать данные, и *Bean-компоненты сеанса*, чтобы представить транзакции, безопасность и подобные функции. Эта универсальность EJB позволяет, например, развернуть BC4J-приложение в EJB Bean-компоненте сеанса.

Хороший способ понять различия между EJB и BC4J — это признать, что две данные технологии находятся на двух концах одного и того же спектра. EJB расширяет приложения по отношению к БД. В свою очередь BC4J расширяет БД по отношению к приложениям. Результатом является то, что EJB

обеспечивает транзакцию, безопасность и подобные услуги, уже представленные в БД. С другой стороны, BC4J обеспечивает GUI-компоненты, уже представленные в приложениях. EJB полагается на **JavaBeans** для GUI-компонентов.

22.1. Enterprise JavaBeans

Enterprise JavaBeans (EJB) — главный компонент шаблона Java 2 Enterprise Edition (J2EE). Основные характеристики EJB-технологии следующие [44]:

- *Ошибкоустойчивость* — структура нацелена на обеспечение надежных основных сервисов для разработки J2EE-приложений.
- *Масштабируемость* — сервер приложения может быть сгруппирован, чтобы улучшить масштабируемость ведущих приложений.
- *Поддержка принципов объектно-ориентированного проектирования* — реализация объектно-ориентированных принципов в проектировании паттернов является необходимым условием EJB, чтобы обеспечить большие, многократного использования и расширяемые системы.
- *Мобильность* — это соответствует идеологии Java «создавать однажды и использовать всюду».
- *Свойства сервера* — EJB должен обслуживать многочисленные типы клиента, такие как апплеты, автономные Swing GUI, Web-приложения (JSP/сервлеты) и т. д.
- *Проект с единственным потоком, который обладает многопоточными особенностями*, — хотя Bean-компонент можно запускать в единственном потоке, он должен демонстрировать многопоточное поведение клиента.

Необходимо подчеркнуть, что EJB — не то же самое, что JavaBeans. JavaBean — скорее стандартный класс Java, имеющий набор методов для каждого свойства, которое он хотел бы представить. Если свойство называется «hi», то имеются методы `setHi()` (задать его значение) и `getHi()` (получить его значение). JavaBeans преобладающе используются в GUI-разработке для объединения набора компонентов (или Bean-компонентов) вместе. EJB имеют несколько иное использование, чем JavaBeans. EJB преобладающе используются в программировании стороны сервера (в противоположность GUI для JavaBeans) и поддерживают различные методы, позволяющие выполнять манипуляции из *ведущего сервера* (J2EE-**контейнер**).

Имеется много источников информации, которые обсуждают все «за» и «против» EJB (и J2EE вообще). Данная глава не вступает в эти дебаты, а скорее представляет EJB в их лучшем виде: как стабильная и расширяемая структура для распределенных приложений, предполагающих наличие расширяемых услуг. Рис. 22.1 показывает полную J2EE-структуру.

Имеются две точки зрения на J2EE-структуру. С одной стороны, структуру можно рассматривать через уровни Client (клиент), J2EE и Enterprise Information System (информационная система предприятия) — EIS. С другой стороны, уровнями являются UI — пользовательский интерфейс (или Presentation — представление), Business Logic — бизнес-логика (BL) и Information System — информационная система (IS). Первая классификация

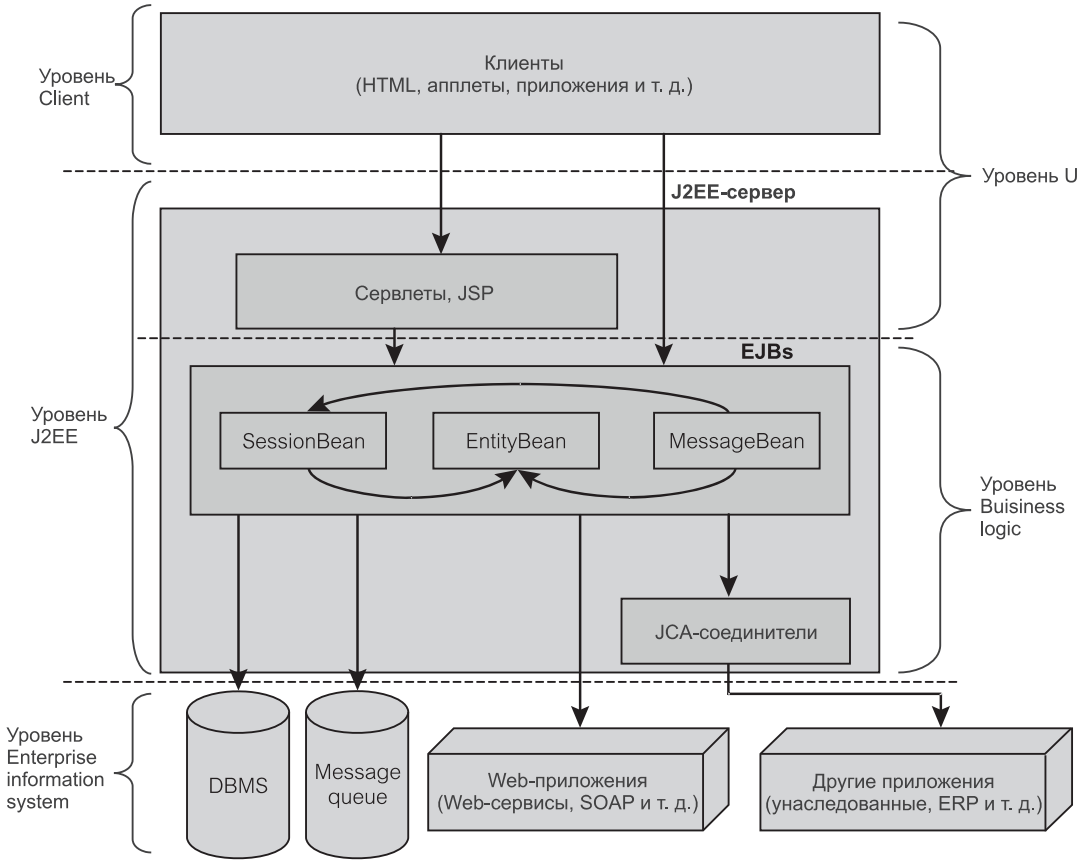


Рис. 22.1. J2EE-структура

(Client, J2EE, EIS) больше связана с технологическими изменениями между уровнями. Вторая (UI, BL, IS) больше связана с распространением информации и обязанностями системы.

Приложения, которые взаимодействуют с пользователями, находятся на **уровне Client**. Они включают апплеты, консольные и GUI-приложения, а также Web-страницы HTML. Уровень Client связывается с **уровнем J2EE**, чтобы получить необходимую информацию и обрабатывать данные. Уровень J2EE состоит из сервлетов, JSP-страниц, Bean-компонентов EJB и компонентов поддержки (типа Java Connector Architecture — JCA, Java Transaction Service — JTS, JMS, JDBC и т. д.). **Уровень EIS** состоит из БД или других источников данных (включая унаследованные системы), чтобы управлять постоянным хранением, сервисов Message Queue (очередь сообщений), чтобы обеспечить поставку сообщений, и связанных с этим сервисов.

Со второй точки зрения на J2EE-структуру нет никакого реального различия между уровнем EIS и **уровнем IS**. Однако **уровень UI** охватывает «полезные средства» уровня Client, но также включает сервлеты и JSP, поскольк-

ку они обеспечивают некоторый тип интерфейса между пользователями и бизнес-логикой. Исключение составляют те сервлеты/JSP, которые не взаимодействуют непосредственно с пользователями.

Уровень UI опирается на **уровень BL** для обеспечения своей бизнес-функциональности. Имеются три типа EJB [99, 100]. **Bean-компоненты сущностей** (entity beans) в уровнях BL или J2EE используются как представления бизнес-сущностей, а **Bean-компоненты сеанса** (session beans) как движущая сила для бизнес-логики. **Bean-компоненты сообщений** (message beans) обеспечивают средства связи в распределенных системах. И Bean-компоненты сеанса, и Bean-компоненты сообщений представляют бизнес-действия, но Bean-компоненты сообщений способны интерпретировать и передавать сообщения другим Bean-компонентам *асинхронно*.

Bean-компоненты находятся на **сервере приложения** J2EE, обычно упоминаемом как J2EE-**контейнер**. Контейнер ответствен за загрузку, доступ к Bean-компонентам, а также за «сбор мусора». Разработчик может выбрать набор управляемых контейнером Bean-компонентов или управляемых Bean-компонентами Bean-компонентов. Полные их названия — Container-Managed Persistent (CMP) Bean-компоненты (управляемые контейнером постоянные Bean-компоненты) и Bean-Managed Persistent (BMP) Bean-компоненты (управляемые Bean-компонентами постоянные Bean-компоненты). С использованием CMP Bean-компонентов контейнер ответствен за обеспечение постоянства Bean-компонентов. С BMP Bean-компонентами эта обязанность перекладывается на плечи (или, скорее, мозги) разработчика (это означает написание SQL-операторов для БД).

22.1.1. Основные принципы EJB

Основным принципом J2EE (и особенно EJB) является тот факт, что с каждым Bean-компонентом обращаются как с **распределенным объектом**. Bean-компоненты могут вызываться из процессов различных типов, как отдаленно расположенных, так и расположенных в том же самом хосте (компьютере). Клиенты могут быть реализованы на основе Java или других окружающих сред (типа C++). Все клиенты могут вызывать компоненты EJB и манипулировать ими.

EJB имеют строгие соглашения именований для обозначения их свойств и интерфейсов, которые должны быть реализованы. Они следуют примеру JavaBeans для именования методов `set()` (задать) и `get()` (получить). Дополнительно EJB требует реализации методов `find()` (найти) и `load()` (загрузить), что позволяет управлять Bean-компонентами из контейнера.

Рис. 22.2 является упрощенной версией полной **EJB-структуры**. Он не содержит J2EE-контейнера, несмотря на то, что контейнер играет главную роль в сопровождении Bean-компонентов. Рис. 22.2 содержит три интерфейса и один класс, которые определяются пользователем. Эти три интерфейса — `MyEJBHome` (мой собственный EJB), `MyEJBObject` (мой EJB-объект) и `MyEJBInterface` (мой EJB-интерфейс). Определяемый пользователем класс — `MyEJB` (мой EJB). Остальные классы и интерфейсы — составная часть J2EE API.

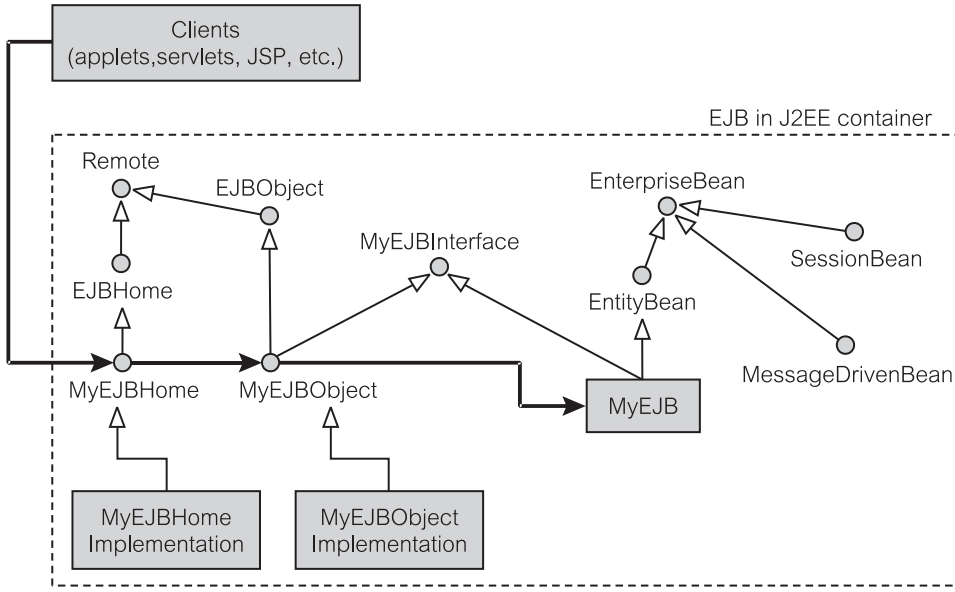


Рис. 22.2. EJB-структура с точки зрения разработчика

J2EE-контейнер автоматически формирует классы типа `MyEJBHomeImplementation` (реализация моего собственного EJB) и `MyEJBObjectImplementation` (реализация моего EJB-объекта). Обратите внимание, что обозначение `MyXXX` интерфейсов и классов дано только для иллюстрации. Было бы более логичным иметь, например, `BankAccountHome` (собственно счет банка), `BankAccountObject` (объект счета банка), `BankAccountInterface` (интерфейс счета банка) и `BankAccountEJB` (EJB счета банка) как набор классов, если приложение касается банковской области. Классы, автоматически генерируемые J2EE-контейнером, специфичны для контейнера. Приложение не имеет никакой информации об автоматически сформированных классах.

Созданные на рис. 22.2 классы формируют три типа Bean-компонентов. Клиенты должны обращаться к ним косвенно через интерфейсы. Имеются интерфейсы *Home* (собственный) и *Remote* (удаленный). Клиент обычно получает ссылку на интерфейс Home (`MyEJBHome`) конкретного Bean-компонента (`MyEJB`) и затем получает соответствующее представление Remote для EJB (`MyEJBObject`) из интерфейса Home. Этим завершается цепочка связи, поскольку клиент оказывается связанным. Клиент вызывает различные бизнес-операции на полученном объекте Remote, как будто бы этот объект представлял реальный Bean-компонент. На заднем плане объект Remote вызывает Bean-компонент, чтобы выполнить работу. Эта цепочка связи обозначена на рис. 22.2 стрелкой от прямоугольника Clients (клиенты).

Bean-компонент квалифицируется таким образом, чтобы его можно было вызывать как Enterprise JavaBean, если он расширяет `javax.ejb.EntityBean` или `javax.ejb.SessionBean` или `javax.ejb.MessageDrivenBean`. Эти Bean-компоненты расширяют `javax.ejb.EnterpriseBean`, который является *сериализуемым* интерфейсом (`java.io.Serializable`). Определяемые пользователем Bean-компоненты никогда не наследуют непосредственно от EnterpriseBean (Bean-компонент предприятия), но от одного из трех типов Bean-компонента.

Особенность **сериализации** `EnterpriseBean` означает, что все Bean-компоненты могут сохраняться — они сериализуются с помощью соответствующих методов `readObject()` (читать объект) и `writeObject()` (записывать объект). Эта особенность позволяет размещать или перезагружать Bean-компоненты всякий раз, когда это будет необходимо. Например, если число Bean-компонентов в контейнере превышает некоторый предел, контейнер может решить сериализовать некоторые Bean-компоненты. Когда эти Bean-компоненты потребуются снова, они будут *десериализованы*.

Обратите также внимание, что на рис. 22.2 `EJBHome` (собственный EJB) и `EJBObject` (EJB-объект) расширяют интерфейс `java.rmi.Remote`. Это означает, что оба объекта разработаны так, чтобы они могли поддерживать удаленное обращение. Эти объекты могут быть переданы по Интернету и вызываются из различных Java Virtual Machines (JVM) — виртуальных машин Java, используя Remote Method Invocation (удаленный вызов метода) с Internet Inter-ORB Protocol (RMI-IIOP) — протокол, определяющий передачу сообщений между сетевыми объектами по TCP/IP.

Так как EJB разработан как распределенный объект, его развертывание клиентом требует немного работы. Клиент не может связываться непосредственно с EJB, вместо этого он должен получить ссылку на `EJBHome`. Эта ссылка устанавливает канал связи с Bean-компонентом. Ссылка на `EJBHome` устанавливается с помощью некоторого именованного сервиса (Naming Service). Более того, объект `EJBHome` не может непосредственно обращаться к Bean-компоненту. Он может обеспечить этот доступ, получая ссылку на объект `EJBObject`. Единственной обязанностью `EJBHome` является создание, нахождение и уничтожение объектов `EJBObject`. Объекты `EJBObject` — именно те объекты, которые знают, как вызвать Bean-компонент.

Листинг 22.1 показывает, как клиент может получить ссылки на `EJBHome` и `EJBObject`, чтобы косвенно управлять EJB. Первоначально контекст EJB-контейнера получается из свойства системы в строке 13. Для этого контекста требуется псевдоним *MovieHome* (собственный фильм). Поиск псевдонима нацелен на класс по имени `MovieHome` (строка 15), который является интерфейсом, расширяющим **`EJBHome`** (`MovieHome` показан на рис. 22.2 как `MyEJBHome` — мой собственный EJB). Так как EJBHome является удаленным объектом (наследует `java.rmi.Remote`), требуется перевести его в известный класс. Это обеспечивается методом `narrow()` (ограничить) объекта `PortableRemoteObject` (портативный отдаленный объект) — строка 14.

Листинг 22.1. MovieClient использует Movie EJB

MovieClient использует Movie EJB

```
 1:  package presentation;
 2:  import javax.ejb.*;
 3:  import javax.naming.*;
 4:  import java.rmi.*;
 5:  import javax.rmi.PortableRemoteObject;
 6:  import java.util.*;
 7:  import domain.entity.*;
 8:  public class MovieClient{
 9:     public static void main(String args[]){
10:        MovieHome home = null;
11:
12:        try {
13:           Context ctx = new
                       InitialContext(System.getProperties());
14:           home = (MovieHome) PortableRemoteObject.narrow(
15:              ctx.lookup("MovieHome"), MovieHome.class);
16:
17:           home.create("Neil Jordan",
                         "Interview with the Vampire",
18:                       "The vampire bla bla bla");
19:           //...
20:           Iterator i =
                 home.findByDirector("Neil Jordan").iterator();
21:           System.out.println("These movies directed by Neil
                                                  Jordan:");
22:           while (i.hasNext()) {
23:              Movie movie = (Movie)
                            PortableRemoteObject.narrow(
24:                 i.next(), Movie.class);
25:              System.out.println(movie.getTitie());
26:           }
27:        }
28:     }
29:  }
```

Как только будет получен интерфейс Home (собственный), его методы создания могут быть использованы, чтобы создавать новые экземпляры Bean-компонентов (строка 17). Строки 20–27 показывают, как используется `Movie` — кинофильм (под-интерфейс `EJBObject`), чтобы представить `MovieBMPBean` (BMP Bean-компонент кинофильма) или `MovieCMPBean` (CMP Bean-компонент кинофильма). Бизнес-метод (`getTitle()` — получить заголовок) может быть вызван из полученного объекта `EJBObject`.

22.1.2. Bean-компоненты сущностей

Bean-компоненты сущностей наиболее используемы в J2EE-приложениях. Они представляют бизнес-объекты типа класса `Movie` (кинофильм) в приложении `MovieActor` («Фильм-Актер»). Главная причина того, что они столь полезны, заключается в том, что они являются **постоянными**. Поскольку они хранятся постоянно, то переживают аварии компьютеров сервера и клиента.

Как было упомянуто ранее, имеются два типа Bean-компонентов сущностей в зависимости от того, как обеспечивается их сохранность. Сохранность постоянных Bean-компонентов, **управляемых самими Bean-компонентами** (Bean-Managed Persistent — BMP), обеспечивается вручную разработчиком. Если управление сохранением оставлено контейнеру, постоянный Bean-компонент называется компонентом, **управляемым контейнером** (Container-Managed Persistent — CMP). Решение, о том, какое управление постоянством использовать, зависит от ряда факторов:

- *Сопровождение* — CMP легче обслуживать (поддерживать). Нет никаких используемых SQL-операторов и нет никакой необходимости в ручной подготовке соединения с БД. Нужно только обеспечить (главным образом) абстрактные методы, все из которых будут реализованы контейнером.
- *Заранее существующая БД* — если имеется ранее существующая рабочая БД, компоненты BMP могут быть более выгодны.
- *Знание SQL* — компоненты CMP не требуют большого знания (если таковое вообще требуется) языка SQL. EJB-Query Language (EJB-QL) — язык запросов EJB — является упрощенным SQL-вариантом для CMP, если вообще возникнет такая потребность.
- *Зависимость от продавца контейнера* — продавцы контейнеров обеспечивают различные уровни услуг в управлении постоянным хранением. Это делает компоненты CMP более уязвимыми к изменениям в возможностях контейнеров.

Листинг 22.2 показывает пример реализации Bean-компонента сущности для класса `Movie` в приложении `MovieActor`. Это BMP Bean-компонент сущности, который реализует `MovieInterface` (интерфейс фильма). Bean-компонент обладает методами, называемыми `ejbxxx`, типа `ejbFindByDirector()` (EJB — найти по режиссеру). Эти методы обычно реализуются, чтобы соответствовать *аналогично* объявленным методам в объектах `EJBObject` (EJB-объект). Например, `ejbFindByDirector()` задается, чтобы обеспечить вызовы этого метода методом `findByDirector()` (найти по режиссеру) объекта `MovieHome` (собственный фильм) или методом `findByDirector()` объекта `MovieLocalHome` (локально расположенный фильм). Слово `ejb` — является префиксом к имени метода, чтобы показать его EJB-реализацию в Bean-компоненте (сравните с его объявлением в объекте `EJBHome` — собственный EJB). Остальная часть BMP-методов имеет паттерны, аналогичные `ejbFindByDirector()`. Соединение устанавливается до того, как может быть сформирован запрос к источнику данных. Листинг 22.2 показывает, как обеспечивается соединение от накопителя соединения (`ejbPool` — EJB-накопитель). Метод также должен очищать все полученные ресурсы.

Листинг 22.2. MovieBMPBean для приложения MovieActor

MovieBMPBean для приложения MovieActor

```
public class MovieBMPBean extends EntityBean, MovieInterface
{
   //объявление некоторых полей, поддерживаемых этим
Bean-компонентом
   String director, title;
   //...вырезано для сохранения места
   public void setDirector(String director){
      this.director = director;
   }
   public String getDirector(){
      return director;
   }
   public Collection ejbFindByDirector(String director)
   throws FinderException {
      PreparedStatement pstmt = null;
      Connection conn = null;
      Vector v = new Vector();
      try {
         conn = getConnection();
         pstmt = conn.prepareStatement(
                 "select id from movie where director = ?");
         pstmt.setString(1, director);
         ResultSet rs = pstmt.executeQuery();
         while (rs.next()) {
            String id = rs.getString("id");
            v.addElement(new MoviePK(id));
         }
         return v;
      }catch (Exception e) {
         throw new FinderException(e.toString());
      }finally {
         try { if (pstmt != null) pstmt.close (); }
                                        catch (Exception e) {}
         try { if (conn != null) conn.close(); }
                                        catch (Exception e) {}
      }
   }
   private Connection getConnection() throws Exception {
      try {
         Context ctx = new InitialContext();
         javax.sql.DataSource ds = (javax.sql.DataSource)
            ctx.lookup("java:comp/env/jdbc/ejbPool");
         return ds.getConnection();
      }catch (Exception e) {
```

```
            System.err.println("Could not locate data source.
                                                    Reason:");
            e.printStackTrace();
            throw e;
        }
    }
    //...вырезано для сохранения места
}
```

Листинг 22.3 позволяет выполнить сравнение CMP-реализации с ее BMP-коллегой в листинге 22.2. `MovieCMPBean` (CMP Bean-компонент объекта Movie) имеет набор методов `set/get` (задать/получить) без их реализации. Реализация этих методов оставлена контейнеру. Bean-компонент также не имеет полей класса в отличие от BMP.

Листинг 22.3. MovieCMPBean для приложения MovieActor

MovieCMPBean для приложения MovieActor

```
public class MovieCMPBean extends EntityBean, MovieInterface
{
    //все методы set/get должны быть реализованы контейнером
    public abstract void setDirector(String director);
    public abstract String getDirector();

    public String ejbCreate(String director, String title,
                                    String desc){
        setDirector(director); //вызов методов set() вместо
                               //размещения в полях
        setTitle(title);
        setDescription(desc);
        return null; //CMP не нуждается в возвращаемой величине
    }
    //...вырезано для сохранения места
}
```

Требуется метод `ejbCreate()` (EJB-создание) и в свою очередь он отражает метод `create()` объекта `EJBHome/EJBLocalHome` (собственное EJB-вместилище/локальный собственный EJB). BMP требует наличия `ejbCreate()`, чтобы размещать значения в постоянных полях (переменных класса), в то время как CMP вызывает метод `set()`, чтобы выполнить эту работу. Методы `set()`, реализованные контейнером, обеспечивают подобный эффект (размещение постоянных полей).

Методы устройства поиска (`ejbFindxxx()`), хотя и объявлены, не должны быть реализованы в CMP. Реализация обеспечивается в описателе развертывания (листинг 22.4) в форме EJB-QL. Задавая оператор EJB-QL в описателе развертывания, контейнер может сформировать свои собственные классы, чтобы реализовать методы устройства поиска.

Листинг 22.4 представляет описатель развертывания для Bean-компонента класса `Movie`. Объявление задает, что описатель развертывания следует спецификации EJB 2.0. Имя EJB — `Movie` со связанными с ним классами (собственные, отдаленные, локальные интерфейсы). Важно отметить, что объявление также определяет, является ли это BMP или CMP. BMP имеет свой тип постоянства для Bean, в то время как CMP будет иметь тип `Container` — контейнер (закомментированные строки относятся к CMP-определенным вхождениям).

Так как `MovieBMPBean` использует соединение с источником данных, этот ресурс также объявлен в описателе развертывания. Элемент `reentrant` (элемент с возможностью повторного вхождения) — особенность, которая позволяет Bean-компоненту быть используемым в среде *с несколькими потоками*. Элемент с возможностью повторного вхождения — условие, при котором Bean-компонент вызывает другой Bean-компонент, который в свою очередь опять вызывает исходный Bean-компонент. Большинство Bean-компонентов используется *в единственном потоке*, но элемент с возможностью повторного вхождения указывает на контейнер, к которому отдельный Bean-компонент имеет многопоточный доступ. Bean-компонент, использующий единственный поток, все еще обеспечивает клиенту многопоточное поведение. Это происходит потому, что контейнер может моделировать несколько потоков для клиента (используя объединение Bean-компонентов).

Листинг 22.4. Описатель развертывания для приложения MovieActor (ejb-jar.xml) — Bean-компоненты сущностей

Описатель развертывания ejb-jar.xml для Bean-компонентов сущностей

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
  Enterprise
JavaBeans 2.0//EN' 'http://Java.sun.com/dtd/ejb-jar_2_0.dtd'>
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>Movie</ejb-name>
      <home>domain.entity.MovieHome</home>
      <remote>domain.entity.Movie</remote>
      <local-home>domain.entity.MovieLocalHome</local-home>
      <local>domain.entity.MovieLocal</local>
      <!-- <ejb-class>domain.entity.MovieCMPBean</ejb-class> -->
      <ejb-class>domain.entity.MovieBMPBean</ejb-class>
      <!-- <persistence-type>Container</persistence-type> -->
      <persistence-type>Bean</persistence-type>
      <prim-key-class>domain.entity.MoviePK</prim-key-class>
      <reentrant>False</reentrant>
      <!-- приведенное ниже требуется, если нужны BMP или
                                              явные ресурсы-->
```

```
        <resource-ref>
            <res-ref-name>jdbc/ejbPool</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
         </resource-ref>
         <!--
         следующее только для CMP
         <cmp-version>2.x</cmp-version>
         <abstract-schema-name>MovieBean</abstract-schema-name>
         <cmp-field> <field-name>director</field-name> </cmp-field>
         <cmp-field> <field-name>title</field-name> </cmp-field>
         <!--поместить другие поля здесь -->
         <query>
            <query-method>
               <method-name>findByDirector</method-name>
               <method-params>
                  <method-param>java.lang.String</method-param>
               </method-params>
            </query-method>
            <ejb-ql>
               <![CDATA[
                 SELECT OBJECT(a) FROM MovieBean AS a WHERE
                 director = ?1]]>
            </ejb-ql>
         </query>
         другие запросы размещаются здесь
         -->
      </entity>
   </enterprise-beans>
   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>Movie</ejb-name>
            <method-intf>Local</method-intf>
            <method-name>*</method-name>
         </method>
         <method>
            <ejb-name>Movie</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Required</trans-attribute>
      </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

Тег <res-auth> (ресурс авторов) определяет идентификацию, необходимую для соединения с БД. Если значение этого тега — Container, то средство развертывания будет конфигурировать авторизацию как составную часть описателя развертывания. Если величина — Application, то EJB должен определить точное пользовательское имя/пароль для соединения.

Наконец, раздел описателя компоновки в пределах описателя развертывания определяет, требуется ли управление транзакциями для Bean-компонентов, и, если требуется, то как осуществляется это управление и какие методы требуют такой услуги. Методы интерфейсов Remote (удаленный) и Local (локальный) должны быть помечены как требующие управляемой контейнером поддержки транзакций. Групповой символ (*) используется, чтобы указать, что все методы нуждаются в поддержке транзакциями.

Листинг 22.4 показывает, что описатель следует правилам CMP версии 2.x. Имя версии БД — MovieBean (Bean-компонент фильмов) с набором имен полей (столбцов в БД). Описатель использует EJB-QL, чтобы получить реализацию метода findByDirector(). EJB-QL-запрос заключен в стандартные для XML ключевые слова CDATA, чтобы сообщить программе синтаксического разбора, что нужно игнорировать содержимое запроса. Это необходимо, потому что содержимое запроса может иметь зарезервированные в XML символы типа меньше чем (<), больше чем (>) и другие.

22.1.3. Bean-компоненты сеанса

Bean-компоненты сеанса представляют бизнес-логику и бизнес-операции. Bean-компоненты сеанса управляют Bean-компонентами сущностей. Главное различие между Bean-компонентом сеанса и Bean-компонентом сущности — ресурс Bean-компонента. Bean-компонент сеанса существует только в течение сеанса, в то время как Bean-компонент сущности может жить намного дольше. Bean-компонент сеанса не постоянен. Он поддерживается сервером в памяти в течение продолжительности соединения/сеанса.

Имеются два типа Bean-компонентов сеанса в зависимости от их способности поддерживать состояние между обращениями. **Не имеющий состояний Bean-компонент сеанса** — Bean-компонент, который обслуживает запросы, но не поддерживает никакого информационного состояния между запросами. Это напоминает Web-сервер, который не делает различия между первым и последующими запросами. В отличие от этого **обладающий состояниями Bean-компонент сеанса** имеет способность помнить состояния между запросами и поэтому способен обогатить взаимодействия с клиентами. Второй и последующие запросы клиента к серверу могут привести к различным отображаемым страницам.

Примеры не имеющих состояний Bean-компонентов сеанса: 1) запрос загрузить файл с Web-сервера, 2) Web-запрос к поисковой машине, 3) «Web-червяк» (механизм поиска в WWW), который собирает информацию на основе определенного запроса, или 4) механизм перевода, чтобы перевести данный текст или Web-страницу на другой язык.

Листинг 22.5. Описатель развертывания для приложения MovieActor (ejb-jar.xml) — Bean-компоненты сеанса

Описатель развертывания ejb-jar.xml для MovieActor — Bean-компоненты сеанса

```
<!--это нужно соединить с предыдущими ejb-jar-компонентами -->
<ejb-jar>
  <enterprise-beans>
     <session>
       <ejb-name>MovieSearcher</ejb-name>
       <home>control.MovieSearcherHome</home>
       <remote>control.MovieSearcher</remote>
       <ejb-class>control.MovieSearcherBean</ejb-class>
       <!--Это для не имеющего состояния Bean-компонента сеанса
       <session-type>Stateless</session-type>
       -->
       <session-type>Stateful</session-type>
       <transaction-type>Container</transaction-type>
       <env-entry>
          <description>
             This is an example of passing environment
             variable to bean.
          </description>
          <!--
             JNDI-размещение, идентифицируемое через
             java:comp/env/mycategory/myfield.
          -->
          <env-entry-name>mycategory/myfield</env-entry-name>
          <env-entry-type>java.lang.String</env-entry-type>
          <env-entry-value>Here is the Value</env-entry-value>
        </env-entry>
      </session>
   </enterprise-beans>
</ejb-jar>
```

Примеры обладающих состояниями Bean-компонентов сеанса: 1) корзина для магазинов самообслуживания, которая обслуживает список изделий, заказанных в течение выполнения покупки, 2) менеджер корректировки ПО, который помнит отобранные корректировки, — клиент требует выполнить их перед загрузкой и инсталляцией в виде пакета, 3) диалоговый доктор, который диагностирует болезнь, требуя, чтобы пациент ответил на ряд вопросов, или 4) диалоговая система помощи, которая управляется пользователем с помощью последовательности вопросов, чтобы определить проблему.

Контейнер определяет, является ли его описатель развертывания Bean-компонентом, не имеющим состояний или обладающим состояниями. Это выполняется при входе для `session-type` — тип сеанса (листинг

22.5). Обладающие состояниями Bean-компоненты сеанса рассматриваются как нормальные Java-объекты в Java-приложении. Контейнер управляет всей необходимой работой фонового режима, чтобы управлять доступностью Bean-компонентов.

22.2. Бизнес-компоненты для Java

В PCMEF-шаблоне бизнес-компоненты соответствуют уровням `domain` — предметная область (`entity` — сущность и `mediator` — посредник) и `foundation` — основание. В EJB бизнес-компоненты состоят из Bean-компонентов сеанса и сущностей (и управляемых сообщениями Bean-компонентов, не рассмотренных в разделе 22.1). В **Business Components for Java** (BC4J) — бизнес-компоненты для Java — бизнес-компоненты фирмы Oracle называются **объектами сущностей**. В отличие от EJB пакет BC4J также поддерживает и **объекты-представления**. В результате этого BC4J позволяет быстро формировать полностью работоспособные модули приложения в комплекте с уровнем `presentation` (представление).

Многое из обработки в приложении предприятия состоит из операций программы создания-чтения-корректировки-удаления (create-read-update-delete — CRUD) над содержимым бизнес-компонентов. Современные интегрированные среды разработки (integrated development environment — IDE) могут формировать код и создавать приложения на основе описания бизнес-компонентов. Описания могут быть получены из схемы БД, из схемы языка XML (eXtensive Markup Language — расширяемая спецификация языка, предназначенного для создания страниц WWW) или из подобных источников информации о метаданных.

Среды разработки IDE типа JDeveloper фирмы Oracle обеспечивают работу мастеров, используемых в процессе разработки, редакторов компонентов и среду программирования для формирования выполняемого кода Java и XML-кода для бизнес-компонентов [13, 54]. Код приложения формируется автоматически и может быть по желанию переформирован. После формирования он может быть далее настроен и помещен в окончательное приложение.

22.2.1. Создание компонентов сущностей

Объект сущности является представлением в памяти данных БД — их определение, характеристики хранилища и связанные с этим бизнес-правила. Объект сущности может быть автоматически сформирован на основе таблицы БД, представления (то есть реляционного представления таблицы) или на основе объединения нескольких таблиц и/или представлений.

Формирование объектов сущностей, использующее среды разработки типа JDeveloper, основано на тех же самых правилах объектно-реляционного отображения, которые используются в учебном примере EM и описаны в разделе 10.2.2. Однако EM-итерации в этой книге не используют преимущества шаблона бизнес-компонентов типа EJB или BC4J. В образовательных целях в учебном примере EM выбрана реализация бизнес-компонентов «с нуля» и со своим собственным способом формирования. Это позволяет дать объяснение

PCMEF-шаблона и различных ключевых паттернов типа *Коллекции идентичности объектов*, *Преобразователя данных* и *Загрузки по требованию* (раздел 15.3). Очевидно, что EM мог бы использовать среду разработки для формирования бизнес-компонентов, а затем настройки и перепрограммирования их, чтобы обеспечить определенные потребности EM.

Приложение, содержащее бизнес-компоненты, сформированные на основе BC4J, состоит из классов Java и XML-файлов. Приложение может иметь доступ к данным в реляционной БД, объектам сущностей в кэше и передавать их на экран как объекты-представления. Создание объекта сущности приводит к формированию Java-источника и соответствующего XML-файла. Java-файл обеспечивает реализацию компонента сущности. Класс, реализованный в Java, использует XML-программу синтаксического разбора, чтобы понять формат компонента сущности, а также прикладное объектно-реляционное отображение и любые бизнес-правила, которым должен подчиняться объект сущности.

XML для компонентов сущности

Так же как Java является основным языком для Интернет-приложений, а SQL — основным языком для доступа к БД, **XML** — преобладающий язык для обмена данными. XML — язык разметки документов. Как и в наборном деле, термин «разметка» — нечто, инструктирующее, как должен быть напечатан (представлен) документ, но сам напечатанный текст не содержит никаких инструкций разметки.

Поэтому **XML-документ** состоит из текстовых данных (которые должны быть представлены) и данных разметки (которые не представляются). Как **язык разметки**, XML описывает, какая часть документа является текстовыми данными, какая часть является данными разметки и каково назначение элементов разметки и атрибутов. Значение данных разметки определяется посредством имен тегов. Следовательно, содержимое XML-документа является *самодокументируемым*.

В отличие от HTML сам XML не определяет правила форматирования документа. Для этой цели используется отдельный XML Stylesheet Language (XSL) — язык стилей XML. Также в отличие от HTML XML не определяет фиксированный набор допустимых тегов. В то время как HTML главным образом является языком форматирования документов, XML является языком представления и обмена документов. Поскольку формат XML-документа не установлен, документ может быть изменен в процессе обмена им между отправителями и получателями. Могут быть добавлены новые теги. Неожиданные теги могут игнорироваться.

Листинг 22.6 показывает отдельные строки из файла `Department.xml`, который определяет объект сущности `Department` (отдел). XML-элемент по имени `<Entity>` (сущность) является корнем документа. Он определяет, что объект сущности `Department` соответствует таблице Oracle по имени `DEPARTMENT` (строки 7–9) и что его Java-файл реализации называется `DepartmentImpl` — реализация отдела (строка 12).

XML-файл имеет элемент `<Attribute>` (атрибут) для каждого столбца в таблице `DEPARTMENT`, отобранного проектировщиком/программистом для

объектно-реляционного отображения. Элемент <Attribute> в строках 19–34 определяет свойства атрибута DepartmentCode (код отдела). Свойства включают Java- и SQL-типы данных, допустимы ли значения null, точность атрибута, является ли атрибут первичным ключом и т. д.

Элемент <AccessorAttribute> — атрибут средства доступа (строки 48–55) определяет допустимую навигацию (ассоциацию) между объектами сущности Department и Employee (служащий). Элемент <Key> — ключ (строки 64–73) отражает тот факт, что DepartmentCode является первичным ключом.

Листинг 22.6. XML-файл объекта сущности — Department.xml

Department.xml (выборка) для объекта сущности Department

```
 4:     <Entity
 5:        Name="Department"
 6:        DBObjectType="table"
 7:        DBObjectName="DEPARTMENT"
 8:        AliasName="Department"
 9:        BindingStyle="Oracle"
10:        UseGlueCode="false"
11:        CodeGenFlag="4"
12:        RowClass="PSE2BusinessComponents.DepartmentImpl" >
19:     <Attribute
20:        Name="DepartmentCode"
21:        IsNotNull="true"
22:        Precision="3"
23:        Type="java.lang.String"
24:        ColumnName="DEPARTMENT_CODE"
25:        ColumnType="CHAR"
26:        SQLType="CHAR"
27:        TableName="DEPARTMENT"
28:        PrimaryKey="true"
29:        RetrievedOnUpdate="true"
30:        RetrievedOnInsert="true" >
31:        <DesignTime>
32:           <Attr Name="_DisplaySize" Value="3" />
33:        </DesignTime>
34:     </Attribute>
35:     <Attribute
36:        Name="DepartmentName"
...
47:     </Attribute>
48:     <AccessorAttribute
49:        Name="Employee"
50:        Association"PSE2BusinessComponents.
                                  FkEmployeeRefDepcodeAssoc"
```

```
51:        AssociationEnd="PSE2BusinessComponents.
                          FkEmployeeRefDepcodeAssoc.Employee"
52:        AssociationOtherEnd="PSE2BusinessComponents.
                        FkEmployeeRefDepcodeAssoc.Department"
53:        Type="oracle.jbo.RowIterator"
54:        IsUpdateable="false" >
55:     </AccessorAttribute>
...
64:     <Key
65:        Name="PkDepartment" >
66:        <AttrArray Name="Attributes">
67:           <Item Value="PSE2BusinessComponents.Department.
                                                DepartmentCode" />
68:        </AttrArray>
69:        <DesignTime>
70:         <Attr Name="_DBObjectName" Value="PK_DEPARTMENT" />
71:           <Attr Name="_isPrimary" Value="true" />
72:        </DesignTime>
73:     </Key>
...
96:  </Entity>
```

Java для компонентов сущности

Java-файл реализации для объекта сущности, описанного в `Department.xml`, называется `DepartmentImpl.java`. Файл автоматически формируется в IDE. И Java-класс, и XML-документ могут быть изменены, чтобы создать настроенное приложение. Однако сформированный по умолчанию код для компонентов сущности достаточен, чтобы создать компоненты-представления (раздел 16.4.2) и сформировать рабочий модуль приложения (раздел 16.4.3).

Листинг 22.7 содержит фрагмент кода, сформированного в IDE, для `DepartmentImpl.java`. `DepartmentImpl` является подклассом `oracle.jbo.server.EntityImpl` (строка 11). Существенную часть произведенного кода составляют методы `get()` и `set()` для всех атрибутов в `DepartmentImpl`. Например, `getDepartmentName()` (получить название отдела) — строки 72–75 — вызывает `getAttributeInternal()` (получить внутренний атрибут), унаследованный от `EntityImpl` (реализация сущности), чтобы получить доступ к значению `DepartmentName` (имя отдела), находящемуся в объекте сущности. Java-тип объекта, который содержит значение этого атрибута, определен в элементе `<Attribute>` файла `Department.xml` (листинг 22.6). Константа `DEPARTMENTNAME` в строке 14 содержит целую величину, которая служит индексом контейнера (массива) значений атрибута внутри объекта сущности `Department` [13].

Метод `getEmployee()` — получить информацию о служащем (строки 126–129) обеспечивает доступ к объекту сущности (`Employee`), связанному с `Department`. Эта связь определена в элементе `<AccessorAttribute>` XML-документа (строки 48–55 в листинге 22.6).

Метод `CreatePrimaryKey()` (создать первичный ключ) обеспечивает первичный ключ в объекте сущности `Department`. Первичный ключ определен в элементе `<Key>` XML-документа (строки 64–73 в листинге 22.6).

22.2.2. Создание компонентов-представлений

Java-приложение может быть полезным, получая стандартные функциональные возможности CRUD от бизнес-компонентов. Не написав ни единой строки кода, тем не менее, приложение может отображать строки данных, добавлять новые данные в БД, удалять данные из БД и корректировать содержимое БД. В случае BC4J объекты-представления обслуживают цели обеспечения уровня `presentation` (представление) в приложении.

Листинг 22.7. Java-файл объекта сущности — DepartmentImpl.java

DepartmentImpl.java (выборка) для объекта сущности Department

```
1:      package PSE2BusinessComponents;
2:      import oracle.jbo.server.EntityImpl;
...
11:     public class DepartmentImpl extends EntityImpl
12:     {
13:        protected static final int DEPARTMENTCODE = 0;
14:        protected static final int DEPARTMENTNAME = 1;
15:        protected static final int EMPLOYEE = 2;
16:        protected static final int OUTMESSAGE = 3;
...
72:        public String getDepartmentName()
73:        {
74:         return (String)getAttributeInternal(DEPARTMENTNAME);
75:        }
...
81:        public void setDepartmentName(String value)
82:        {
83:           setAttributeInternal(DEPARTMENTNAME, value);
84:        }
126:       public RowIterator getEmployee()
127:       {
128:        return (RowIterator)getAttributeInternal(EMPLOYEE);
129:       }
...
145:  public static Key createPrimaryKey(String departmentCode)
146:        {
147:           return new Key(new Object[] {departmentCode});
148:     }
153:  }
```

Объект-представление обеспечивает визуализацию SQL-запроса к БД. Запрос может завершиться отфильтрованным и отсортированным подмножеством атрибутов из одного или нескольких объектов сущностей. Получение данных из нескольких объектов сущностей аналогично выполнению операции объединения таблиц БД.

Самостоятельная концепция **связи представлений** позволяет обеспечить визуализацию отношения родителя-потомка, наподобие отношения между счетом и строками счета или между отделом и служащими. Визуализация отношения родителя-потомка использует отдельный объект-представление (называемый представлением родителя), который обеспечивает объект родителя как контекст для всех характеристик (потомков) объектов. Информация характеристик изменяется, когда требуется и отображается другой активный объект-родитель.

XML для компонентов-представлений

Листинг 22.8 показывает отдельные строки из файла `DepartmentView.xml`, который определяет объект-представление `DepartmentView` (представление отдела). Исходный элемент по имени `<ViewObject>` (объект просмотра) определяет SQL-запрос (строки 6–8) для `DepartmentView`. Элемент `<EntityUsage>` (использование сущности) сообщает, какой объект сущности связан с этим объектом-представлением. Элементы `<ViewAttribute>` (просмотр атрибутов) определяют атрибуты объекта сущности, которые нужно «рассмотреть». Элементы `<ViewLinkAccessor>` (просмотр связей средства доступа) определяют связи представлений, которые объект-представление может использовать, чтобы установить отношения родителя-потомка.

Листинг 22.8. XML-файл объекта-представления — DepartmentView.xml

DepartmentView.xml (выборка) для объекта-представления DepartmentView

```
4:   <ViewObject
5:      Name="DepartmentView"
6:      SelectList="Department.DEPARTMENT_CODE,
7:         Department.DEPARTMENT_NAME"
8:      FromList="DEPARTMENT Department"
9:      BindingStyle="Oracle"
10:     CustomQuery="false"
11:     ComponentClass="PSE2BusinessComponents.
                                    DepartmentViewImpl">
...
25:     <EntityUsage
26:        Name="Department"
27:        Entity="PSE2BusinessComponents.Department" >
...
```

```
32:     </EntityUsage>
33:     <ViewAttribute
34:        Name="DepartmentCode"
35:        IsNotNull="true"
36:        EntityAttrName="DepartmentCode"
37:        EntityUsage="Department"
38:        AliasName="DEPARTMENT_CODE"
39:        ColumnType="VARCHAR2" >
...
48:     <ViewLinkAccessor
49:        Name="EmployeeView"
50:        ViewLink="PSE2BusinessComponents.
                                FkEmployeeRefDepcodeLink"
51:        Type="oracle.jbo.RowIterator"
52:        IsUpdateable="false" >
53:     </ViewLinkAccessor>
...
60:  </ViewObject>
```

Java для компонентов-представлений

Java для компонентов-представлений наследует большую часть своего кода из IDE-библиотеки класса. В случае BC4J родительский класс называется `ViewObjectImpl` — реализация объекта-представления (листинг 22.9). Обеспечиваемые функциональные возможности включают порядок сортировки для отображаемых строк информации, критерии поиска для ограничения числа отображаемых строк, размещение навигатора (итератора) строк в отдельном объекте строк (первый, последний, следующий) и т. д.

Листинг 22.9. Java-файл объекта-представления — DepartmentViewImpl.java

DepartmentViewImpl.java для объекта-представления DepartmentView

```
package PSE2BusinessComponents;
import oracle.jbo.server.ViewObjectImpl;

public class DepartmentViewImpl extends ViewObjectImpl
{
   public DepartmentViewImpl()
   {
   }
}
```

22.2.3. Создание модуля приложения

Модуль приложения для бизнес-компонентов использует определения компонентов сущностей и объектов-представлений, чтобы обеспечить приложение функциональными возможностями, используемыми по умолчанию. Модуль сгенерирован в BC4J. Сформированный код может установить соединение с БД и обеспечить контекст транзакции для операций над БД. Модуль может быть развернут на среднем уровне, например, на EJB.

Как и в случае компонентов сущностей и компонентов-представлений, модуль приложения определен парой — XML-документом и Java-файлом. Модуль определяет, какие представления являются доступными, и объединяет представления в отдельный шаблон. Рисунки 22.3 и 22.4 являются примерами двух представлений, выполняемых из сгенерированного модуля приложения. Рис. 22.3 показывает простое представление `OutMessage` (исходящее сообщение). Рис. 22.4 отображает представление основных характеристик, которое показывает список объектов `OutMessage`, созданных мастером объекта `Entity` (сущность).

Резюме

1. *Бизнес-компонент* — объект многократного использования, который занимает средний уровень между пользовательским интерфейсом и БД. Две типичные технологии компонентов: 1) *Enterprise JavaBeans* (EJB), представляющая мир Java, и 2) *Business Components for Java* (BC4J), представляющая мир БД.

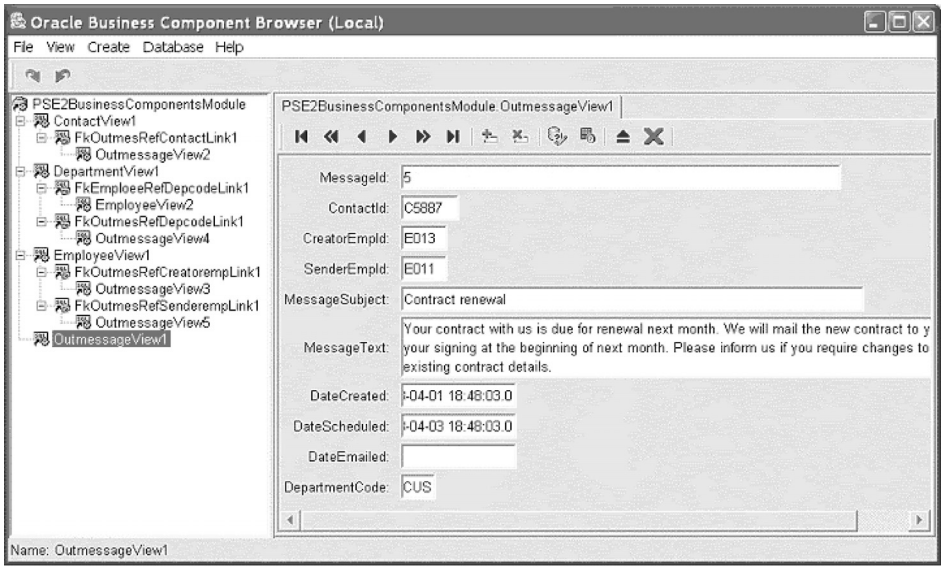


Рис. 22.3. Модуль приложения, показывающий простое представление
Источник: образ экрана Oracle Business Component Browser из www.otn.oracle.com, перепечатано с разрешения Oracle Corporation

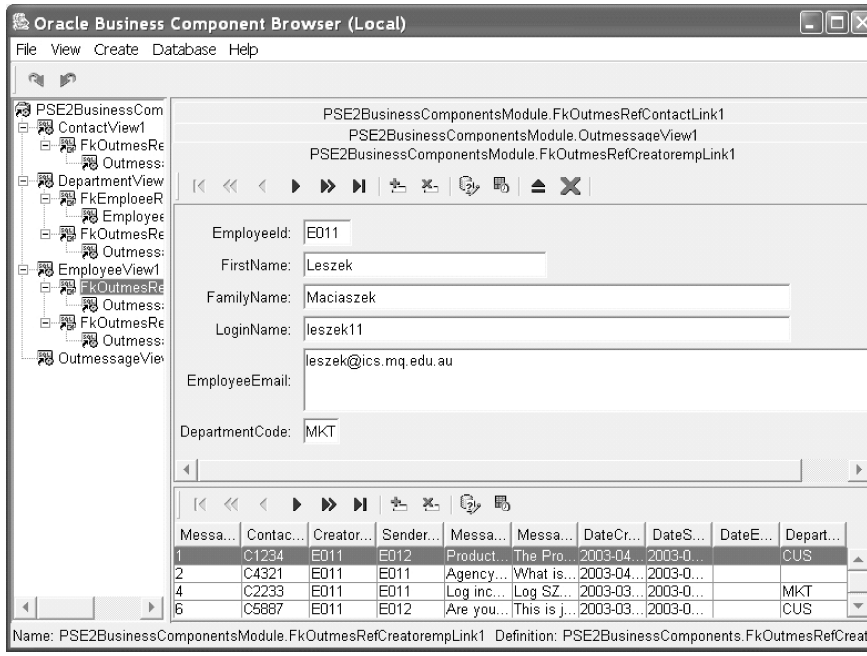


Рис. 22.4. Модуль приложения, показывающий представление основных характеристик Источник: образ экрана Oracle Business Component Browser из www.otn.oracle.com, перепечатано с разрешения Oracle Corporation

2. Компоненты в EJB называются *Bean-компонентами*. EJB предлагает *Bean-компоненты сущностей*, чтобы хранить данные, и *Bean-компоненты сеанса*, чтобы представлять транзакции, безопасность и подобные функции. EJB основана на *JavaBeans* для GUI-компонентов.
3. EJB — основная составная часть шаблона Java 2 Enterprise Edition (J2EE). Bean-компоненты в J2EE находятся на *сервере приложения*, обычно упоминаемом как J2EE-*контейнер*. С каждым Bean-компонентом обращаются как с *распределенным объектом*.
4. *Bean-компоненты сущностей* — наиболее часто используемые Bean-компоненты в J2EE-приложениях. Они представляют бизнес-объекты. Эти объекты являются *сохраняемыми*.
5. *Bean-компоненты сеанса* представляют бизнес-логику и бизнес-операции. Они управляют Bean-компонентами сущностей. Bean-компонент сеанса существует только во время продолжительности сеанса, в то время как Bean-компонент сущности постоянен. Bean-компоненты сеанса могут быть *не имеющими состояния* или *обладающими состояниями*.
6. В BC4J бизнес-компоненты известны как *объекты сущностей*. В отличие от EJB BC4J также поддерживает *объекты-представления*. BC4J-генерируемое приложение бизнес-компонентов состоит из Java-классов и XML-файлов.

7. *Объект сущности* — представление в памяти данных из БД — их определение, характеристики хранения и связанные бизнес-правила. Объект сущности может быть автоматически сформирован на основе таблицы БД (то есть, представление реляционной таблицы) или на основе объединения нескольких таблиц и/или представлений.
8. *Объект-представление* обеспечивает визуализацию SQL-запроса на основе БД. Запрос завершается отфильтрованным и отсортированным подмножеством атрибутов из одного или нескольких объектов сущностей.
9. *Модуль приложения* BC4J для бизнес-компонентов использует определения объектов сущностей и объектов-представлений, чтобы сгенерировать приложение с функциональными возможностями по умолчанию. Сформированный код способен установить соединение с БД и обеспечить контекст транзакции над БД.

Ключевые термины

Обзорные вопросы

1. Являются ли бизнес-компоненты многократно используемыми? Объясните.
2. Являются ли бизнес-компоненты распределенными? Объясните.
3. Перечислите и обсудите различия между EJB Bean-компонентами сущностей и BC4J объектами сущности.

4. EJB использует JavaBeans для GUI-компонентов. Объекты-представления BC4J являются GUI-компонентами. Каковы различия между JavaBeans и объектами-представлениями?
5. Какова последовательность шагов, которую должно выполнить приложение клиента, чтобы запустить EJB?
6. Какие компоненты должны быть определены, чтобы реализовать Bean-компонент предприятия?
7. Каковы главные шаги для разработки BC4J-приложения?
8. Найдите в Интернете последние разработки и новые технологии для бизнес-компонентов. Имеются ли какие-то изменения в направлениях? Как EJB и BC4J подходят для любых новых тенденций?

Глава

23 Итерация 3. Аннотированный код

В этой главе представлен код, связанный с *проектированием данных* в учебном примере управления электронной почтой (Email Management — EM). Стиль представления, приведенный в этой главе, подобен подходу, используемому в главе 18. Представлены копии экрана диаграмм классов (созданные средством **yDoc** фирмы yWorks) и они сопровождаются документацией **Javadoc**. Код, который не изменился с предыдущих итераций, в значительной степени опущен при обсуждении в этой главе. Полный код для этой и предыдущих итераций можно посмотреть и при необходимости загрузить с Web-сайта книги.

Глава начинается с обзора кода для итерации 3, сопровождаемого детальными рассмотрениями конкретных пакетов и классов. Специальный акцент сделан на объяснении классов и кода, поддерживающих новые функциональные возможности, а также на структурных требованиях, типа ежедневного итогового отчета или единицы работы.

По сравнению с главами 13 и 18 эта глава включает отдельное обсуждение кода БД (еще не упоминавшееся в более ранних главах части 4). Оно демонстрирует существенную роль, которую играет код БД в итерации 3 в достижении *безопасности и целостности БД* и в обеспечении *управления параллелизмом транзакций*. Большинство модификаций по сравнению с итерацией 2 относится к пакету `mediator` (посредник), где начинаются все транзакции пользователя.

23.1. Обзор кода

Итерация 3 подсистемы EM является заключительной итерацией учебного примера. Эта итерация концентрируется на проблемах проектирования данных и связана с некоторыми нерешенными моментами в отношении структурного шаблона PCMEF+. Однако шаблон остается неизменным, а названия пакетов — те же самые, что и в предыдущих итерациях. По сравнению с итерацией 2 главные изменения, представленные в итерации 3, следующие:

- введение `MUnitOfWork` (класс «единица работы» пакета `mediator`), чтобы обеспечить *поддержку транзакций* для приложения;
- весь SQL-код перемещен на сервер БД в форме *хранимых процедур/функций*;

- все транзакции пользователя начинаются в `MModerator` (класс «координатор» пакета `mediator`);
- *права доступа* применяются ко всем операциям, чтобы ограничить тип исходящих сообщений, просматриваемых служащими;
- права доступа могут быть изменены только пользователем в роли администратора безопасности/БД;
- реализованы функциональные возможности создания ежедневных *итоговых отчетов*;
- исходящие сообщения могут быть изменены перед передачей по электронной почте.

Рис. 23.1 показывает заключительную модель классов для итерации 3. Изменения по сравнению с итерацией 2 включают дополнительные классы (на рисунке они оттененные). Новые классы реализуют просмотр прав доступа, ежедневных сообщений, единиц работы и другие незначительные изменения из-за рефакторинга. За исключением нескольких ассоциаций, имена ролей для ассоциаций были удалены, чтобы улучшить удобочитаемость диаграммы.

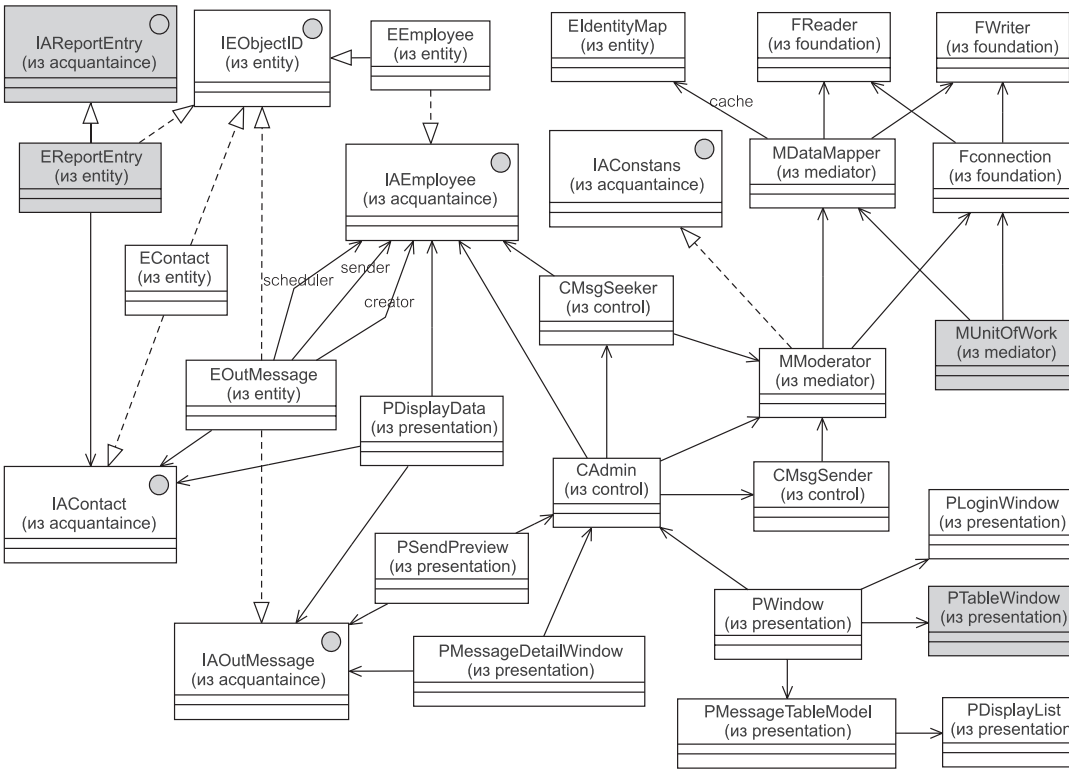


Рис. 23.1. Диаграмма классов для итерации 3 подсистемы EM

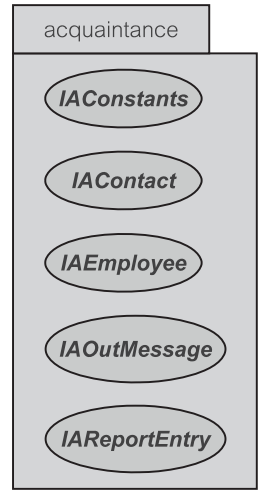


Рис. 23.2. Пакет Acquaintance и его классы в EM 3

23.2. Пакет Acquaintance

Единственный дополнительный интерфейс, представленный в пакете `acquaintance` (знакомство) на рис. 23.2 — это `IAReportEntry` (интерфейс «вхождение в отчет» пакета `acquaintance`). Интерфейс помогает в формировании ежедневных итоговых отчетов, передавая операции по электронной почте. Можно также поместить как дополнительный интерфейс в этом пакете `IAAutorizationRule` (интерфейс «правило доступа» пакета `acquaintance`), хотя это и не реализовано в итерации 3 и оставлено как упражнение для пользователя.

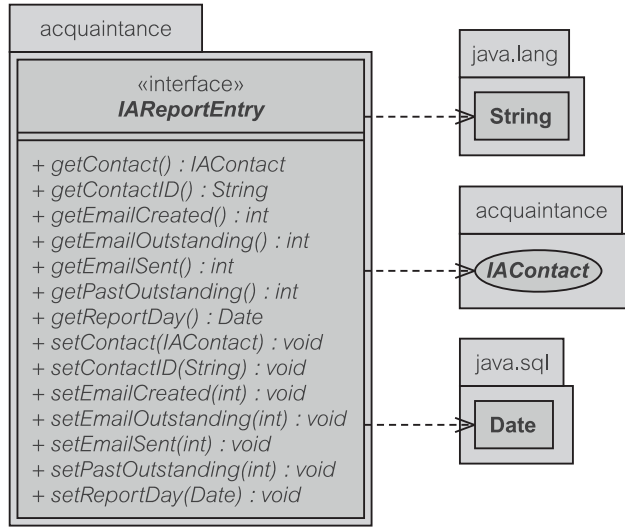


Рис. 23.3. Интерфейс IAReportEntry в EM 3

Другие интерфейсы, объявленные в пакете `acquaintance` — те же самые, что и в итерации 2.

23.2.1. Интерфейс IAReportEntry

`IAReportEntry` (интерфейс «вхождение в отчет» пакета `acquaintance`) является интерфейсом, который обеспечивает единственный ввод строки отчета в БД. Содержание отчета состоит из информации о деловом партнере и статистики относительно исходящих сообщений создавшего их служащего, типа числа созданных исходящих сообщений или числа невыполненных исходящих сообщений для указанной даты. Рис. 23.3 показывает определение интерфейса `IAReportEntry`.

23.3. Пакет Presentation

В пакет `presentation` (представление), который содержит все восемь классов и один внутренний класс, введен дополнительный класс, как показано на рис. 23.4. Дополнительный класс `PTableWindow` (класс «окно таблицы» пакета `presentation`) используется, чтобы отображать **права доступа**, а также ежедневные отчеты.

23.3.1. Класс PWindow

Общедоступная сигнатура класса `PWindow` (класс «окно» пакета `presentation`) не была изменена по сравнению с ее первоначальной версией (рис. 18.6). Однако дополнительная реализация и изменения в `PWindow` представлены приватными методами. `PWindow` становится **Фасадом** (см. раздел 9.3.1) для различных классов пакета `presentation` (представление), типа `PTableWindow` (класс «окно таблицы» пакета `presentation`).

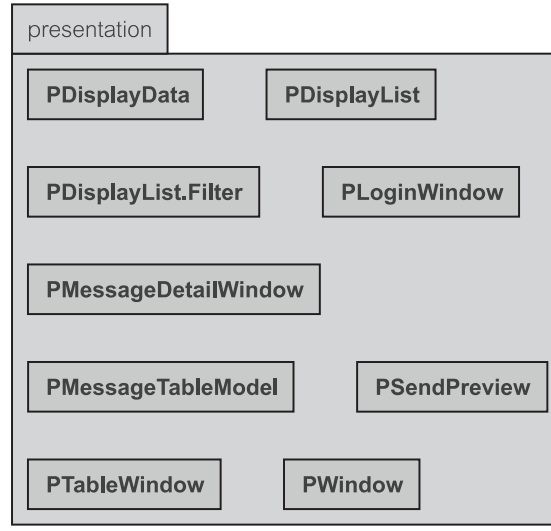


Рис. 23.4. Пакет Presentation и его классы в EM 3

Конструирование PWindow следует той же самой процедуре, которая рассмотрена в разделе 18.3.1. Устанавливается ассоциация с CAdmin (класс «администратор» пакета control), диалоговое окно регистрационного имени запрашивает идентификацию пользователя, и данные извлекаются, чтобы заполнить соответствующие поля. Дополнительные поля типа списка деловых партнеров отчета также заполняются на этой стадии.

Следующее обсуждение посвящено трем подпотокам (раздел 19.2.3) документа сценария использования для итерации 3. Это подпотоки S8 (средство формирования отчета), S1 (процедура авторизации) и S7 (средство модификации исходящего сообщения).

Заполнение списка деловых партнеров в отчете

Сценарий использования Produce Daily Report (создание ежедневного отчета) обеспечивает для пользователя вариант формирования **ежедневных итоговых отчетов** относительно операций по электронной почте только для определенных деловых партнеров. Метод populateReportContact() (заполнить отчет деловыми партнерами) в PWindow используется, чтобы обеспечить это (листинг 23.1). Список деловых партнеров извлекается из объекта admin (администратор) с помощью его метода listContacts() (деловые партнеры списка) — строка 53. Метод listContacts() возвращает набор IAContact (интерфейс «деловой партнер» пакета acquaintance). Имена в списке деловых партнеров формируются из фамилий и имен IAContact (строка 56).

Листинг 23.1. Метод populateReportContact() в PWindow, чтобы поместить список деловых партнеров в отчете

```
Метод populateReportContact() в PWindow, чтобы поместить список
деловых партнеров в отчете

49:     private void populateReportContact(){
50:        synchronized(rptContactCmb){
51:           rptContactCmb.removeAllItems();
52:           rptContactCmb.addItem("");
53:           Iterator it = admin.listContacts().iterator();
54:           while(it.hasNext()){
55:             IAContact ctc = (IAContact) it.next();
56:             String contactName = ctc.getFamilyName() +", "+
                                          ctc.getFirstName();
57:             rptContactCmb.addItem(contactName);
58:           }
59:        }
60:     }
```

Пользователь может пожелать сформировать отчет для всех деловых партнеров вместо отчета для конкретного делового партнера. Это может быть получено путем задания в комбинированной строке ввода rptContactCmb

(комбинированная строка ввода делового партнера в отчет) пустого поля (строка 52).

Окно отчета

Имеется много способов сформировать окно отчета. Подход, реализованный в итерации 3, использует преимущество **сервисного класса**, созданного и заполненного соответствующими данными. Это — динамический подход многократного использования. Данные должны быть сформированы и переданы сервисному классу заранее. Сервисный класс ничего не знает относительно действий, которые должны быть выполнены над данными.

Листинг 23.2 показывает, как формируется отчет. Данные инкапсулированы в `AbstractTableModel` (абстрактная модель таблицы) и передаются сервисному классу по имени `PTableWindow` — строка 499. Сначала требуется выяснить, хочет ли пользователь сформировать отчет для конкретной даты (то есть, заполнено ли поле `dayTxt` — текст дня, строки 483–488). Детали отчета извлекаются на данную дату или, в случае ошибки, на текущую дату (строка 487). Точно так же нужно знать, должен ли отчет быть сформирован для всех возможных деловых партнеров или только для одного конкретного делового партнера (строки 489–496).

Если пользователь указал, что отчет для конкретной даты без задания идентификатора (id) делового партнера, то извлекаются все детали отчета для указанной даты (строка 492). Если пользователем задается идентификатор делового партнера как условие формирования отчета, то программа должна извлечь `ContactID` (идентификатор делового партнера). Однако `rptContactCmb` возвращает имя делового партнера, а не его идентификатор (строка 489) (см. метод `populateReportContact()` в листинге 23.1, строка 56). Имя делового партнера должно быть преобразовано в его идентификатор, потому что метод в `CAdmin` требует именно его (строки 494–495).

`AbstractTableModel` формируется из списка деталей отчета путем вызова `populateReportTable()` (строка 498). Эта модель передается экземпляру `PTableWindow` вместе с соответствующим текстом заголовка и другими параметрами. Строки символов `Print` — печать и `Close` — закрыть (строка 499) должны сообщить объекту `PTableWindow`, что нужно создать две кнопки с данными названиями. Эта операция показывает, как `PTableWindow` может динамически создавать кнопки.

Нет смысла формировать кнопку без указания соответствующего средства приема ее действий. Строки 502–516 определяют два таких средства, подключенные к двум кнопкам, созданным объектом `PTableWindow`. Первое средство (строки 502–511) контролирует команду печати. Второе (строки 512–516) просто закрывает экземпляр `PTableWindow`.

Так как выполнение печати может потребовать много времени, желательно инкапсулировать такую операцию в свой собственный поток. Это удалит из потока печати бремя необходимости ожидать команду печати. Обратите внимание, что поток печати — точно такой же поток, который используется при вызове метода `actionPerformed()` (выполненное действие) в случае возникновения события при нажатии мыши.

Листинг 23.2. Метод reportBtnActionPerformed() в PWindow для отображения окна отчета

Метод reportBtnActionPerformed() в PWindow для отображения окна отчета

```
480: private void reportBtnActionPerformed(java.awt.event.
                                                ActionEvent evt) {
481:       Collection reports = null;
482:       java.sql.Date day = null;
483:       try{
484:          java.util.Date d = java.text.DateFormat.
                                              getDateInstance(
           java.text.DateFormat.SHORT).parse(dayTxt.getText());
485:          day = new java.sql.Date(d.getTime());
486:       }catch(Exception exc){
487:       day = new java.sql.Date(System.currentTimeMillis());
488:       }
489:       String contactID = rptContactCmb.getSelectedItem().
                                                    toString();
490:       if(contactID != null && contactID.trim().
                           length() == 0) contactID = null;
491:
492:       if(contactID == null) reports =
                                        admin.getReport(day);
493:       else{
494:          contactID = getContactIDFromName(contactID);
495:          reports = admin.getReport(day,contactID);
496:       }
497:
498:    AbstractTableModel md = populateReportTable(reports);
499:    final PTableWindow win = new PTableWindow(this,true,md,
                           "Activity Report",
                  "Report generated "+day+
                     (contactID==null?"":("on "+contactID)),
                  new String[]{"Print","Close"});
500:
502:       win.addActionListener(0, new java.awt.event.
                                          ActionListener(){
503:        public void actionPerformed(java.awt.event.
                                          ActionEvent evt){
504:             //чтобы не завязнуть в потоке печати
505:             (new Thread(){
506:                public void run(){
507:                   print(win);
508:                }
509:             }).start();
510:          }
```

```
511:        });
512:        win.addActionListener(1, new java.awt.event.
                                                ActionListener(){
513:           public void actionPerformed(java.awt.event.
                                                ActionEvent evt){
514:              win.close();
515:           }
516:        });
517:        win.show();
518: }
```

Отчет о деятельности

Отчет о деятельности (называемый ежедневным итоговым отчетом) показан на рис. 23.5. Отчет представляет число исходящих сообщений, созданных, скорректированных и посланных служащими в конкретный день. Вычисления выполнены хранимой процедурой, представленной в листинге 23.44. Реализация дает отображение отчета, который рассмотрен ранее в разделе 23.3.1.

Другое новое окно в итерации 3, предназначенное для отображения прав доступа, было рассмотрено в главе 20 и показано на рис. 20.11.

Рис. 23.5. Окно ежедневного отчета

Печать отчета

Логика, обеспечивающая распечатку отчета, размещена в PTableWindow. Печать может быть выполнена на данных, извлеченных из GUI-компонентов, либо альтернативно логика печати может просто использовать некоторые особенности GUI-компонентов, чтобы выполнить печать. Второй подход — это принцип WYSIWYG (what you see is what you get — что ты видишь на экране, то и получишь при печати). Этот подход основан на логике рисования для выполнения работы. Итерация 3 реализует смесь первого и второго подходов. Это объясняется в разделе 23.3.2.

Листинг 23.3 иллюстрирует, как логика печати размещается в PTableWindow и поэтому освобождает объект PWindow от какого-либо знания логики печати. PWindow никак не влияет на размещение или содержимое печати.

Листинг 23.3. Метод print() в PWindow

Метод print() в PWindow

```
520:    private void print(Printable target){
521:       PrinterJob pj = PrinterJob.getPrinterJob();
522:       pj.setPrintable(target);
523:       if(pj.printDialog()) {
524:          try{
525:             pj.print();
526:          }catch(Exception printEx){
527:             printEx.printStackTrace();
528:          }
529:       }
530:    }
```

Объект PTableWindow, который реализует интерфейс Printable (печатаемый), загружается в параметр метода print() (печатать) как объект target (цель) — строка 520. Работа печати запрашивается из PrinterJob (работа принтера) (строка 521). Работа принтера в основном представляет работу, предлагаемую сервису печати, который содержит инструкции аппаратным средствам принтера. Печатаемый объект target (PTableWindow) затем регистрируется как экземпляр работы принтера (строка 522). Это указывает, что печатаемый объект target (цель) полностью ответствен за обеспечение всей информации относительно того, что нужно печатать.

Прежде чем печать будет выполнена, пользователю дают возможность выбрать формат печати (строка 523). Это сделано с помощью вывода диалогового окна печати, в котором пользователь может определить местоположение принтера, формат бумаги и т. д.

Заполнение таблицы отчета

Данные строк отчета, извлеченные из CAdmin (листинг 23.2, строка 498), должны быть преобразованы в формат, подходящий для отображения на экране с

помощью PTableWindow. Это делает метод populateTableModel() (заполнение модели таблицы) — листинг 23.4. Метод начинается с создания внутреннего класса DefaultTableModel (модель таблицы, используемая по умолчанию). Этот класс обеспечивает все необходимые реализации AbstractTableModel. Редактирование в таблице DailyReport (ежедневный отчет) должно быть ограничено, так как нет никакого смысла в разрешении пользователю редактировать таблицу (строки 552–554).

Соответствующие столбцы таблицы DailyReport регистрируются в строках 557–561. Наконец, модель таблицы заполняется соответствующим содержимым, извлеченным из данного списка IAReportEntry — интерфейс «вхождение в отчет» пакета acquaintance (строки 562–567).

Листинг 23.4. Метод populateTableModel() в PWindow

Метод populateTableModel() в PWindow

```
550:    private AbstractTableModel populateReportTable
                                        (Collection reports){
551:       DefaultTableModel md = new DefaultTableModel(){
552:          public boolean isCellEditable(int row, int col){
553:             return false;
554:          }
555:       };
556:
557:       md.addColumn("Contact");
558:       md.addColumn("Email Created");
559:       md.addColumn("Email Sent");
560:       md.addColumn("Email Outstanding");
561:       md.addColumn("Email Past Outstanding");
562:       Iterator it = reports.iterator();
563:       while(it.hasNext()){
564:          IAReportEntry re = (IAReportEntry) it.next();
565:          String contactName = re.getContact().
                getFamilyName() +
                ", "+re.getContact().getFirstName();
566:          md.addRow(new Object[]{
                contactName,
             ""+re.getEmailCreated(),
             ""+re.getEmailSent(),
             ""+re.getEmailOutstanding(),
             ""+re.getPastOutstanding()});
567:       }
568:       return md;
569:    }
```

Отображение окна авторизации

Окно авторизации использует тот же самый принцип, что и формирование окна отчета. Он использует `PTableWindow`, чтобы отобразить права доступа. Это демонстрирует полезность `PTableWindow` как сервисного класса. С небольшими модификациями конструирования данных может использоваться тот же самый класс `PTableWindow`, чтобы показать содержимое различных окон.

Листинг 23.5 показывает, как конструируется окно авторизации. Изменять права доступа может только администратор системы (служащий ADM-отдела). Это разрешение модификации помещено в БД и доступно для `canEditAuthorizationRules()` (можно редактировать права доступа) из класса `CAdmin` (строка 617). Права доступа извлекаются из `CAdmin` через метод `getAuthorizationRules()` — получить права доступа (строка 618). Затем они возвращаются как отображение между «from_department» (исходный отдел) и «target_department» (целевой отдел) — раздел 19.2.3, таблица 19.1. Извлеченные права доступа преобразуются в структуру данных, которая может быть понятна для `PTableWindow`. Это преобразование выполняется при обращении к методу `constructAuthorizationRules()` — конструировать права доступа (строка 619).

Листинг 23.5. Метод authorizationBtnActionPerformed() в PWindow

```
Метод authorizationBtnActionPerformed() в PWindow

616:    private void authorizationBtnActionPerformed
                                          (ActionEvent evt) {
617:       boolean canEdit = admin.canEditAuthorizationRules();
618:       Map authorization = admin.getAuthorizationRules();
619:       final DefaultTableModel md =
            constructAuthorizationRules(canEdit,authorization);
620:       String[] buttons = null;
621:       if(canEdit) buttons = new String[]{"Save","Close"};
622:       else buttons = new String[]{"Close"};
623:
624:       final PTableWindow win =
              new PTableWindow(this,true,md,
              "Authorization Rules",
              "Your department: "+admin.getEmployee().
              getDepartmentCode(), buttons);
625:       int closeIndex = 0;
626:       if(canEdit){
627:          closeIndex = 1;
628:          win.addActionListener(0, new java.awt.event.
                                          ActionListener(){
629:             public void actionPerformed(ActionEvent evt){
630:                (new Thread(){
631:                   public void run(){
```

```
632:                        saveAuthorizationRules(md);
633:                     }
634:                  }).start();
635:               }
636:            });
637:         }
638:         win.addActionListener(closeIndex,
                                      new ActionListener(){
639:            public void actionPerformed(ActionEvent evt){
640:               win.close();
641:            }
642:         });
645:         win.setDefaultRenderer(Object.class,
             new DefaultTableCellRenderer(){
646:            public Component
                           getTableCellRendererComponent(JTable
               table,
647:            Object value,
648:            boolean isSelected,
649:            boolean hasFocus,
650:            int row,
651:            int column){
652:               if(value != null){
653:                 javax.swing.JCheckBox ch = null;
654:                 if(value.toString().equals("true"))
                       ch = new javax.swing.JCheckBox("", true);
655:                 else if(value.toString().equals("false"))
                       ch = new javax.swing.JCheckBox("",false);
656:
657:                 if(ch == null)
                      return super.getTableCellRendererComponent(
                    table,value,isSelected,hasFocus,row,column);
658:
660:                  if(isSelected)
                        ch.setBackground(java.awt.Color.BLUE);
661:                 return ch;
662:               }else
                    return super.getTableCellRendererComponent(
                    table,value,isSelected,hasFocus,row,column);
663:            }
664:         });
665:
668:         win.setDefaultEditor(Object.class,
               new DefaultCellEditor(new JCheckBox()));
669:
670:         win.show();
671:      }
```

Строки 620–624 конструируют PTableWindow для целей окна авторизации (это подобно подходу, используемому для окна отчета — раздел 23.3.1, листинг 23.2). Вместо кнопки Print окно авторизации имеет кнопку Save (сохранить) — строка 621. Кнопка Save видима только в случае, если пользователь был идентифицирован как сотрудник ADM-отдела. Если служащему не разрешено изменять правила, кнопка Save невидима и доступна только кнопка Close (строки 621–622).

Права доступа сохраняются через обращение к методу saveAuthorizationRules() — сохранить права доступа (строка 632). Этот метод вызывается внутри потока, чтобы обеспечить более понятный отклик пользователя.

Настройка PTableWindow, связанная с отображением матрицы авторизаций, выполняется после создания модели таблицы точно так же, как сделано при реализации окна отчета. В таблице необходимо отобразить более сложный GUI-компонент, потому что окно авторизации содержит не только простой вводимый текст. Это достигается переопределением метода предоставления (отображения). PTableWindow имеет такой метод, называемый setDefaultRenderer (задать средство визуального отображения по умолчанию). Код клиента должен определить тип класса, для которого требуется настройка предоставления, и обеспечить средство визуального отображения для этого типа (строка 645).

Визуальное отображение таблицы выполняется с помощью JTable (класс «таблица» пакета Swing), извлекая компонент, ответственный за это отображение содержимого JTable, из его списка TableCellRenderers — средства визуального отображения ячеек таблицы (вызывая метод getTableCellRendererComponent() — получение средства визуального отображения ячейки таблицы, находящийся в объекте TableCellRenderer — средство визуального отображения ячейки таблицы, строка 646). Итерация 3 отображает набор выключателей в качестве представления матрицы авторизаций, и это достигается возвращением экземпляра JCheckBox — класс «выключатель» пакета Swing (строки 653–655). Выключатели будут создаваться только в том случае, если содержимое, которое будет отображено, имеет величину true/false (истина/ложь). Если величина отлична от true/false, то применяется средство визуального отображения, используемое по умолчанию и зарегистрированное в JTable (строки 657 и 662). Визуальное улучшение обеспечивается также тем, что выбранный выключатель окрашивается в синий цвет (строка 660).

Поскольку выключатели используются как содержимое окна таблицы, нужно также зарегистрировать редактор в случае, если пользователь должен изменять значения выключателей (строка 668). Как упомянуто ранее, редактирование может быть выполнено только пользователем из ADM-отдела.

Преобразование из матрицы правил в таблицу авторизации

Права доступа, извлеченные из CAdmin, находятся в формате отображения между «from_department» и «target_department». Необходимо преобразовать это представление в представление, понятное для объекта PTableWindow,

которым является AbstractTableModel. Это преобразование выполняет метод constructAuthorizationRules() — сформировать правила авторизации (листинг 23.6).

Построенное окно авторизации сильно напоминает представление матрицы авторизаций (раздел 19.2.3, таблица 19.1). Это означает, что таблица не редактируема в столбце 0 (строка 576). Она имеет крайний левый столбец и заголовок, чтобы отобразить список отделов (строки 580–589).

Как только конструирование расположения таблицы будет закончено, таблица готова к заполнению данными, содержащимися в коллекции (строки 592–609). Алгоритм строит массив строк, представляющих строки в таблице (строка 596). Каждая строка содержит одно и то же число столбцов, все из которых должны иметь значение true или false. Значение true представляет непосредственное отображение между «from_department» (левая сторона таблицы) и «target_department» (заголовок таблицы). Все строки первоначально устанавливаются в false, чтобы указать, что нет никакого подобного отображения (строка 598). Значение true помещается в массив, если определено такое отображение (строки 599–607, особенно строки 601 и 606). Наконец, строка помещается как строка таблицы через метод addRow() (добавить строку) в строке 608.

Сохранение измененных прав доступа

Когда происходит изменение прав доступа и выполняется метод saveAuthorizationRules() — сохранение прав доступа (листинг 23.5, строка 632), предпринимается попытка сохранить это изменение, как показано в листинге 23.7. Пользователю позволено выполнить сохранение несколько раз, если предыдущие попытки закончились ошибкой (строка 676 и строки 683–684).

Права доступа, размещенные в измененном объекте AbstractTableModel, должны быть преобразованы обратно к представлению, понятному для CAdmin. Листинг 23.6 преобразовывает отображение (которое возвращено к CAdmin) в AbstractTableModel, а листинг 23.7 выполняет обратное преобразование. Преобразование выполняется методом extractAuthorizationRules() — выделить права доступа (строка 679). Преобразованное Map (отображение) передается методу saveAuthorizationRules объекта CAdmin для дальнейшей обработки. Этот метод может сформировать исключение, чтобы указать, что или сохранение не удалось, или пользователю не разрешено изменять права доступа. В любом случае пользователь имеет возможность продолжить попытку сохранить измененные величины (строки 683–684).

Преобразование из таблицы авторизации в матрицу правил

Как только что было объяснено, величины, размещенные в таблице авторизации (AbstractTableModel), должны быть преобразованы в величины, понятные для CAdmin(Map). Это преобразование выполняется методом extractAuthorizationRules() (выделить права доступа), который показан в листинге 23.8.

Листинг 23.6. Метод constructAuthorizationRules() в PWindow

Метод constructAuthorizationRules() в PWindow

```
573:    private DefaultTableModel constructAuthorizationRules(
           final boolean canEdit, java.util.Map authorization){
574:       DefaultTableModel md = new DefaultTableModel(){
575:          public boolean isCellEditable(int row, int col){
576:             if(col == 0) return false;
577:             return canEdit;
578:          }
579:       };
580:       md.addColumn("Department");
581:
582:       //формирование левой и заглавной частей таблицы
583:       Iterator it = authorization.entrySet().iterator();
584:       List left = new LinkedList();
585:       while(it.hasNext()) {
586:          Map.Entry e = (Map.Entry) it.next();
587:          md.addColumn(e.getKey());
588:          left.add(e.getKey());
589:       }
590:       //заполнение данными
591:       it = authorization.entrySet().iterator();
592:       while(it.hasNext()){
593:          Map.Entry e = (Map.Entry) it.next();
594:          List l = (List) e.getValue();
595:          Iterator it2 = l.iterator();
596:          Object row[] = new Object[1+left.size()];
597:          row[0] = e.getKey();
598:          for(int i=left.size();i>0;i--) row[i] =
                                        new Boolean(false);
599:          while(it2.hasNext()){
600:             String match = it2.next().toString();
601:             int index = left.indexOf(match);
602:             if(index == -1) {
603:                System.out.println("WRONG value "+match+
                      " from "+row[0]);
604:                continue;
605:             }
606:             row[index+1] = new Boolean(true);
607:          }
608:          md.addRow(row);
609:       }
610:       return md;
611:    }
```

Листинг 23.7. Метод saveAuthorizationRules() в PWindow

Метод saveAuthorizationRules() в PWindow

```
674:    private void saveAuthorizationRules(AbstractTableModel
                                                          md) {
675:       boolean tryAgain = true;
676:       while(tryAgain){
677:          tryAgain = false;
678:          try{
679:             java.util.Map extracted =
                                 extractAuthorizationRules(md);
680:             admin.saveAuthorizationRules(extracted);
681:          }catch(Exception exc){
683:             int ok = JOptionPane.showConfirmDialog(this,
                 "Error in saving modification, would you like
                  to try again?",
              "Error saving Authorization",
              JOptionPane.OK_CANCEL_OPTION);
684:          if(ok == JOptionPane.OK_OPTION) tryAgain = true;
685:          }
686:       }
687:    }
```

Проходится каждая строка в таблице, и в коллекцию добавляется соответствующая ей пара ключевых величин (строка 695). Однако в коллекцию авторизации должны быть вставлены только заданные значения таблицы. Для этой цели таблица опять должна быть просмотрена, и лишь значение «true» должно привести к добавлению соответствующей пары ключевых величин (строки 697–702). Пара ключевых величин получается из имени столбца таблицы (левой стороны таблицы и ее заголовка), как показано в строках 694 и 699 соответственно.

Удаление исходящего сообщения

Итерация 3 дает возможность служащему удалить исходящее сообщение в соответствии с его/ее разрешением, заданным матрицей авторизации (раздел 19.2.3, подпоток S6). Это делается с помощью объекта `PMessageDetailWindow` (класс «окно деталей сообщения» пакета `presentation`), отображающего содержимое исходящего сообщения и в то же самое время позволяющего пользователю редактировать или удалять исходящее сообщение (листинг 23.9, строка 718).

Листинг 23.8. Метод extractAuthorizationRules() в PWindow

Метод extractAuthorizationRules() в PWindow

```
690:  private java.util.Map extractAuthorizationRules
                                          (TableModel model){
691:       java.util.Map authorization = new HashMap();
692:       int rows = model.getRowCount();
693:       for(int i=0;i<rows;i++){
694:          Object key = model.getValueAt(i, 0);
695:          authorization.put(key, new LinkedList());
696:          for(int j=1;j<rows+1;j++){
697:             if(model.getValueAt(i, j).toString().
                                        equals("true")){
698:              LinkedList list = (LinkedList) authorization.
                                              get(key);
699:                list.add(model.getColumnName(j));
700:                authorization.put(key, list);
701:                System.out.println(key+" --> "+model.
                                    getColumnName(j));
702:             }
703:          }
704:       }
705:       return authorization;
706:    }
```

Листинг 23.9. Метод deleteBtnActionPerformed() в PWindow

Метод deleteBtnActionPerformed() в PWindow

```
711: private void deleteBtnActionPerformed(ActionEvent evt) {
712:    int []rows=viewTable.getSelectedRows();
713:    if(rows.length == 0) return;
714:    boolean needRefreshing = false;
715:
716:    //подтверждение удаления одного сообщения за раз
717:    for(int i=0;i<rows.length;i++){
718:       PMessageDetailWindow msgWin =
        new PMessageDetailWindow(this,true, admin,false);
719:       msgWin.setTarget((IAOutMessage)model.
                                  getRawObject(rows[i]));
720:          msgWin.show();
721:          needRefreshing |= msgWin.hasModified();
722:       }
723:    //обновление после завершения всего, сохранение времени
724:       if(needRefreshing) refreshContent();
725:    }
```

Когда нажимается кнопка Delete (удалить), PMessageDetailWindow отображается для каждого выбранного в таблице исходящего сообщения (строки 712 и 718). Когда исходящее сообщение будет изменено или удалено, таблица обновляется (строки 721 и 724).

Метод setTarget() (задать цель) объекта PMessageDetailWindow предназначен для изменения содержимого исходящего сообщения, отображенного в нем. Это окно является модальным и поэтому не позволяет пользователю переключиться назад на главное окно, пока оно остается активным (строка 718, второй параметр равен true, что указывает на модальность).

Изменение исходящего сообщения

Нет больших различий между удалением исходящего сообщения и изменением исходящего сообщения в реализации метода updateBtnActionPerformed() (выполнить действие кнопки корректировки), показанного в листинге 23.10. Корректировка исходящего сообщения (раздел 19.2.3, подпоток S7) использует PMessageDetailWindow с несколько измененным флагом (строка 734, четвертый параметр имеет значение true).

Листинг 23.10. Метод updateBtnActionPerformed() в PWindow

Метод updateBtnActionPerformed() в PWindow

```
727:   private void updateBtnActionPerformed(ActionEvent evt) {
728:       int []rows=viewTable.getSelectedRows();
729:       if(rows.length == 0) return;
730:
731:       boolean needRefreshing = false;
732:       //показать содержимое каждого выделенного сообщения
733:       for(int i=0;i<rows.length;i++){
734:         PMessageDetailWindow msgWin =
              new PMessageDetailWindow(this,true, admin,true);
735:          msgWin.setTarget((IAOutMessage)model.
                 getRawObject(rows[i]));
736:          msgWin.show();
737:          needRefreshing |= msgWin.hasModified();
738:       }
739:    //обновление после завершения всего, сохранение времени
740:       if(needRefreshing) refreshContent();
741:    }
```

Четвертый параметр определяет, является ли исходящее сообщение, показанное в PMessageDetailWindow, редактируемым. Только измененные исходящие сообщения приведут к изменению таблицы исходящих сообщений (строки 737 и 740).

Создание исходящего сообщения

Создание нового исходящего сообщения основывается на `PMessageDetailWindow` как удобном классе, который может отобразить содержимое исходящего сообщения. Листинг 23.11 показывает, что `PMessageDetailWindow` для этой цели создан аналогично действию редактирования в листинге 23.10. Отличие состоит в том, что класс `PMessageDetailWindow`, создающий новое сообщение, не имеет используемого ранее исходящего сообщения в качестве своего содержимого, как это показано в листинге 23.10, строка 735 (метод `setTarget()`). Вместо этого для данного случая формируется пустое исходящее сообщение с помощью вызова `showNewMessage()` — показать новое сообщение (строка 840).

Листинг 23.11. Метод createNewMsgBtnActionPerformed() в PWindow

Метод createNewMsgBtnActionPerformed() в PWindow

```
837:    private void createNewMsgBtnActionPerformed
                                              (ActionEvent evt) {
838:       PMessageDetailWindow msgWin =
              new PMessageDetailWindow(this,true, admin,true);
839:       msgWin.setTitle("Create New Message");
840:       msgWin.showNewMessage();
841:       msgWin.show();
842:       if(msgWin.hasModified()) refreshContent();
843:    }
```

Таблица исходящих сообщений в основном окне будет обновлена, только если добавление произойдет успешно (строка 842).

23.3.2. Класс PTableWindow

`PTableWindow` (класс «окно таблицы» пакета `presentation`) служит **сервисным классом** для отображения формируемой таблицы в окне. Этот класс способен динамически формировать ряд кнопок. Каждая кнопка может иметь много приемников.

Листинг 23.12 показывает резюме методов для этого класса. Этот класс сообщает клиенту относительно состояния кнопки, которая нажата с помощью его метода `getReturnStatus()` (получить возвращаемое состояние). Возможность печати обеспечивается методом `print()` (печатать). Печать в этом классе выполняется отображением содержимого таблицы в простой формат таблицы для производства печати. Различные редакторы и средства визуального отображения, используемые по умолчанию для таблицы, могут быть изменены путем вызова `setDefaultEditor()` (задать редактор по умолчанию) и `setDefaultRenderer()` (задать средство визуального отображения по умолчанию).

Поскольку этот класс способен запомнить состояние действий пользователя, обеспечивается метод ручного расположения окна, чтобы позволить кли-

енту управлять выбором расположения окна (через метод `setDisposeWhenClose()` — задать расположение при закрытом окне).

Динамическая регистрация кнопок

`PTableWindow` может обеспечить **динамическое размещение кнопок**. Это сделано в его конструкторе, как показано в листинге 23.13. Строки 33–39 показывают стандартную инициализацию свойств. Когда никакая кнопка не нужна для окна, по умолчанию автоматически создается кнопка `Close` — закрыть (строки 41–50). Однако если в параметре задается несколько текстов с именами кнопок, то динамически формируются кнопки так, как показано в строках 53–66.

Каждая создаваемая кнопка должна иметь метку, которая указывается параметром `buttons` — кнопки (строка 53). Чтобы различать кнопки, используется команда операции (строка 58). Она используется, чтобы определить, какая кнопка нажата; эта команда также используется для регистрации приемника с помощью метода `addActionListener()` — добавить приемник операции (будет объяснено позже) и для получения значения возвращаемого состояния метода `getReturnStatus()` — получение возвращаемого состояния (также будет объяснено позже).

Листинг 23.12. Резюме методов класса PTableWindow

```
void          addActionListener(int buttonIndex, java.awt.event.
              ActionListener listener)
                  Разрешение клиентам добавлять действия к конкретной кнопке.
void          close()
                  Разрешение клиенту закрыть окно в любой момент времени.
int           getReturnStatus()
                  Возвращает полученное в этом окне состояние (если окно должно
                  быть закрыто).
static void   Main(java.lang.String[] args)
                  Тестирование этого окна.
int           print(java.awt.Graphics graphics,
              java.awt.print.PageFormat pageFormat, int pageIndex)
                  Печать окна так, как его видит пользователь.
void          setDefaultEditor(java.lang.Class c,
              javax.swing.table.TableCellEditor editor)
                  Замена редактора для компонента таблицы, отображаемого в этом
                  окне.
void          setDefaultRenderer(java.lang.Class c,
              javax.swing.table.TableCellRenderer renderer)
                  Замена средства визуального отображения для компонента таблицы,
                  отображаемого в этом окне.
void          setDisposeWhenClose(boolean dispose)
                  Позволяет клиенту указать, нужно ли отобразить автоматически
                  окно, когда оно больше не нужно.
```

Листинг 23.13. Конструктор класса PTableWindow

Конструктор класса PTableWindow

```
33:    public PTableWindow(Frame parent, boolean modal,
         AbstractTableModel model,String title,String subtitle,
           String[] buttons) {
34:           super(parent, modal);
35:           this.model = model;
36:           initComponents();
37:           titleLbl.setText(title);
38:           subtitleLbl.setText(subtitle);
39:
40:  //кнопки не заданы, по умолчанию используется кнопка Close
41:           if(buttons == null){
42:              javax.swing.JButton btn = new javax.swing.
                                              JButton("Close");
43:              btn.setToolTipText("Close this window");
44:              btn.addActionListener(new ActionListener() {
45:              public void actionPerformed(ActionEvent evt) {
46:                 doClose(0);
47:              }
48:           });
49:           buttonPanel.add(btn);
50:           return;
51:        }else{ //динамическое формирование кнопок
52:           for(int i=0;i<buttons.length;i++){
53:              javax.swing.JButton btn = new JButton
                                                 (buttons[i]);
58:              btn.setActionCommand(""+i);
59:              btn.addActionListener(new java.awt.event.
                                             ActionListener() {
60:              public void actionPerformed(ActionEvent evt) {
61:                    doClose(Integer.parseInt(evt.
                                    getActionCommand()));
62:                 }
63:              });
64:              buttonPanel.add(btn);
65:           }
66:        }
67:     }
```

Добавление приемников к динамически сформированным кнопкам

Поскольку кнопки добавляются к `PTableWindow` динамически, подключение **приемников** после формирования кнопок становится интересной проблемой. Вспомните, что кнопки создаются динамически и трудно опреде-

лить, к какой кнопке обратиться для надлежащей регистрации приемника. Кроме того, PTableWindow не возвращает список кнопок, которые были созданы, так как это невозможно в его конструкторе.

Однако решение этой проблемы оказывается весьма простым. Все кнопки должны быть помещены в контейнер, к которому можно обращаться, учитывая порядок создания этих кнопок. Таким образом, первая кнопка в списке представляет первый индекс параметра buttons в аргументе конструктора. Листинг 23.14 показывает реализацию этой идеи. Созданные кнопки (листинг 23.13, строка 64) добавляются в единственный контейнер (buttonPanel — панель кнопок). Метод addActionListener() требует индекса кнопки, размещенного в контейнере (листинг 23.14, строка 76).

Метод getComponent(index) (получить компонент) из JPanel — класс «панель» пакета Swing (контейнера) используется, чтобы извлечь кнопку по данному индексу.

Листинг 23.14. Метод addActionListener() в PTableWindow

Метод addActionListener() в PTableWindow

```
74:     public void addActionListener(int buttonIndex,
                java.awt.event.ActionListener listener){
75:         try{
76:            ((javax.swing.JButton)buttonPanel.
                                    getComponent(buttonIndex)).
                  addActionListener(listener);
77:         }catch(Exception exc){}
78:     }
```

Возвращаемое состояние кнопки

Клиенты должны быть информированы относительно того, какая кнопка нажата на PTableWindow. Поскольку кнопки создаются динамически, имеются два явных результата, которые могут быть возвращены с помощью метода getReturnStatus(). Может быть возвращен индекс нажатой кнопки. С другой стороны, может быть возвращен текст кнопки. Оба подхода позволяют клиенту знать, какая кнопка нажата, и соответствующим образом ответить на событие.

Листинг 23.15 показывает удивительно простую реализацию первого подхода. Возвращаемое состояние регистрируется через параметр, данный в doClose() — закрыть (листинг 23.13, строка 61). Метод doClose() может или оставить PTableWindow на экране, или закрыть его. Это зависит от значения disposeWhenClosed (отобразить, когда закрывается), установленного методом setDisposeWhenClose(), или от величины возвращаемого состояния. Величина –1 указывает, что окно должно быть закрыто без нажатия какой-либо из его кнопок, но, возможно, нажатием вместо этого приемника окна (типа представленного символом перекрестия в правом верхнем его углу).

Листинг 23.15. Метод doClose() в PTableWindow

Метод doClose() в PTableWindow

```
142:    private void doClose(int retStatus) {
143:       returnStatus = retStatus;
144:       if(disposeWhenClosed || retStatus == -1) close();
145:    }
```

Печать в PTableWindow

`PTableWindow` обеспечивает простой механизм печати. Название окна наряду с заголовком и содержимым таблицы печатаются с помощью метода `print()`, представленного в листинге 23.16. Первоначально графика, обеспечиваемая параметром функции, преобразуется в печатаемую область (строка 182). Как только это будет сделано, названия печатаются с небольшим промежутком между главным названием и подзаголовком (строки 183–185). Названия печатаются с помощью их метода `paint()` (рисовать). Больше места выделяется над заголовком таблицы (строки 186 и 192). Наконец, печатается содержимое таблицы с помощью метода `paint()` объекта `JTable` (класс «таблица» пакета Swing).

Листинг 23.16. Метод print() в PTableWindow

Метод print() в PTableWindow

```
179:    public int print(java.awt.Graphics graphics,
         java.awt.print.PageFormat pageFormat,
         int pageIndex) throws java.awt.print.PrinterException
          {
180:       if(pageIndex == 0){
181:          Graphics2D g2 = (Graphics2D) graphics;
182:          g2.translate(pageFormat.getImageableX(),
                 pageFormat.getImageableY());
183:          titleLbl.paint(g2);
184:          g2.translate(0,40); //переместить на 40 пунктов
185:          subtitleLbl.paint(g2);
186:          g2.translate(0,30);
187:
188:       //это должно использоваться для печати всей графики
189:       //paint(g2);
          //использование метода paint для выполнения работы
190:
192:          table.getTableHeader().paint(g2);
193:          g2.translate(0, table.getRowHeight());
194:          table.paint(g2);
195:          return PAGE_EXISTS;
196:       }else return NO_SUCH_PAGE;
198:    }
```

23.4. Пакет Control

Пакет control (управление) состоит из трех классов (рис. 18.9). По сравнению с итерацией 2 итерация 3 изменяет только CAdmin (класс «администратор» пакета control), чтобы удовлетворить дополнительным требованиям (см. главу 19). Другие классы пакета control в итерации 3 не изменены.

Новые методы итерации 3 в CAdmin следующие: canEditAuthorizationRules() (можно редактировать права доступа), getAuthorizationRules() (получить права доступа), saveAuthorization() (сохранить авторизацию), deleteMessage() (удалить сообщение), getReport() (получить отчет) и updateMessage() (скорректировать сообщение) — рис. 23.6. Эти новые методы переадресовывают все запросы к MModerator (класс «координатор» пакета mediator) и поэтому нет никакой необходимости обсуждать их в этом разделе.

23.5. Пакет Entity

Реализация интерфейса IAReportEntry (интерфейс «вхождение в отчет» пакета acquaintance) имеет добавленный класс EReportEntry (класс «вхождение в отчет» пакета entity) к пакету entity (сущность). Класс EIdentityMap (класс «коллекция идентичности объектов» пакета entity) должен был слегка измениться, чтобы обслуживать этот дополнительный класс пакета entity. Имеется набор методов, чтобы управлять размещением и извлечением отчетов. Рис. 23.7 показывает содержимое пакета entity. Фактически имеется другой внутренний класс по имени EIdentityMap.Epair (двойник EIdentityMap), который используется, чтобы хранить пару первичных ключей для преобразования EReportEntry в EIdentityMap. Однако этот класс — приватный класс в EIdentityMap и поэтому не показан в диаграмме.

Как и другие объекты entity, EReportEntry реализует IEObjectID (интерфейс «идентификатор объекта» пакета presentation), и поэтому на него можно ссылаться из EIdentityMap и других классов, которые используют IEObjectID. Ссылки из EIdentityMap к IEObjectID не могут быть сформированы, поскольку EIdentityMap обслуживает коллекцию этих IEObjectID.

23.5.1. Класс EIdentityMap

К EIdentityMap (класс «коллекция идентичности объектов» пакета entity) были добавлены методы, предназначенные поддерживать извлечение и хранение отчетов. Рис. 23.8 показывает дополнительные методы: getReports() (получить отчеты) и getReport() (получить отчет), registerReport() (зарегистрировать отчет) и unregisterReport() (снять регистрацию отчета). Так как EReportEntry (класс «вход в отчет» пакета entity) является новым типом пакета entity (сущность), нужно также ввести коллекцию, называемую rptPKToOID (первичный ключ отчета — в идентификатор объекта), которая преобразует первичный ключ отчета в его идентификатор объекта (object identifier — OID). Этот подход подобен

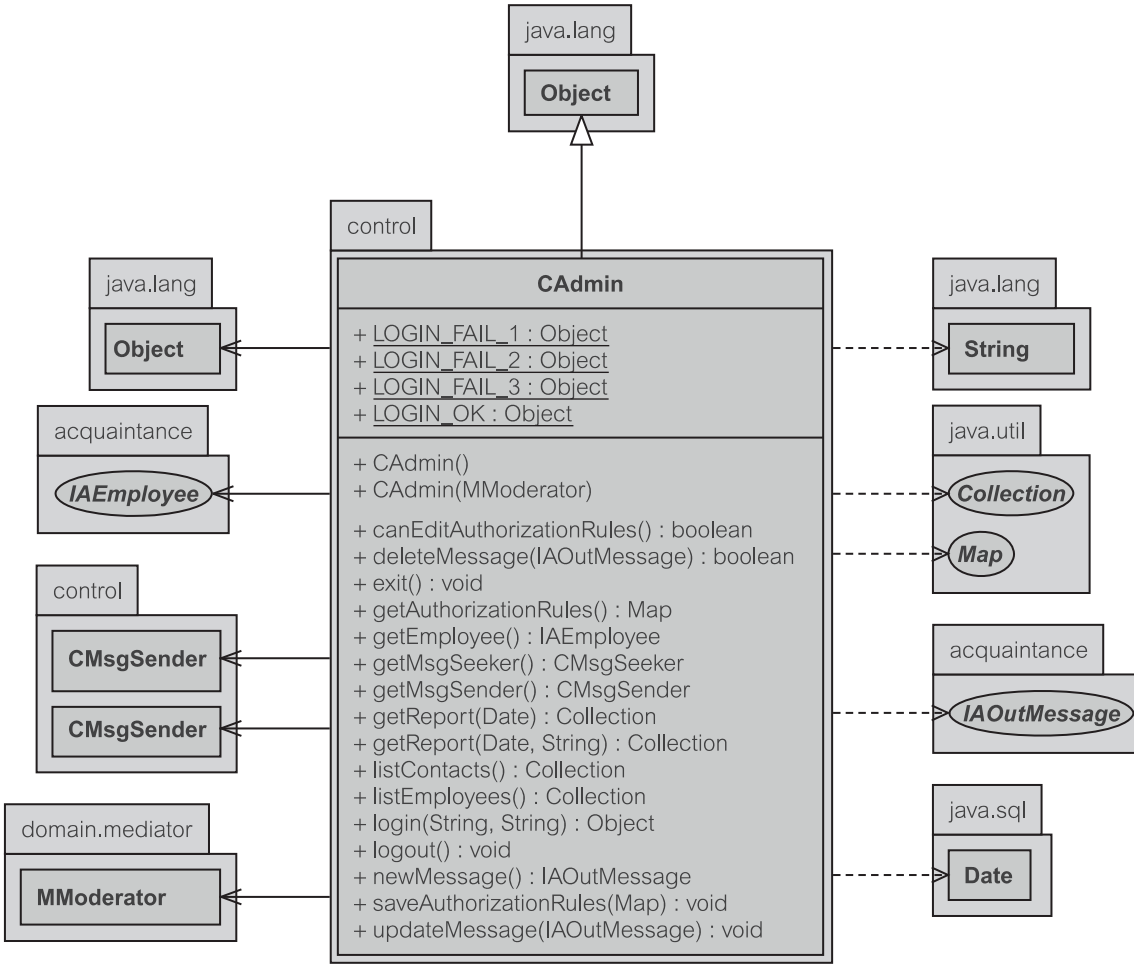


Рис. 23.6. Класс CAdmin в EM 3

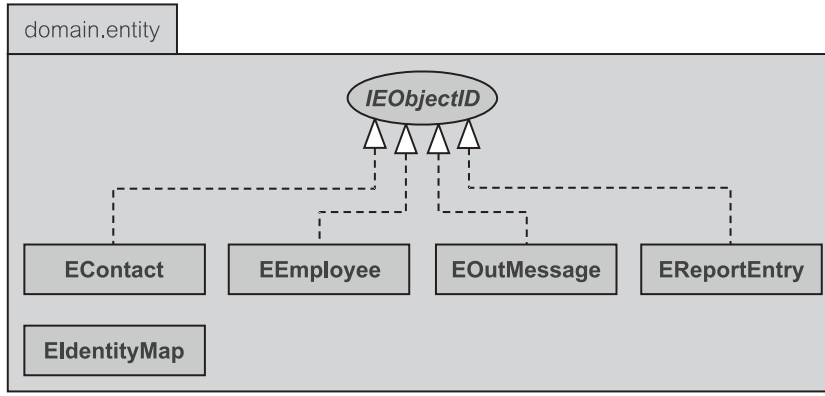


Рис. 23.7. Классы в пакете entity

предыдущим подходам преобразования для исходящих сообщений, деловых партнеров и служащих (раздел 18.5.1, листинг 18.29).

Регистрация и удаление отчета

Метод `registerReport()` (листинг 23.17) получает OID, задаваемый объектом `oidObject` (объект с OID), и помещает его сначала в коллекцию отчетов (строка 92) и затем в `OIDToObj` — OID преобразовать в объект (строка 93). Вхождения в коллекцию отчета осуществляются в форме объекта `EPair` (класс «пара данных» пакета `entity`). Объект `EPair` содержит дату и идентификатор (id) делового партнера, которые определяют условия формирования отчета.

Листинг 23.17. Методы registerReport() и unregisterReport() в EIdentityMap

Методы registerReport() и unregisterReport() в EIdentityMap

```
 89:    public void registerReport(IEObjectID oidObject) {
 90:       Integer oid = new Integer(oidObject.getOID());
 91:       IAReportEntry rpt = (IAReportEntry)oidObject;
 92:       rptPKToOID.put(new EPair(rpt.getReportDay(),
                                    rpt.getContactID()),oid);
 93:       OIDToObj.put(oid, oidObject);
 94:    }
 97:    public void unregisterReport(IEObjectID oidReport) {
 98:       IAReportEntry rpt = (IAReportEntry)oidReport;
 99:       rptPKToOID.remove(new EPair(rpt.getReportDay(),
              rpt.getContactID()));
100:       OIDToObj.remove(new Integer(oidReport.getOID()));
101:    }
```

Метод `unregisterReport()` сформирован вызовом метода `remove()` (удалить) соответствующих коллекций (строки 99 и 100). К тому же, чтобы удалить конкретный отчет из его коллекции отчетов, объект `EPair` должен быть сформирован из даты и идентификатора делового партнера удаляемого отчета и использоваться в качестве ключа в процессе удаления (строка 99).

Извлечение отчета

Имеется ряд методов, которые могут использоваться, чтобы извлечь вхождения отчетов. Список вхождений отчетов в течение конкретного дня может быть извлечен из `EIdentityMap` (кэша) с помощью метода `getReport()` (листинг 23.18). Этот метод получает дату и извлекает все вхождения отчетов для этой даты. Если нет никаких доступных отчетов, то будет возвращена пустая коллекция. Если заданная дата отсутствует, то считается, что клиент хотел бы извлечь все доступные отчеты (строка 38).

Просматривается каждое вхождение в коллекцию отчетов (`rptPKToOID`), и проверяется значение даты — является ли отчет нужной даты (строка 45).

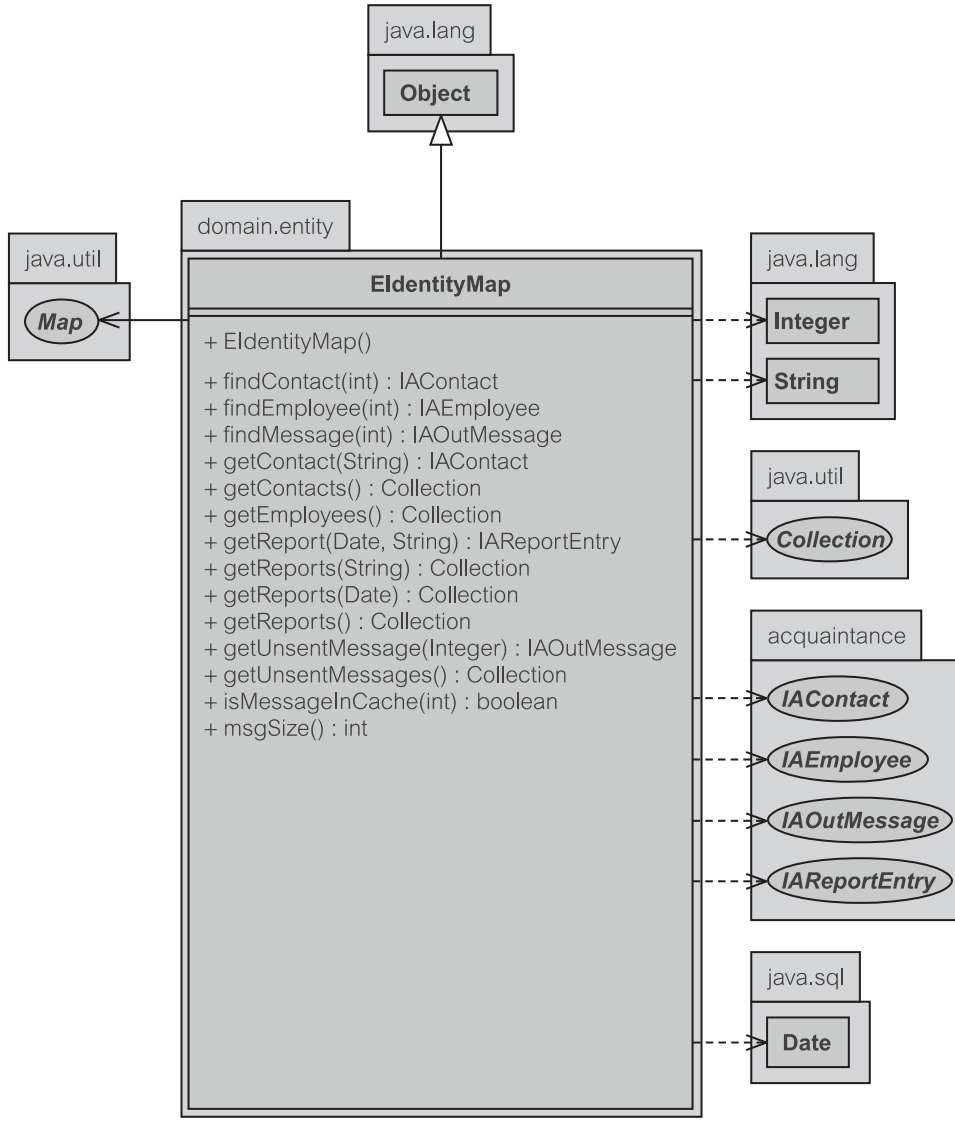


Рис. 23.8. Класс EIdentityMap в EM 3

Если пользователь должен извлечь все доступные отчеты, то может быть вызван метод `getReports()` для выполнения этой операции (листинг 23.19). Метод просто проходит по всем зарегистрированным отчетам в коллекции (строки 53–57) и находит отчет, который имеет соответствующий `OID` — идентификатор объекта (строка 55). Вспомните, что преобразование, поддерживаемое в `rptPKToOID`, — от `EPair` к `OID`. Поэтому возвращаемая величина `next()` в строке 55 — `OID`.

Листинг 23.18. Метод getReport() в EIdentityMap — извлечение по дате

Метод getReport() в EIdentityMap — извлечение по дате

```
37:     public Collection getReport(java.sql.Date day){
38:        if(day == null) return getReports();
39:
40:        Collection c = new ArrayList();
41:        Iterator it = rptPKToOID.entrySet().iterator();
42:        while (it.hasNext()){
43:           Map.Entry entry = (Map.Entry) it.next();
44:           EPair key = (EPair)entry.getKey();
45:           if(day.equals(key.first)) c.add(entry.getValue())
46:        }
47:        return c;
48:     }
```

Листинг 23.19. Метод getReports() в EIdentityMap

Метод getReports() в EIdentityMap

```
51:  public Collection getReports(){
52:    Collection c = new ArrayList();
53:    Iterator it = rptPKToOID.values().iterator();
54:    while (it.hasNext()){
55:      Object o = findReport(((Integer)it.next()).intValue()
56:           if(o != null) c.add(o);
57:        }
58:        return c;
59:     }
```

Когда известен `OID` отчета, можно быстро извлечь отчет. Это выполняет метод `findReport()` (найти отчет) в листинге 23.20. Поскольку коллекция `OIDToObj` обслуживает преобразование между всеми `OID` и связанными с ними объектами, быстрый поиск на основе `OIDToObj` дает соответствующий отчет, если он существует в кэше (строка 77). Данные, размещенные в коллекции, должны быть объектами, и поэтому параметр `reportOID` (идентификатор отчета) должен быть преобразован в тип `Integer` (целое).

Листинг 23.20. Метод findReport() в EIdentityMap

Метод findReport() в EIdentityMap

```
76:     private IAReportEntry findReport(int reportOID){
77:       return (IAReportEntry) OIDToObj.get(new
                                          Integer(reportOID);
78:     }
```

Можно также извлечь вхождения отчетов для конкретного идентификатора (id) делового партнера. Это обеспечивается методом `getReports(contactID)`, показанным в листинге 23.21. Снова, этот метод подобен методу в листинге 23.18 с незначительным различием в том, как выбирается конкретный отчет. Листинг 23.18 сравнивает дату отчета (строка 45), в то время как листинг 23.21 сравнивает идентификатор делового партнера отчета. Когда идентификатор делового партнера отсутствует, считается, что пользователь хочет извлечь все доступные отчеты (листинг 23.19).

Листинг 23.21. Извлечение отчетов для конкретного делового партнера

Метод getReports(contactId) в EIdentityMap

```
62:     public Collection getReports(String contactID){
63:        if(contactID == null) return getReports();
64:
65:        Collection c = new ArrayList();
66:        Iterator it = rptPKToOID.entrySet().iterator();
67:        while (it.hasNext()){
68:           Map.Entry entry = (Map.Entry) it.next();
69:           EPair key = (EPair)entry.getKey();
70:           if(contactID.equals(key.second)) c.add(entry.
                                                    getValue());
71:        }
72:        return c;
73:     }
```

23.6. Пакет Mediator

Многие аспекты пакета `mediator` (посредник) были обсуждены в предыдущих главах. Например, глава 15 рассмотрела паттерны для пакета `mediator`, глава 20 использовала пакет `mediator` (`MDataMapper` — класс «преобразователь данных» пакета `mediator`), чтобы иллюстрировать вызовы хранимой процедуры, а глава 21 представила класс `MUnitOfWork` (класс «единица работы» пакета `mediator`), который может управлять транзакциями и параллелизмом. `MUnitOfWork` — новый класс в итерации 3 (рис. 23.9).

Эволюция пакета `mediator` от итерации 2 к итерации 3 привела к следующим усовершенствованиям в последней:

- Весь доступ к БД осуществляется через набор хранимых процедур/функций.
- Реализация прав доступа, используя преимущества дополнительных триггеров и ограничений БД.
- Управление транзакциями и параллелизмом с помощью `MUnitOfWork` и `MMediator`.
- Реализация формирования ежедневных отчетов в `MMediator`.

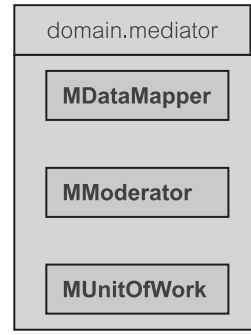


Рис. 23.9. Классы в пакете Mediator системы EM 3

23.6.1. Класс MModerator

Класс `MModerator` — класс «координатор» пакета `mediator` (рис. 23.10) перехватывает все извлечения и модификации объекта сущности. Он переадресовывает эти запросы к `EIdentityMap` (класс «коллекция идентичности объектов» пакета `entity`) или к `MDataMapper` (класс «преобразователь данных» пакета `mediator`) — как потребуется. `MModerator` ответствен за обеспечение того, чтобы операции были выполнены в соответствии с их природой **транзакции**, используя `MUnitOfWork` (класс «единица работы» пакета `mediator`).

Некоторые из дополнительных методов в `MModerator` — это `canEditAuthorizationRules()` (можно редактировать права доступа), `getAuthorizationRules()` (получение прав доступа), `saveAuthorizationRules()` (сохранение прав доступа) и `searchReports` (искать отчеты). Другие методы изменены, чтобы включить особенности транзакций итерации 3.

Права доступа

Листинг 23.22 показывает, как измененные **права доступа** сохраняются в БД. Права доступа могут быть изменены пакетом `presentation` — представление (раздел 23.3.1.8). Метод `saveAuthorizationRules()` в `MModerator` переадресовывает выполнение к методу `saveAuthorizationRules()` объекта `MDataMapper`. Кроме того, он обеспечивает, что выполнение `saveAuthorizationRules()` объекта `MDataMapper` будет инкапсулировано в транзакцию.

Метод `saveAuthorizationRules()` включает автоматическую фиксацию начала транзакции (строка 184) и создает **точку сохранения** (строка 186). Точка сохранения в качестве своей метки имеет случайное число. Это должно устранить появления противоречивых точек сохранения, создаваемых при параллельном доступе к БД. Обратите внимание, что использование точек сохранения вынуждает EM-приложение использовать, как минимум, JDK 1.4. Во время написания кода пакет Oracle не должен полностью реали-

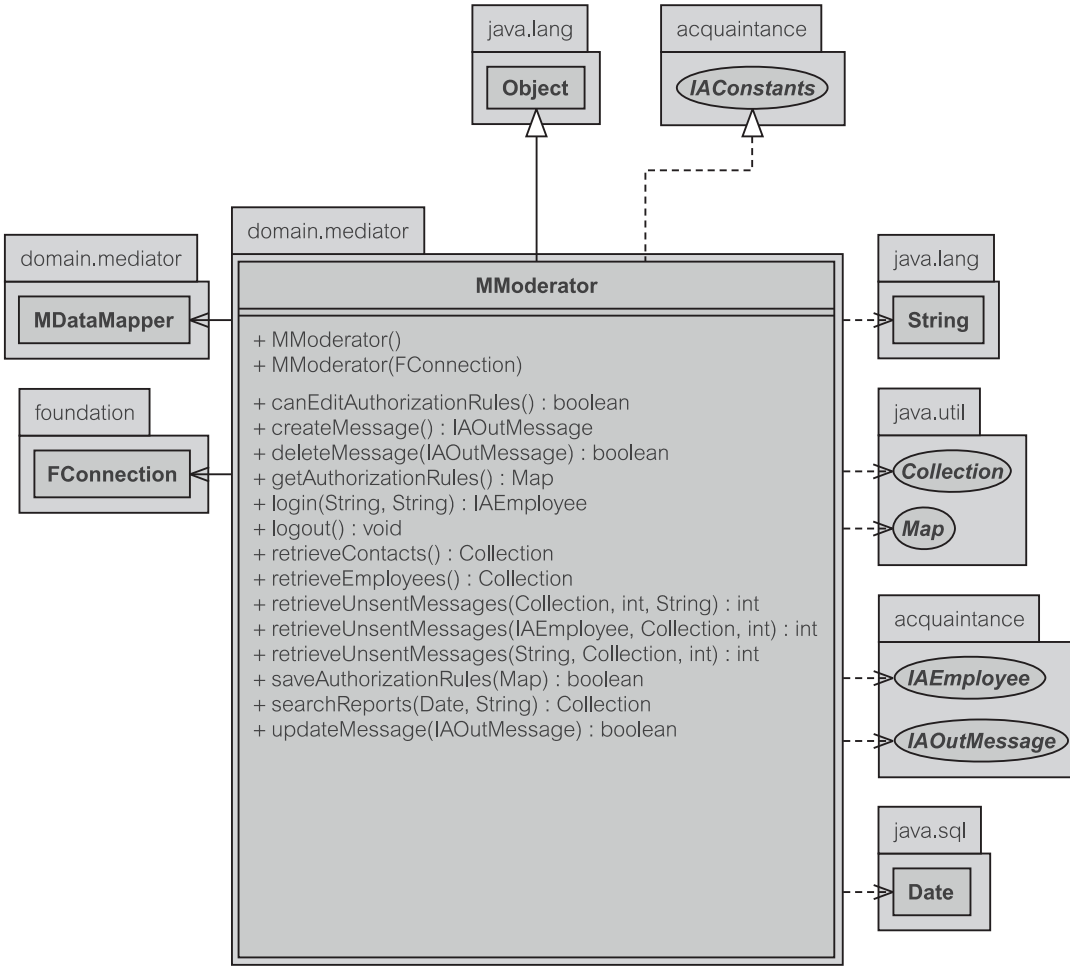


Рис. 23.10. Класс MModerator в ЕМ 3

зовывать спецификации точек сохранения JDK 1.4 в интерфейсе `Connection` (соединение). Однако Oracle обеспечивает свою собственную аналогичную систему задания точек сохранения. Это показано в строке 186.

Как только точка сохранения будет создана, `MModerator` пытается вызвать `saveAuthorizationRules()` объекта `MDataMapper` (строка 188). `MModerator` фиксирует транзакцию, если она будет выполнена успешно (строка 189). Однако если будут исключения, возникшие в течение этого выполнения, произойдет откат транзакции, а точка сохранения освободится (строки 191 и 192). Независимо от того, будет ли транзакция успешной или нет, автоматическая фиксация соединения повторно устанавливается в свое первоначальное состояние (строка 195).

Листинг 23.22. Метод saveAuthorizationRules() в MModerator

Метод saveAuthorizationRules() в MModerator

```
178:    public boolean saveAuthorizationRules(java.util.Map
                                        rules)throws Exception{
179:       //Это может быть помещено в единицу работы, если
              сделано следующее:
180:       //* права доступа перемещены в сам класс
               вместо использования коллекции
181:       //* единица работы изменена так, чтобы управлять
               любыми объектами сущности,
182:       //однако сейчас мы покажем упрощенную версию
183:       boolean origAutoCommit = connection.getAutoCommit();
184:       connection.setAutoCommit(false);
185:       //случайный выбор имени точки сохранения, чтобы
             избежать конфликтов в БД
186:       oracle.jdbc.OracleSavepoint savepoint = connection.
               setSavepoint("AuthorizationRuleSavepoint"+
              (Math.random()*Integer.MAX_VALUE));
187:       try{
188:          mapper.saveAuthorizationRules(rules);
189:          return connection.commit();
190:       }catch(Exception exc){
191:          connection.rollback(savepoint);
192:          connection.releaseSavepoint(savepoint);
193:          throw exc;
194:       }finally{
195:          connection.setAutoCommit(origAutoCommit);
196:       }
197:    }
```

Извлечение отчета

`MModerator` обеспечивает единый метод извлечения отчетов, удовлетворяющих всем допустимым условиям отчета (только дата, только деловой партнер или и то, и другое). Извлечение отчета обеспечивается методом `searchReports()`, который получает дату и идентификатор делового партнера в качестве своих параметров (листинг 23.23). Этот листинг является реализацией структуры, изображенной на рис. 21.10 (раздел 21.3.3).

Метод `searchReports()` проверяет, заполнен ли параметр `day` (день). Если он не заполнен, считается, что пользователь намерен извлечь сегодняшние отчеты (строка 138). Затем метод явно начинает транзакцию, вызывая метод `begin()` (начать) объекта `FConnection` (класс «соединение» пакета `foundation`) — строка 144.

В зависимости от того, задан ли параметр `contactID` (идентификатор делового партнера) или нет, метод будет вызывать различные варианты метода `retrieveReports()` (извлечь отчеты) объекта `MDataMapper`. Если

contactID задан, то очевидно, что пользователь хочет извлечь отчет для конкретного делового партнера на указанную дату (строки 147–152). Если информация о деловом партнере не задана, извлекается список отчетов для всех деловых партнеров на конкретный день.

Листинг 23.23. Метод searchReports() в MModerator

Метод searchReports() в MModerator

```
136:  public Collection searchReports(java.sql.Date day,
      String contactID){
137:  //если day не задан, то сегодняшняя дата
138:  if(day == null)
           day = new java.sql.Date(System.currentTimeMillis());
139:
140:   boolean successful = false;
141:   Collection col = null;
142:   try{
143:      //явное начало транзакции
144:    connection.begin();
147:    if(contactID != null && contactID.trim().length() != 0){
148:         Object rpt = mapper.retrieveReport(day,contactID);
149:            if(rpt == null) col = new LinkedList();
150:            col = java.util.Collections.singleton(rpt);
151:            successful = true;
152:          }else{
153:            col = mapper.retrieveReport(day);
154:            successful = true;
155:          }
156:       }catch(RuntimeException exc){
157:          throw exc;
158:       }catch(Exception exc2){
159:          exc2.printStackTrace(); //отладка
160:       }finally{
161:          if(successful) connection.end();
162:          else{ //удаление всех изменений
163:             connection.rollback();
164:             try{
                  connection.setAutoCommit(true);
               }catch(Exception exc3){exc3.printStackTrace();}
165:          }
166:       }
167:
168:       if(col == null) return new LinkedList();
169:       return col;
170:    }
```

Успешное извлечение гарантирует, что транзакция будет завершена (строка 161). Отказ (исключение) вызывает откат транзакции (строка 163).

Создание исходящего сообщения

Класс `MModerator` обеспечивает и создание исходящего сообщения в **транзакции**. Он использует класс `MUnitOfWork`, чтобы выполнить транзакцию, как показано в листинге 23.24. Прежде чем операция будет выполнена, из `MDataMapper` извлекается новое незаполненное исходящее сообщение. Это создает контейнер, в который данное сообщение может быть помещено.

Листинг 23.24. Метод createMessage() в MModerator

Метод createMessage() в MModerator

```
111:    public IAOutMessage createMessage(){
112:       IAOutMessage msg = mapper.retrieveNewMessage();
113:       MUnitOfWork work = MUnitOfWork.newCurrent();
114:       work.registerNew((IEObjectID) msg);
115:       return msg;
116:    }
```

`MUnitOfWork` инициализируется в строке 113 и изменяет объекты сущности, зарегистрированные в новом объекте `work` — работа (строка 114). Метод `newCurrent()` (новый текущий) в `MUnitOfWork` гарантирует, что новый экземпляр **единицы работы** создан именно для целей этой транзакции.

Метод `registerNew()` (регистрировать новый) указывает единице работы, что исходящее сообщение только что было создано, и поэтому необходимо сделать добавление к БД, когда транзакция будет зафиксирована. Заметьте, однако, что новое исходящее сообщение в это время не зафиксировано. Оно просто отмечено как новое. Возможности фиксации включены в метод `updateMessage()` — корректировка сообщения (будет объяснено позже).

Не фиксируя только что созданные исходящие сообщения, программа может отказаться от исходящего сообщения, если пользователь решит отменить его создание. Это не нарушает целостность БД, так как не было никакого добавления или объявления нового исходящего сообщения в БД.

Корректировка исходящего сообщения

Имеются две ситуации, при которых исходящее сообщение нужно будет корректировать. Изменению может подвергнуться как *существующее*, так и *новое* исходящее сообщение. Листинг 23.25 показывает оба случая в одном методе.

Во время создания нового исходящего сообщения единица работы явно создается методом `newCurrent()` (листинг 23.24, строка 113). Это означает, что вызов `getCurrent()` класса `MUnitOfWork` в строке 101 обеспечи-

вает экземпляр MUnitOfWork, созданный ранее. Однако getCurrent() возвращает null, если нет никакого экземпляра MUnitOfWork, созданного ранее. Это возможно, так как нет обращения к методу createMessage() (создать сообщение), и в действительности пользователь пробует корректировать ранее существующее сообщение. В этом случае должна быть создана новая единица работы (строка 105).

Корректировка исходящего сообщения лишь регистрирует его как *измененное* в MUnitOfWork (строка 106). Это вынуждает единицу работы выполнить корректировку измененной сущности и сохранить ее значение в БД, когда транзакция будет зафиксирована (строка 107).

Листинг 23.25. Метод searchReports() в MModerator

Метод searchReports() в MModerator

```
97:     public boolean updateMessage(IAOutMessage msg) {
101:       MUnitOfWork work = MUnitOfWork.getCurrent();
102:
103:       //work может быть null, если выполняется только
           //корректировка (без создания
104:       //нового исходящего сообщения)
105:       if(work == null) work = MUnitOfWork.newCurrent();
106:       work.registerDirty((IEObjectID)msg);
107:       return work.commit();
108:    }
```

Удаление исходящего сообщения подобно процедуре корректировки, рассмотренной выше, за исключением того, что единица работы всегда создается через его метод newCurrent(). Соответственно, вызов registerDirty() (регистрировать измененный) заменяется вызовом registerRemoved() (регистрировать удаленный).

23.6.2. Класс MDataMapper

В итерации 3 было изменено большинство методов MDataMapper (класс «преобразователь данных» пакета mediator), чтобы обеспечить преобразование всех SQL-запросов в хранимые процедуры или функции. Это рассмотрено в примере главы 20 (листинг 20.3).

Другие усовершенствования, сделанные в MDataMapper, включают рефакторинг извлечения исходящего сообщения. В итерации 2 метод извлечения исходящих сообщений retrieveMessages() (извлечь сообщения) может сформировать SQL-исключение, указывающее на *слишком много открытых курсоров* (too many open cursors). Это происходит только тогда, когда имеется большое количество исходящих сообщений, извлеченных из БД. Причина такого исключения связана с тем, что извлечение исходящего сообщения должно иметь ссылки на другие сущности. Когда исходящее сообщение извлекается, должны быть определены: его деловой партнер, создатель и

служащие-отправители, а также и другие ссылки. Эта проблема решена в итерации 3 задержкой определения ссылок до последней стадии извлечения исходящего сообщения (то есть только после того, как все исходящие сообщения будут извлечены). Решение (которое здесь не обсуждается) заключается в методе `completeReferences()` (завершить ссылки) класса `MDataMapper`.

Есть и некоторые другие методы, введенные в `MDataMapper`. Эти методы главным образом поддерживают размещение или извлечение ежедневных отчетов и подходы к авторизации. Рис. 23.11 отображает полный список методов в `MDataMapper`.

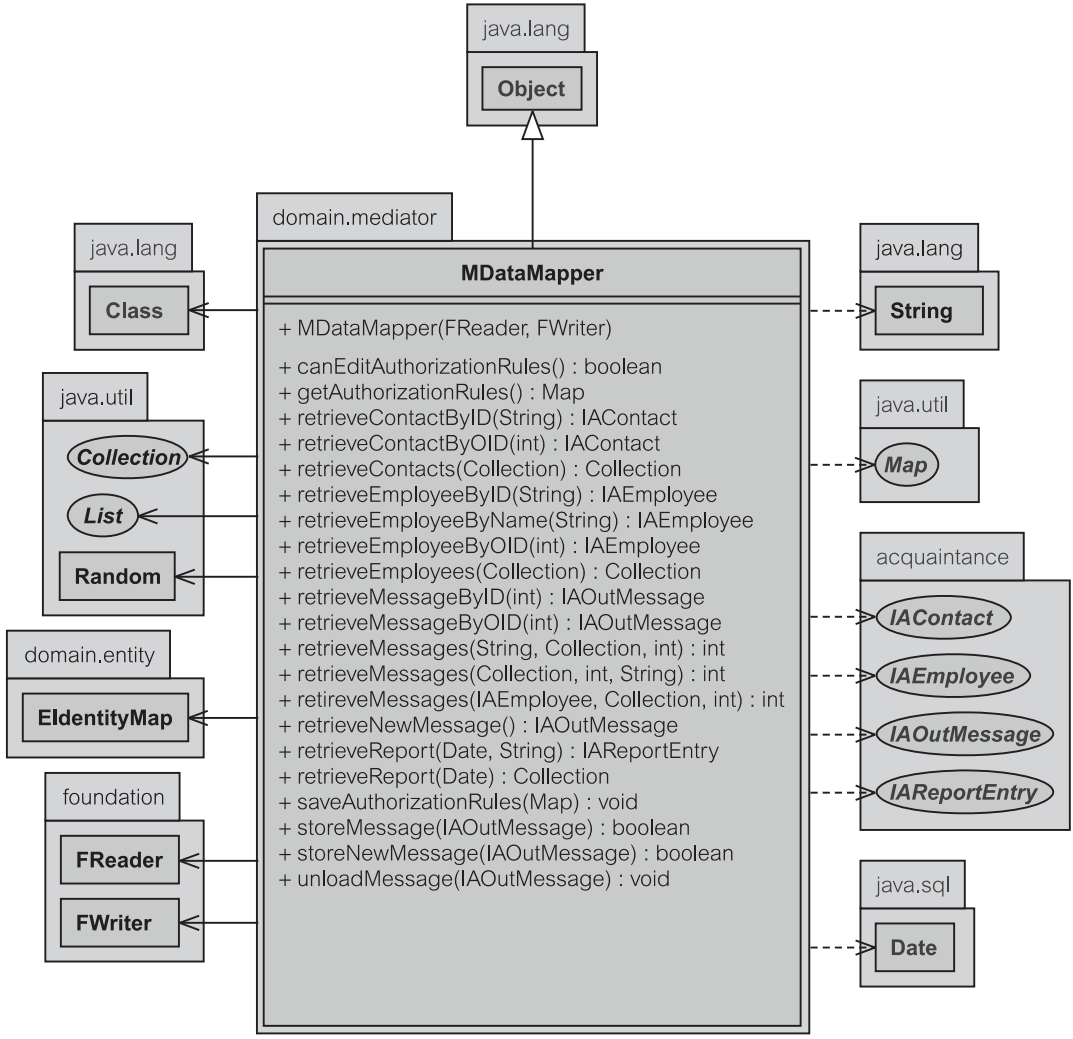


Рис. 23.11. Класс MDataMapper в EM 3

Изменения в существовавших методах

Листинг 23.26 показывает изменения метода `unloadMessage()` (выгрузка сообщения) в итерации 3. Это изменение представляет и многие аналогичные изменения в `MDataMapper`, которые здесь не обсуждаются. Советуем заинтересованному читателю просмотреть документацию, доступную на Web-сайте книги.

Простой SQL-оператор, ранее используемый в итерации 2 (строка 155), был заменен вызовом хранимой процедуры (строки 156–159). Метод `execute()` (выполнить) класса `FWriter` (класс «запись» пакета `foundation`) используется, чтобы обеспечить выполнение хранимой процедуры `usp_delete_outmessage` (хранимая процедура удаления исходящего сообщения) — строка 156. Метод требует строку, чтобы определить оператор `call` (вызов) в имени хранимой процедуры. Параметры для хранимой процедуры помечены знаками вопроса. Каждый параметр определен классом `Class` (класс), чтобы указать тип параметра хранимой процедуры, и классом `Object` (объект), чтобы представить величину параметра.

Листинг 23.26. Метод unloadMessage() в MDataMapper

Метод unloadMessage() в MDataMapper

```
144:    public void unloadMessage(IAOutMessage msg) throws
                                                        Exception{
145:       msg.setDirty(true);
146:       cache.unregisterMessage((IEObjectID)msg);
147:       //пропустить удаление несуществующего сообщения
148:       Integer oid = new Integer(((EOutMessage)msg).
                                                    getOID());
149:       if(newMsgOIDs.contains(oid)){
150:          newMsgOIDs.remove(oid);
151:          return;
152:       }
153:
154:       try{
155:    //st = writer.delete("delete from outmessage where
                                          message_id = ?");
156:          writer.execute("{ call
                                usp_delete_outmessage(?)}",
157:             new Class[]{Integer.class},
158:             new Object[]{new Integer(msg.getMessageID())}
159:          );
160:       }catch(Exception exc){
161:          exc.printStackTrace();
162:          throw exc;
163:       }
164:    }
```

Извлечение отчета в MDataMapper

Обязанность извлечения отчетов возложена на метод `retrieveReport()` (извлечь отчет) класса `MDataMapper`, показанный в листинге 23.27. Этот метод просто извлекает данные из БД и размещает их в `EIdentityMap` (класс «коллекция идентичности объектов» пакета `entity`). Извлечение отчета из БД предполагает, что последняя была заполнена отчетами. Это не обязательно означает, что отчеты были сформированы и сохранены заранее. Они могут быть сформированы хранимой процедурой, когда это будет необходимо, и временно размещены в БД для извлечения приложением. Таков подход, принятый в итерации 3.

Листинг 23.27. Метод retrieveReport() в MDataMapper

Метод retrieveReport() в MDataMapper

```
595:    public Collection retrieveReport(java.sql.Date day) {
596:   //проверка, является ли требуемый день сегодняшней датой
597:       Date now = new java.sql.Date(System.
                                currentTimeMillis());
598:       Calendar cal = GregorianCalendar.getInstance();
599:       cal.setTime(now);
600:       Calendar dayCal = GregorianCalendar.getInstance();
601:       dayCal.setTime(day);
602:
603:       boolean isToday = false;
610:       isToday = cal.get(Calendar.DATE) == dayCal.
                                          get(Calendar.DATE) &&
611:       cal.get(Calendar.MONTH) ==
                                dayCal.get(Calendar.MONTH) &&
612:       cal.get(Calendar.YEAR) == dayCal.get(Calendar.YEAR);
613:
614:       if(isToday)
615:          generateUpdateReports();
617:       EReportEntry rpt = null;
618:       java.sql.ResultSet rs=null;
619:       IAContact contact;
620:       Collection results = null;
621:       if(!isToday) results = cache.getReports(day);
622:       if(!results.isEmpty()) return results;
623:       try {
625:          rs = reader.query("{ ? = call
                usp_retrieve_dailyreport(?)}",
                       new Object[]{day});
626:          while(rs.next()) {
627:             contact = null;
628:             contact = cache.getContact
                                (rs.getString("contact_id"));
```

```
629:              if(contact == null) contact =
                  retrieveContactByID(rs.getString
                                              ("contact_id"));
630:              int created = rs.getInt("num_emails_created");
631:              int sent = rs.getInt("num_emails_sent");
632:              int outstanding =
                            rs.getInt("num_emails_outstanding");
633:              int pastout =
                       rs.getInt("num_emails_past_outstanding");
634:
635:              rpt = new EReportEntry(day,contact,created,
                                  sent,outstanding,pastout);
636:              cache.registerContact((IEObjectID)contact);
637:              cache.registerReport((IEObjectID)rpt);
638:              results.add(rpt);
639:           }
640:        } catch (Exception exc) {
641:           exc.printStackTrace();
642:           return null;
643:        }finally{
644:           if(rs != null) reader.closeResult(rs);
645:        }
646:        return results;
647:     }
```

Листинг 23.27 извлекает список отчетов для конкретного дня. Имеются некоторые другие изменения этого метода в `MDataMapper`, однако они не рассматриваются в данной главе, поскольку характеризуются только незначительными деталями.

Итерация 3 предполагает, что прошлые отчеты для дней, отличающихся от сегодняшней даты, не нужно заново формировать в БД. Однако запрос на формирование отчета на сегодняшнюю дату требует каждый раз создания нового отчета. Это происходит потому, что сегодняшние сообщения по электронной почте пока не являются историей в том смысле, что новые сообщения постоянно обрабатываются различными служащими, использующими подсистему EM.

Код в листинге 23.27 определяет, является ли день требуемого отчета сегодняшней датой (строки 610–612). Если это сегодняшняя дата, то `retrieveReport()` (извлечь отчет) формирует отчет в БД (строки 614–615). Попытка извлечь отчет из кэша выполняется, если параметр `day` (день) не является сегодняшней датой. Его можно сразу же видеть, если отчет требовался недавно и находится в кэше. Если это так, то нет никакой необходимости обращения к БД (строки 621–622). Непосредственное извлечение из БД выполняется строкой 625. Она использует метод `query()` (запрос) класса `FReader` (класс «чтение» пакета `foundation`) для обращения к БД. Строки 626–639 показывают формирование и регистрацию отчета.

Загрузка прав доступа в MDataMapper

Хотя права доступа разработаны в итерации 3 как реализация простого преобразования, а не как собственный класс типа `EAuthorizationRule` (класс «права доступа» пакета `entity`), процессы их загрузки/выгрузки все же управляются классом. Этот класс — `MDataMapper`. `MDataMapper` ответствен за все преобразования данных между объектами строк в БД и представлением их сущности в приложении.

Листинг 23.28 показывает, как права доступа извлекаются из БД и преобразуются в представление их сущности (только простая `Map` — коллекция). Хранимая процедура `usp_retrieve_auth_rules` (хранимая процедура извлечения прав доступа) выполняется методом `query()` класса `FReader` (строка 794). Результат выполнения просматривается (строки 796–813) и создается соответствующая коллекция с вхождениями, подобными вхождениям в матрице авторизации (раздел 19.2.3, подпоток S1.1).

Коллекция содержит преобразование от «from_deparment» (исходный отдел) к списку «target_department» (целевой отдел). Первоначально «from_department» регистрируется в коллекции с пустым списком (строки 800–803). Затем вход для «target_department» формируется из списка, принадлежащего «from_department» (строка 804), когда коллекция не была скорректирована измененным списком (строка 805). Служащий отдела должен иметь доступ ко всем исходящим сообщениям его/ее собственного отдела и тех отделов, которые реализованы преобразованием, выполненным в строках 804–812.

Рис. 23.12 представляет структуру коллекции. Рисунок показывает, как список «to_department» отображается из единственного входа «from_department». Как минимум, должно быть преобразование от отдела к самому себе, типа показанного для `from_department3` (исходный отдел 3).

Сохранение прав доступа в MDataMapper

Администратор безопасности может изменять текущие права доступа. Сохранение измененных прав требует преобразования от вхождений коллекции к вхождениям БД. Это в основном преобразование метода `getAuthorizationRules()` (получить права доступа). Метод, который

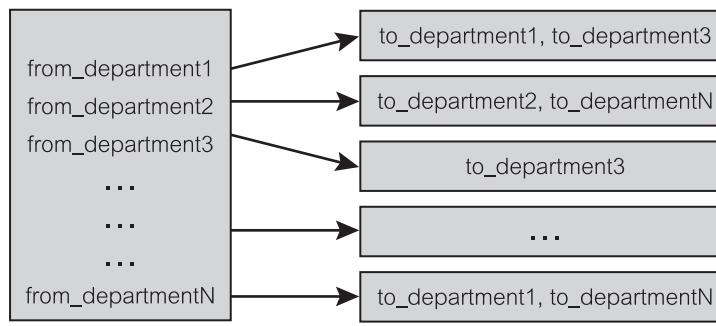


Рис. 23.12. Права доступа в Map

сохраняет права доступа — `saveAuthorizationRules()` (сохранить права доступа) — листинг 23.29.

Листинг 23.28. Метод getAuthorizationRules() в MDataMapper

Метод getAuthorizationRules() в MDataMapper

```
789:    public java.util.Map getAuthorizationRules(){
790:       String sql = "{ ? = call usp_retrieve_auth_rules }";
791:       java.sql.ResultSet rs = null;
792:       java.util.Map m = null;
793:       try{
794:           rs = reader.query(sql,null);
795:           m = new java.util.HashMap();
796:           while(rs.next()){
797:              String code1 =
                              rs.getString("from_dept_code");
798:              String code2 =
                             rs.getString("target_dept_code");
799:              java.util.List l =
                                 (java.util.List) m.get(code1);
800:              if(l == null){
801:                l = new java.util.LinkedList();
802:                if(!code1.equals(code2)) l.add(code1);
803:              }
804:              l.add(code2);
805:              m.put(code1,l); //заменить старый код
806:
807:            //сделать вход для code2, если он не существует
808:              if(m.get(code2) == null){
809:                 java.util.List ll =
                                    new java.util.LinkedList();
810:                 ll.add(code2);
811:                 m.put(code2,ll);
812:              }
813:          }
814:       }catch(Exception exc){
815:       }finally{
816:          if(rs != null) reader.closeResult(rs);
817:       }
818:       return m;
819:    }
```

Листинг 23.29. Метод saveAuthorizationRules() в MDataMapper

Метод saveAuthorizationRules() в MDataMapper

```
840:    public void saveAuthorizationRules(java.util.Map rules)
        throws Exception{
841:       //очистка входов в таблице авторизации
842:       String sql = "{ call usp_clear_auth_rules }";
843:       CallableStatement st = null;
844:       try{
845:          st = writer.prepareCall(sql);
846:          st.execute();
847:       }catch(Exception exc) {
848:          exc.printStackTrace();
849:          throw exc;
850:       }finally{
851:          if(st != null) writer.closeStatement(st);
852:          st = null;
853:       }
854:
855:       //заполнение формы из карты
856:       sql = "{ call usp_upd_auth_rules(?,?) }";
857:       try{
858:          st = writer.prepareCall(sql);
859:          java.util.Set entries = rules.entrySet();
860:          for(Iterator it = entries.iterator();
                                          it.hasNext();){
861:             java.util.Map.Entry entry =
                                    (Map.Entry) it.next();
862:             String key = entry.getKey().toString();
863:             Iterator targets =
                   ((Collection)entry.getValue()).iterator();
864:             while(targets.hasNext()){
865:                st.setString(1, key);
866:                st.setString(2, targets.next().toString());
867:                st.execute();
868:             }
869:          }
870:       }catch(Exception exc){
871:          throw exc;
872:       }finally{
873:          if(st != null) writer.closeStatement(st);
874:          st = null;
875:       }
876:    }
```

Поскольку права доступа получены в формате таблицы и коллекция не помнит, которые из ее вхождений были изменены, процесс корректировки становится несколько затруднительным. Единственный способ выполнить корректировку — удаляя в БД все предыдущие вхождения и заменяя их новыми вхождениями, которые содержатся в `Map`. Это обеспечивается выполнением хранимой процедуры `usp_clear_auth_rules` (хранимая процедура очистки прав доступа) — строки 842–847. Как только БД будет очищена, конкретные вхождения `Map` могут быть поочередно помещены в БД (строки 856–869, особенно строки 859–863).

Чтобы кэшировать строки хранимой процедуры, используется `java.sql.CallableStatement` (строка 858). `CallableStatement` (вызываемый оператор) может многократно использоваться с различными параметрами в разные времена после того, как он будет построен. Это видно в строках 865 и 866, где параметры `CallableStatement` устанавливаются динамически перед выполнением (строка 867). Хранимая процедура `usp_upd_auth_rules` (хранимая процедура корректировки прав доступа) была рассмотрена ранее в главе 20, листинг 20.11.

23.6.3. Класс MUnitOfWork

Это — новое дополнение к пакету `mediator` (посредник), предназначенное для выполнения **единицы работы** (unit of work) в EM (раздел 21.2.4). Методы, уже обсужденные в главе 21, в данной главе не будут рассмотрены. Текущий проект `MUnitOfWork` (класс «единица работы» пакета `mediator`) имеет дело только с `IEObjectID` (интерфейс «идентификатор объекта» пакета `presentation`) как элементом его списка. Рис. 23.13 показывает класс `MUnitOfWork`. Ассоциации с `FConnection` (класс «соединение» пакета `foundation`), `MDataMapper` (класс «преобразователь данных» пакета `mediator`) и шаблоном `Collection` (коллекция) не показаны ради экономии места.

Реализация `MUnitOfWork` выполняет транзакцию в последовательности регистрации ее сущностей. Несмотря на то, что сущности регистрируются по трем различным категориям (будет рассмотрено позже), она (реализация) фиксирует изменения в БД в последовательности регистрации сущностей. Это гарантирует, что ссылки от сущностей, зарегистрированных позже, к сущностям, зарегистрированным ранее, не нарушатся при фиксации изменений в БД.

Статические методы вводятся в `MUnitOfWork`, чтобы обеспечить реализацию его *одноэлементного* паттерна, как подробно рассмотрено в разделе 21.2.4. Типичные операции, использующие `MUnitOfWork`, проиллюстрированы в разделе 21.3.2, рис. 21.9. Эта глава объясняет несколько дополнительных деталей реализации `MUnitOfWork`.

Приложение, разработанное с учетом параллелизма, может использовать выгоды `ThreadLocal` (локальный поток), чтобы хранить уникальный экземпляр переменной для потока (отображение на основе потока). `ThreadLocal` используется в `MUnitOfWork`, чтобы обеспечить свой собственный единственный экземпляр на каждое обращение к транзакции.

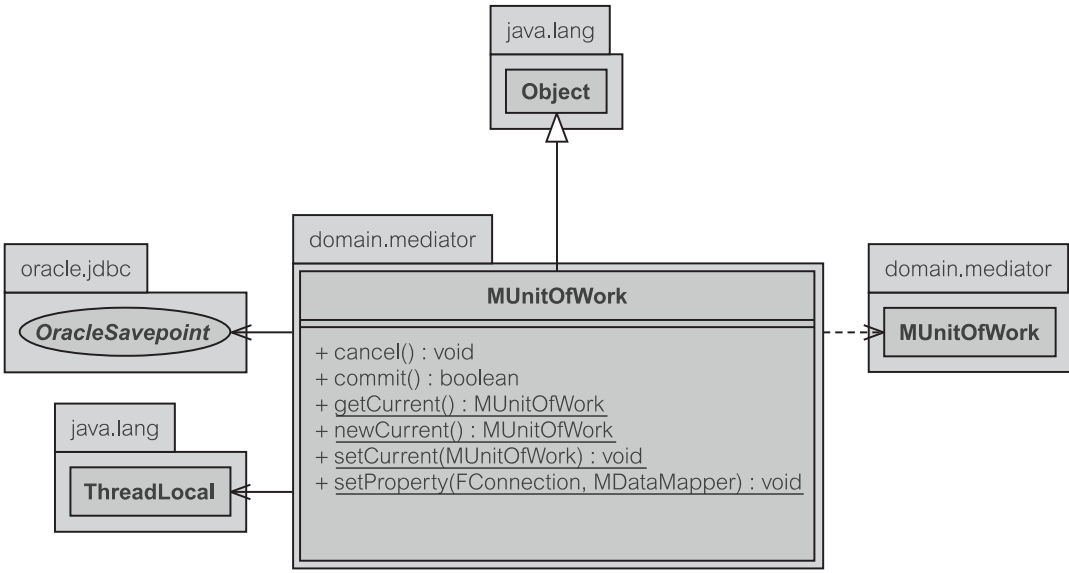


Рис. 23.13. Класс MUnitOfWork в EM 3

`ThreadLocal` можно рассматривать как преобразование от потока к своей собственной версии `MUnitOfWork`.

Как только единица работы будет получена, может быть выполнен ряд регистраций для транзакции через методы `registerNew()` (регистрировать новый), `registerDirty()` (регистрировать измененный) и `registerRemoved()` (регистрировать удаленный). Поскольку это транзакция, отказ в любой ее части вызывает откат.

Получение MUnitOfWork

Большинство методов в `MUnitOfWork` являются статическими. Метод `newCurrent()` (новый текущий) — единственный способ получить новый экземпляр `MUnitOfWork`, как показано в листинге 23.30. Конструктор по умолчанию (строка 93) создает экземпляр единицы работы, прежде чем он будет зарегистрирован как самый последний объект `MUnitOfWork` согласно одноэлементному паттерну (строка 95).

Одноэлементный паттерн поддерживается экземпляром `ThreadLocal`, называемым `currentUnit` — текущая единица (строка 101). `ThreadLocal` обеспечивает только два метода — `set()` (задать), чтобы изменить величину, содержащуюся в `ThreadLocal`, и `get()` (получить), чтобы извлечь помещенную величину. Задание этой единицы работы параметру `currentUnit` приводит к тому, что самая последняя ссылка становится для последующего метода `getCurrent()` (получить текущую) данной единицей работы (строка 108).

Листинг 23.30. Методы newCurrent(), setCurrent() и getCurrent() в MUnitOfWork

Методы newCurrent(), setCurrent() и getCurrent() в MUnitOfWork

```
92:     public static MUnitOfWork newCurrent() {
93:        MUnitOfWork work = new MUnitOfWork();
95:        setCurrent(work);
96:        return getCurrent();
97:     }
100:    public static void setCurrent(MUnitOfWork unit) {
101:       currentUnit.set(unit);
102:    }
106:    public static MUnitOfWork getCurrent() {
107:       if (connection == null || mapper == null)
              throw new IllegalStateException(
                 "Please set the connection and mapper first");
108:       return (MUnitOfWork) currentUnit.get();
109:    }
```

Регистрация новой сущности в MUnitOfWork

`MUnitOfWork` может регистрировать объект сущности как находящийся в одном из трех состояний. Сущность может быть зарегистрирована 1) как *новое* вхождение (и поэтому может быть вставлено в БД), 2) как *измененное* вхождение (означающее, что вхождение БД может быть скорректировано измененной версией этой сущности) или 3) как *удаленное* вхождение (и поэтому соответствующее вхождение БД может быть удалено).

Листинг 23.31 иллюстрирует регистрацию новой сущности в единице работы. Чтобы помнить порядок регистрации в течение всего процесса регистрации, используется внутренний класс. Этот подход применяется для всех регистраций (*новое*, *скорректированное* или *удаленное* вхождение). Внутренний класс называется `MPair` (класс «пара данных» пакета `mediator`). Это простой контейнер двух объектов: целое число, чтобы представить номер регистрации, и сущность, которая будет зарегистрирована. Этот отдельный объект `MPair` помещен в контейнер `newObjects` — новые объекты (строки 115 и 118).

Листинг 23.31. Метод registerNew() в MUnitOfWork

Метод registerNew() в MUnitOfWork

```
114:    public boolean registerNew(IEObjectID obj) {
115:       MPair p = new MPair(new Integer(index++), obj);
116:
117:       if (dirtyObjects.contains(p) ||
              removedObjects.contains(p) ||
              newObjects.contains(p)) return false;
118:       newObjects.add(p);
119:       return true;
120:    }
```

Необходимо проверять, что новая сущность, которая будет зарегистрирована, не была зарегистрирована ранее. Если она была зарегистрирована ранее, то нет никакой необходимости регистрировать ее заново и поэтому регистрация будет отменена (строка 117).

Регистрация измененной сущности в MUnitOfWork

Подобно методу `registerNew()` метод `registerDirty()` в листинге 23.32 создает объект `MPair`, который будет размещен в контейнере `dirtyObjects` — измененные объекты (строки 127 и 132). Снова выполняется тщательная проверка, чтобы гарантировать непротиворечивость в единице работы. Если измененная сущность, которая должна быть зарегистрирована, была ранее зарегистрирована для удаления (то есть она существует в контейнере `removedObjects`), то регистрация отменяется (строка 129). Однако если измененная сущность была предварительно зарегистрирована как новая или измененная, то регистрация просто игнорируется.

Рисунки 21.7 и 21.9 в главе 21 показывают, как транзакции, включающие единицу работы, выполняют корректировки зарегистрированных измененных сущностей.

Листинг 23.32. Метод registerDirty() в MUnitOfWork

Метод registerDirty() в MUnitOfWork

```
126:    public boolean registerDirty(IEObjectID obj) {
127:       MPair p = new MPair(new Integer(index++), obj);
129:       if (removedObjects.contains(p)) return false;
131:       if (newObjects.contains(p) ||
                                  dirtyObjects.contains(p))
              return true;
132:       dirtyObjects.add(p);
133:       return true;
134:    }
```

Удаление сущности в MUnitOfWork

Удаление сущности подобно принципам регистрации (листинг 23.33). Создается объект `MPair` для размещения в `removedObjects` (строки 142 и 149). Выполняется проверка, чтобы определить, существует ли объект сущности, который будет удален в контейнере `newObjects`. Если он существует, то он должен быть удален из контейнера `newObjects` (строка 145). Это может произойти, если пользователь отменит создание новой сущности. Точно так же, если сущность, которая будет удалена, существует в контейнере `dirtyObjects`, то она должна быть удалена из этого контейнера (строка 149). В конце концов, не так уж много случаев при обновлении сущности, когда они должны быть удалены.

Листинг 23.33. Метод registerRemoved() в MUnitOfWork

Метод registerRemoved() в MUnitOfWork

```
141:  public void registerRemoved(IEObjectID obj) {
142:    MPair p = new MPair(new Integer(index++), obj);
145:    if (newObjects.remove(p)) return;
149:    dirtyObjects.remove(p);
150:
151:    if (!removedObjects.contains(p)) removedObjects.add(p);
152:    }
```

Объект сущности, регистрируемый для удаления, будет лишь помещен в контейнер `removedObjects`, если он не был предварительно зарегистрирован в этом контейнере (строка 151). Двойная регистрация может вызывать исключения при выполнении транзакции.

Примеры выполнения единицы работы для целей удаления объектов сущности показаны на рисунках 21.8 и 21.9 в главе 21.

Фиксация MUnitOfWork

`MUnitOfWork` может иметь два состояния, как только все сущности должным образом будут зарегистрированы: зафиксированное или отмененное. Зафиксированное состояние завершается записью зарегистрированных сущностей в БД, в то время как отмененное состояние игнорирует все регистрации.

Рис. 21.9 в главе 21 показывает всю единицу работы в действии. Листинг 23.34 содержит код, который соответствует рис. 21.9. Метод `startTransaction()` (начать транзакцию) предназначен для получения транзакции из БД. Если она не может быть получена, то вся последовательность выполнения единицы работы не будет в единой транзакции, и поэтому благоразумно отменить операцию (строки 158–159).

Листинг 23.34. Метод commit() в MUnitOfWork

Метод commit() в MUnitOfWork

```
155:  public synchronized boolean commit)) {
158:     if (!startTransaction())
159:        return false;
161:     //выполнение списка транзакций
162:     try {
163:       executeTransactions(newObjects.iterator(),
           dirtyObjects.iterator(), removedObjects.iterator());
164:       }catch (Exception exc) {
165:          rollback();
167:          return false;
169:       }
172:       if (!connection.end()) rollback();
173:       setCurrent(null);
174:       return true;
175:    }
```

Как только транзакция начнется, последовательность добавлений, корректировок и удалений выполняется в пределах этой транзакции с помощью метода `executeTransactions()` (выполнить транзакции) — строка 163. Этот метод гарантирует, что модификации (добавление, корректировка, удаление) выполняются согласно их последовательности получения в течение регистрации.

Если возникнет исключение, транзакция должна откатиться назад (строки 164–169). Точно так же, когда попытка зафиксировать транзакцию окажется неудачной, должен быть выполнен откат (строка 172).

Успешное выполнение единицы работы отменяет текущий одноэлементный экземпляр, задаваемый объектом `MUnitOfWork` (то есть сформированный в `currentUnit`). Эта отмена выполняется методом `setCurrent()` (задать текущий) с пустой единицей работы. Когда приложение пытается получить пустую единицу работы, это вынуждает создать новый экземпляр единицы работы с помощью метода `newCurrent()`.

Выполнение транзакции

Необходимо зафиксировать изменения объектов сущности в БД в порядке полученных (зарегистрированных) изменений. Это обеспечивается методом `executeTransactions()`, показанным в листинге 23.35. Показано рекурсивное решение `executeTransactions()`, в котором метод пытается определить, какая из этих трех сущностей (`newIt` — новая сущность, `dirtyIt` — измененная сущность или `removedIt` — удаленная сущность) должна быть выполнена в первую очередь (строка 197). Метод использует функцию, называемую `less()` (меньшее количество), чтобы выполнить эту работу. Достаточно сказать, что `less()` возвратит более раннюю зарегистрированную сущность из двух параметров, заданных этой функции.

Как и во всех рекурсивных алгоритмах, должен быть вариант останова. Вариант останова — это когда итераторы больше не имеют у себя никакой сущности (строки 191–195).

Метод выполняет только самое раннее вхождение из трех возможных вхождений. Это означает, что другие два вхождения, которые были уже взяты итераторами (строки 191–193), следует возвратить обратно итераторам (строки 198–200). Такой прием позволяет оставшимся двум сущностям быть обработанными в течение следующего шага рекурсивного действия (строка 207).

В зависимости от типа сущности обработка кончается вызовом 1) `insertNew()` (поместить новую), если это новая сущность, 2) `updateDirty()` (скорректировать измененную), если сущность была зарегистрирована с помощью `registerDirty()`, или 3) `deleteRemoved()` (удалить отмененную), если сущность должна быть удалена (строки 202–204).

Диаграмма последовательности действий, соответствующая листингу 23.35, показана на рис. 21.9 в главе 21.

Листинг 23.35. Метод executeTransactions() в MUnitOfWork

Метод executeTransactions() в MUnitOfWork

```
188:    private void executeTransactions(
        ListIterator newIt, ListIterator dirtyIt, ListIterator
        removedIt) throws Exception {
189:       MPair newP = null, dirtyP = null, removedP = null;
190:
191:       if (newIt.hasNext()) newP = (MPair) newIt.next();
192:       if (dirtyIt.hasNext()) dirtyP = (MPair)
                                                dirtyIt.next();
193:       if (removedIt.hasNext()) removedP = (MPair)
                                              removedIt.next();
194:
195:    if (newP == null && dirtyP == null && removedP == null)
                                                        return;
196:
197:      MPair lowestIndex = less(less(newP, dirtyP),
                    removedP);
198:      if(lowestIndex != newP && newP != null)
                                       newIt.previous();
199:      if(lowestIndex != dirtyP && dirtyP != null)
                                           dirtyIt.previous();
200:       if(lowestIndex != removedP && removedP != null)
                     removedIt.previous();
201:
202:       if(lowestIndex == newP)
              insertNew((IEObjectID) lowestIndex.second);
203:       else if (lowestIndex == dirtyP)
              updateDirty((IEObjectID) lowestIndex.second);
204:       else deleteRemoved((IEObjectID) lowestIndex.second);
207:       executeTransactions(newIt, dirtyIt, removedIt);
208:    }
```

Начало транзакции

Метод `startTransaction()` гарантирует, что старт (начало) транзакции обеспечивается из БД (листинг 23.36). В `MUnitOfWork` это выполняется тем, что `FConnection` (класс «соединение» пакета `foundation`) начинает транзакцию своим методом `begin()` — начать (строка 200), сопровождаемым созданием точки сохранения, чтобы обезопасить выполнение после точки сохранения (строка 220).

Листинг 23.36. Метод startTransaction() в MUnitOfWork

```
Метод startTransaction() в MUnitOfWork

218:    private boolean startTransaction() {
219:       try {
220:         connection.begin();
221:         savepoint = connection.setSavepoint("unitofwork" +
                                                  transactionID);
222:       }catch (Exception exc) {
223:          exc.printStackTrace();
224:
225:          //возвращение автоматической фиксации
226:          if (!connection.getAutoCommit())
227:             try{ connection.setAutoCommit(true);
                    }catch(Exception exc2){}
228:          return false;
229:       }
230:       return true;
231:    }
```

В случае неудачи, например, из-за невозможности получения транзакции от `FConnection` или получения точки сохранения, объект `FConnection` должен быть повторно установлен в первоначальное состояние, в котором автоматическая фиксация устанавливается в true — истина (строки 226–228).

23.7. Пакет Foundation

Модификации пакета `foundation` (основание) в итерации 3 весьма незначительны. Большинство модификаций включает поддержку **транзакций** в `FConnection` (класс «соединение» пакета `foundation`). Дополнительные усовершенствования находятся в `FWriter` (класс «запись» пакета `foundation`) и `FReader` (класс «чтение» пакета `foundation`), чтобы обеспечить вызовы хранимых процедур и хранимых функций.

Этот раздел показывает только несколько образцов кода, чтобы проиллюстрировать типичные операции в пределах `foundation`. Заинтересованные читатели могут найти полный код на Web-сайте книги.

23.7.1. Транзакции в FConnection

Особенности транзакций класса `FConnection` (класс «соединение» пакета `foundation`) зависят от поддержки, обеспечиваемой используемым JDBC-драйвером. Начиная с JDK1.4, Java предоставил много новых особенностей транзакций, которые уже использовались многими продавцами БД. Например, фирма Oracle использовала точку сохранения, откат с точкой сохранения и многие другие особенности с помощью своих собственных клас-

сов. Листинг 23.37 показывает метод, обеспечиваемый фирмой Oracle, чтобы выполнить откат. Вместо использования `java.sql.Savepoint` метод `rollback()` (откат) использует `oracle.jdbc.OracleSavepoint()` (строки 207–208).

Листинг 23.37. Метод rollback() в FConnection

Метод rollback() в FConnection

```
207:    //public boolean rollback(java.sql.Savepoint savepoint){
208:    public boolean rollback(oracle.jdbc.OracleSavepoint
                                                    savepoint){
209:        try{
212:           //connection.rollback(savepoint);
214:   ((OracleConnection)conn).oracleRollback(savepoint);
215:        }catch(Exception exc){
216:           return false;
217:        }
218:        return true;
219:     }
```

Аналогично вместо использования метода `rollback()` класса `java.sql.Connection` нужно использовать метод `oracleRollback()` (откат) класса `oracle.jdbc.OracleConnection` (строки 212–214).

Список методов в `FConnection` показан на рис. 23.14. Он содержит ряд методов типа `getAutoCommit()` (получение автоматической фиксации), `setAutoCommit()` (задание автоматической фиксации), `rollback()` (откат), `setSavepoint` (задание точки сохранения), `releaseSavepoint()` (снять точку сохранения), `begin()` (начать), `end()` (завершить) и другие особенности транзакций.

23.7.2. Операторы Execute в FWriter

В `FWriter` (класс «запись» пакета `foundation`) итерации 3 введен новый метод `execute()` (выполнить), показанный в листинге 23.38. Этот метод позволяет выполнять произвольный `CallableStatement` (вызываемый оператор) в соединении БД. Метод `execute()` (выполнить) задает параметры для `CallableStatement`. Метод в настоящее время поддерживает объекты `null` (пустой указатель), `String` (строка), `java.sql.Date` (дата) и `Integer` (целое число) как возможные параметры выполнения (строки 65–69). Если ряд параметров, объявленных в `parameters` (параметры), не соответствует этим типам, для пользователя формируется исключение, чтобы указать на это (строка 55).

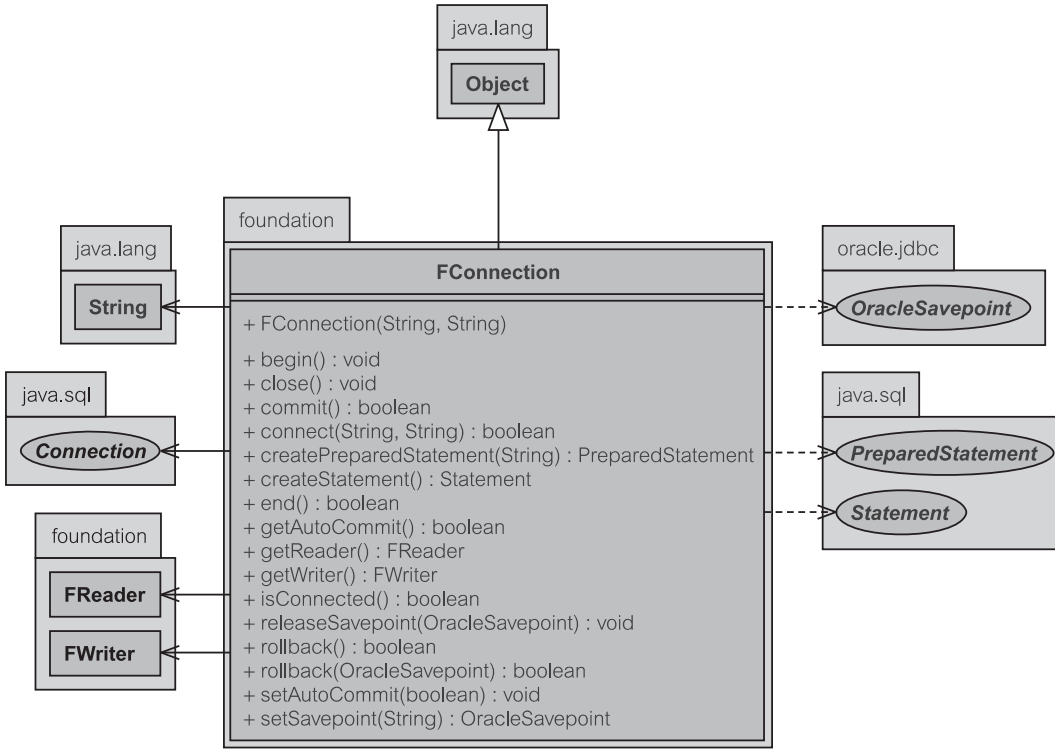


Рис. 23.14. Класс FConnection в EM 3

Листинг 23.38. Метод execute() в FWriter

Метод execute() в FWriter

```
54:    public void execute(String sql, Class[] types, Object[]
       parameters) throws Exception{
55:      if(types != null && parameters.length != types.length)
              throw new IllegalArgumentException(
              "Different sizes of parameters and types given");
56:
57:      CallableStatement st = null;
58:      try{
59:         st = insert(sql);
60:
61:         if(types != null){
62:            int index;
63:            for(int i = 0;i< parameters.length; i++){
64:               index = i + 1;
65:               if(types[i] == null)
                       st.setNull(index,
                          ((Integer)parameters[i]).intValue());
```

```
66:                else if(types[i] == String.class)
                  st.setString(index, parameters[i].toString());
67:                else if(types[i] == java.sql.Date.class)
                        st.setDate(index,
                                   (java.sql.Date)parameters[i]);
68:                else if(types[i] == Integer.class)
                        st.setInt(index,
                           ((Integer)parameters[i]).intValue());
69:                else
                     throw new
                      IllegalArgumentException("Unknown type:" +
                      types[i] + " with value:"+ parameters[i]);
70:                }
71:             }
72:             st.execute();
73:          }catch(Exception exc){
74:             throw exc;
75:          }finally{
76:             if(st != null) closeStatement(st);
77:             st = null;
78:          }
79:       }
```

Для каждого опознанного типа параметра метод `execute()` назначает соответствующую величину вызываемого оператора (строки 65–68). Оператор создается обращением к методу `insert()` (строка 58). Нет никакой необходимости в отдельных методах `delete()` (удалить) или `update()` (корректировать), потому что итерация 3 обратится к хранимым процедурам и функциям вызовом `execute()` в противоположность непосредственному использованию SQL-операторов `insert()`, `delete()` или `update()`.

23.7.3. Запрос к БД в FReader

В БД вводятся хранимые функции, чтобы заменить запросы от клиента. Эти хранимые функции доступны из метода `query()` (запросить) в объекте `FReader` (класс «чтение» пакета `foundation`), показанного в листинге 23.39. Метод подобен методу `execute()` (выполнить) в том смысле, что он передает параметры. Запрос `sql` предназначен для обращения к хранимым функциям, которые возвращают результирующий список (с типом `OracleTypes.CURSOR`, строка 46). Как только будет создан `CallableStatement` (вызываемый оператор) — строка 45, его параметры (списка) сразу же назначаются оператору (строки 48–51).

Выполнение объекта оператора происходит в строке 55. Это выполнение может вызвать исключение, которое захватывается оператором обработки исключения (строки 56–65) прежде, чем оно будет повторно передано клиенту. Захват исключения позволяет выполнить соответствующую очистку оператора через метод `close()` — завершить (строка 63). Наконец, результат выполнения возвращается клиенту в виде `ResultSet` (множество результатов).

Множество результатов напоминает объявление первого параметра, первоначально установленное в строке 46.

Листинг 23.39. Метод query() в FReader

```
Метод query() в FReader

41:     public ResultSet query(String sql, Object o[]) throws
                                                        Exception {
42:        CallableStatement st = null;
43:        ResultSet rs = null;
45:        st = connection.prepareCall(sql);
46:        st.registerOutParameter(1,OracleTypes.CURSOR);
48:        for(int i=0;o != null && i < o.length;i++){
49:           if(o[i] instanceof Integer)
                 st.setInt(i+2, ((Integer)o[i]).intValue());
50:           else if(o[i] instanceof java.sql.Date)
                st.setDate(i+2, (java.sql.Date) o[i]);
51:           else
                 st.setString(i+2,o[i].toString());
53:        }
54:        try{
55:           st.execute();
56:        }catch(Exception exc){
63:           st.close();
64:           throw exc;
65:        }
66:        rs = ((OracleCallableStatement) st).getCursor(1);
68:        return rs;
69:     }
```

23.8. Код БД

Некоторые **хранимые процедуры** и **функции** были обсуждены в главе 21 и поэтому не рассматриваются в этой главе. Имеются два типа хранимых процедур/функций, созданных для итерации 3. Первый — множество функций/процедур, которые будут помещены в **глобальную схему** (глава 20, разделы 20.1.1 и 20.3.2). Второй — множество функций/процедур, которые будут развернуты в **локальных схемах** (раздел 20.1.1, особенно листинг 20.4).

Данный раздел не обсуждает процедуры/функции для локальных схем. Они подобны тем, которые представлены в листинге 20.4. Раздел сконцентрирован на хранимых функциях/процедурах глобальной схемы. Ряд более простых хранимых процедур и функций не рассматривается ради экономии места (они доступны на Web-сайте книги).

23.8.1. Ref Cursor для ResultSet

Использование `ref cursor` (ссылка на курсор, определяющая переменные курсора) в итерации 3 позволяет возвратить множество результатов хранимой функции. Объявление `ref cursor` является Oracle-ориентированным. Другие продавцы БД имеют свои собственные методы возвращения множества результатов. `ref cursor`, объявленный в листинге 23.40, помещен в пакет по имени EMS (служба электронных сообщений).

Листинг 23.40. Ref Cursor для ResultSet

Ref Cursor для ResultSet

```
create or replace package EMS as
   type REF_CURSOR is ref cursor;
end EMS;
```

Листинг 23.41. Хранимая функция usp_retrieve_outmessage_emp()

Хранимая функция usp_retrieve_outmessage_emp()

```
1:   create or replace function usp_retrieve_outmessage_emp(
2:      empid IN varchar2, usern in varchar2) return
                                              EMS.ref_cursor is
3:         c EMS.REF_CURSOR;
4:         inEmpID varchar2(100);
5:   begin
6:         inEmpID := null;
7:         select "employee_id" into inEmpID
8:         from "Employee"
9:         where upper("login_name") = upper(usern);
10:
11:        open c for
12:           Select *
13:           from "OutMessage"
14:           where "date_emailed" is null and
15:                 "sender_emp_id" = empid and
16:                 empid = inEmpID;
17:        return c;
18:        EXCEPTION
19:          WHEN NO_DATA_FOUND THEN
20:           raise_application_error(-20002,
21:          'usp_retrieve_outmessage_emp: unmatch username
                                               and loginname');
22:  end;
```

23.8.2. Извлечение исходящих сообщений

Имеется ряд методов для извлечения исходящих сообщений. Все исходящие сообщения, предназначенные для конкретного служащего, восстанавливаются с помощью хранимой процедуры `usp_retrieve_outmessage_emp()` (хранимая процедура извлечения исходящих сообщений для служащего) — листинг 23.41. Данный `empid` (идентификатор служащего) сравнивается со значением, указанным в параметре `usern` (имя пользователя) — строки 6–9. Если выбор окажется неудачным, возникает исключение, которое захватывается в строках 18–21. Исключение далее передается программе клиента как исключение приложения.

Успешная проверка служащего как законного пользователя приложения позволяет программе продолжать выбор исходящих сообщений, назначенных служащему (строки 11–17).

23.8.3. Извлечение исходящих сообщений отдела

Листинг 23.42 демонстрирует функцию `usp_retrieve_outmessage_dept()` (хранимая процедура извлечения исходящих сообщений для отдела), чтобы извлечь множество исходящих сообщений, которые были назначены всем служащим отдела текущего служащего. Проверка идентификатора служащего (строки 47–50) сделана таким же образом, как и в листинге 23.40. Добавлен дополнительный код, чтобы получить отдел, которому принадлежит служащий (строки 52–55). Если служащий не может быть идентифицирован, то формируется исключение (строки 67–70).

Листинг 23.42. Хранимая функция usp_retrieve_outmessage_dept()

Хранимая функция usp_retrieve_outmessage_dept()

```
40:  create or replace function usp_retrieve_outmessage_dept(
41:     department IN varchar2, usern in varchar2) return
                                                EMS.ref_cursor is
42:     c EMS.REF_CURSOR;
43:     inEmpID varchar2(100);
45:     empDept varchar2(100);
46:  begin
47:     inEmpID := null;
48:     select "employee_id" into inEmpID
49:     from "Employee"
50:     where upper("login_name") = upper(usern);
51:
52:     empDept := 'IND'; --независимый служащий по умолчанию
53:     select "department_code" into empDept
54:     from "Employee"
55:     where "employee_id" = inEmpID;
56:
```

```
57:     open c for
58:        Select *
59:        from "OutMessage"
60:        where "date_emailed" is null and
61:        "sched_dpt_code" = department and
62:        department in
63:           (select "target_dept_code"
64:           from "Authorization"
65:           where "from_dept_code" = empDept);
66:     return c;
67:     EXCEPTION
68:        WHEN NO_DATA_FOUND THEN
69:           raise_application_error(-20004,
70:           'usp_retrieve_outmessage_dept: unmatch username
                                              and loginname');
71:  end;
```

Проверенный служащий может извлекать все исходящие сообщения, назначенные его/ее отделу (то есть исходящие сообщения, намеченные всем служащим отдела) — строки 57–66. Фактически, служащий способен рассмотреть и все исходящие сообщения, принадлежащие другим отделам, в том случае, если это позволяют права доступа (строки 62–65).

23.8.4. Удаление исходящего сообщения

Удалению исходящего сообщения предшествует проверка, что служащему дозволено удалить сообщение. Служащий может удалять только свои собственные исходящие сообщения (созданные им/ей) и другие исходящие сообщения, которые он/она может просматривать согласно матрице авторизации (раздел 19.2.3, подпоток S1.1).

Проверка выполняется хранимой процедурой `usp_delete_outmessage()` (хранимая процедура удаления исходящего сообщения) — листинг 23.43. Строки 97–108 проверяют, может ли служащий удалить исходящее сообщение согласно правам доступа (строки 102–108). Снова, если выбор потерпит неудачу, формируется исключение для пользователя (строки 113–116). Успешная проверка позволяет программе продолжить удаление исходящего сообщения (строки 110–111).

Листинг 23.43. Хранимая процедура usp_delete_outmessage()

Хранимая процедура usp_delete_outmessage()

```
80:  create or replace procedure usp_delete_outmessage(
81:     msgID IN varchar2, usern in varchar2) is
82:     dummy integer;
83:     inEmpID varchar2(100);
84:     empDept varchar2(100);
85:     error_msg varchar2(100);
```

```
86:  begin
87:     --Вырезано для сохранения места, то же, что и
                                    листинг 23.42, строки 47-55
96:
97:     dummy := null;
98:     select 1 into dummy
99:     from "OutMessage"
100:    where "message_id" = msgID and
101:    "creator_emp_id" in
102:       (select "employee_id"
103:       from "Employee"
104:       where "department_code" in
105:          (select "from_dept_code"
106:          from "Authorization"
107:          where "target_dept_code" = empDept)
108:       );
109:
110:    delete from "OutMessage"
111:    where "message_id" = msgID;
112:
113:    EXCEPTION
114:       WHEN NO_DATA_FOUND THEN
115:          raise_application_error(-20007,
116:          'usp_delete_outmessage: unmatch username and
               loginname or you are not allowed to delete
               the message');
118: end;
```

23.8.5. Создание исходящего сообщения

Необходимо выполнить обширные проверки, прежде чем новое исходящее сообщение может быть помещено в БД. Эти проверки выполняет хранимая процедура `usp_create_outmessage()` (хранимая процедура создания исходящего сообщения) — листинг 23.44. Стандартная проверка, определяющая, является ли пользователь служащим в таблице `Employee` (служащий), выполняется в строках 137–140.

Если обнаружено, что создатель исходящего сообщения и служащий, который пробует добавить исходящее сообщение, — различные люди, формируется исключение, чтобы запретить добавление (строки 142–145). Дальнейшая проверка выполняется в строках 153–169. Эта проверка должна гарантировать, что служащий назначает новое исходящее сообщение другому служащему в пределах его/ее досягаемости авторизации, как диктуется матрицей авторизации (раздел 19.2.3, подпоток S1.1). Наконец, когда все подтверждено, делается добавление (строки 171–176).

Листинг 23.44. Хранимая процедура usp_create_outmessage()

Хранимая процедура usp_create_outmessage()

```
120: create or replace procedure usp_create_outmessage(
121:   senderID IN varchar2,
122:   --Остальные параметры и переменные вырезаны
       для сохранения места
136: begin
137:    inEmpID := null;
138:    select "employee_id" into inEmpID
139:    from "Employee"
140:    where upper("login_name") = upper(usern);
141:
142:    if inEmpID <> creatorID then
143:       raise_application_error(-20014,
          ('You cannot pretend to be another person
            in creator field: ' || creatorID));
145:    end if;
146:
147:    --Вырезано для сохранения места, то же,
       что в листинге 23.42, строки 52-55
152:    --Начало новой транзакции для управления формированием
                                                      исключения
153:    begin
154:       dummy := null;
155:       if schedulerID is not null then
156:          select 1 into dummy from dual
157:          where schedulerID in
158:             (select "employee_id"
159:             from "Employee"
160:             where "department_code" in
161:                (select "from_dept_code"
162:                from "Authorization"
163:                where "target_dept_code" = empDept)
164:             );
165:       end if;
166:       EXCEPTION
167:          WHEN NO_DATA_FOUND THEN
168:             raise_application_error(-20015,
                  ('You are not allowed to schedule
                    this message to ' || schedulerID));
169:    end;
170:
171:    insert into "OutMessage"(
172:       "sender_emp_id","sched_emp_id",
                                "creator_emp_id","contact_id",
173:       "message_subject","message_text","date_scheduled",
```

```
174:       "date_emailed","sched_dpt_code","message_id",
                                                "date_created")
175:    values(senderID,schedulerID,creatorID,cntID,subject,
176:       text,scheduled,emailed,deptCode,msgID,created);
177:    EXCEPTION
178:       WHEN NO_DATA_FOUND THEN
179:          raise_application_error(-20016,
                'usp_create_outmessage:
                 unmatch username and loginname');
180: end;
```

Подобная идея применяется для корректировки исходящего сообщения (не показанной в этом разделе). Единственное изменение для корректировки — замена оператора `insert` — поместить (строки 171–176) оператором `update` — скорректировать.

23.8.6. Создание отчета

Вхождения таблицы `DailyReport` (ежедневный отчет), создаваемые с помощью хранимой процедуры `usp_generate_dailyreport()` (хранимая процедура создания ежедневного отчета), показаны в листинге 23.45. Эта хранимая процедура сначала создает список всех деловых партнеров со сформированными или уже переданными по электронной почте исходящими сообщениями (строки 5–9 и 15). Затем она подсчитывает число невыполненных, невыполненных в прошлом, созданных, посланных и общее число исходящих сообщений для каждого служащего (строки 16–37).

В таблицу `DailyReport` делается добавление, даже если текущий деловой партнер не имеет никакого вхождения (для даты формирования) — строки 51–62, что проверяется в строках 45–48. Если таблица `DailyReport` содержит вхождение для делового партнера на эту конкретную дату, то данные в таблице `DailyReport` будут скорректированы (строки 64–71).

Листинг 23.45. Хранимая процедура usp_generate_dailyreport

Хранимая процедура usp_generate_dailyreport

```
 1:  create or replace procedure usp_generate_dailyreport
                                          (usern in varchar2)
 2:  is
 3:     cnt_exists integer;
 4:     c EMS.REF_CURSOR;
 5:     cursor contacts_c is
 6:        select distinct "contact_id"
 7:        from "OutMessage"
 8:        where to_date("date_created",'DD/MM/YY') =
              to_date(SYSDATE,'DD/MM/YY')
 9:        or to_date("date_emailed",'DD/MM/YY') =
              to_date(SYSDATE,'DD/MM/YY');
```

```
10:     outstanding integer;
11:     past_outstanding integer;
12:     sent integer;
13:     created integer;
14:   begin
15:     for selected_contact in contacts_c loop
16:       outstanding := 0;
17:       select count(*) into outstanding
18:       from "OutMessage"
19:       where "contact_id" = selected_contact."contact_id" and
20:         to_date("date_scheduled", 'DD/MM/YY') >=
              to_date(SYSDATE,'DD/MM/YY')
              and "date_emailed" is null;
21:
22:       past_outstanding := 0;
23:       select count(*) into past_outstanding
24:       from "OutMessage"
25:       where "contact_id" = selected_contact."contact_id" and
26:         to_date("date_scheduled",'DD/MM/YY') <
              to_date(SYSDATE,'DD/MM/YY')
              and "date_emailed" is null;
27:
28:       sent := 0;
29:       select count(*) into sent
30:       from "OutMessage"
31:       where "contact_id" = selected_contact."contact_id" and
              "date_emailed" is not null;
32:
33:       created := 0;
34:       select count(*) into created
35:       from "OutMessage"
36:       where "contact_id" = selected_contact."contact_id" and
37:         to_date("date_created",'DD/MM/YY') =
              to_date(SYSDATE,'DD/MM/YY');
38:
39:       cnt_exists := null;
44:       begin
45:          select 1 into cnt_exists
46:            from "DailyReport"
47:            where to_date("date_of_report_day",'DD/MM/YY') =
                     to_date(SYSDATE,'DD/MM/YY')
48:            and "contact_id" = selected_contact."contact_id";
49:
50:            if cnt_exists is null then
51:              insert into "DailyReport"("date_of_report_day",
52:               --Вырезано для сохранения места ...
63:            else
```

```
64:               update "DailyReport"
65:              --Вырезано для сохранения места
72:            end if;
73:
74:            EXCEPTION
75:              WHEN NO_DATA_FOUND THEN
76:              insert into "DailyReport"("date_of_report_day",
77:              --Вырезано для сохранения места
88:         end;
89:     end loop;
90:  end;
```

Деловой партнер может не иметь вхождений в таблицу `DailyReport`. Это вызовет исключение, когда будет выполняться оператор выбора. Для этой цели сделано добавление в таблицу `DailyReport` (строки 74–77).

23.8.7. Триггер для таблицы OutMessage

В целях экономии места в этом разделе показан только один **триггер**. Триггер служит для операций добавления и корректировки таблицы `OutMessage` (исходящее сообщение) — листинг 23.46. Триггер используется для проверки следующих условий:

- Проверка 1: проверка, заполнен ли `dept_code` (код отдела), когда заполнен `sched_emp` (запланированный служащий); `dept_code` служащего должен быть тем же самым, что и `sched_dpt_code` (код запланированного отдела).
- Проверка 2: обеспечение, что когда `sched_emp_id` (идентификатор запланированного служащего) и `sched_dpt_code` пустые, по умолчанию `sched_dpt_code` принимает значение `QUA` (Quality Control Employee — служащий контроля качества — см. раздел 19.2.3).
- Проверка 3: параметр `date_scheduled` (запланированная дата) должен быть заполнен.
- Проверка 4: `date_scheduled` не может быть меньше, чем `date_created` (дата создания).
- Проверка 5: когда параметр `date_emailed` (дата передачи по электронной почте) заполнен, он должен быть больше чем `date_created`.

Листинг 23.46. Триггер для таблицы OutMessage

Триггер для таблицы OutMessage

```
create or replace trigger bins_upd_outmessage
before insert or update
on "OutMessage"
for each row
declare
   dept_code varchar2(20);
begin
```

```
        if :new."sched_emp_id" is not null then
        --достоверность проверки 1
        --код отдела не может быть null, если sched_emp заполнен
        if :new."sched_dpt_code" is null then
           raise_application_error(-20100,
           'Schedule Dept Code is empty');
        end if;

        select "department_code" into dept_code
        from "Employee"
        where "employee_id" = :new."sched_emp_id";

        --код отдела и код отдела запланированного служащего
          д.б. одинаковыми
        if dept_code  :new."sched_dpt_code" then
          raise_application_error(-20101,
          'Schedule Dept Code does not match the employee dept');

        end if;

    end if;

    --sched_emp_id и sched_dpt_code пустые,
    --sched_dpt_code должен быть QUA
    --достоверность проверки 2
    if :new."sched_emp_id" is null and :new."sched_dpt_code"
        is null then
       :new."sched_dpt_code" := 'QUA';
    end if;

    --должен быть, если создается или обращение извне EMS
    --достоверность проверки 3
    if :new."date_scheduled" is null then
       raise_application_error(-20102,
       'Schedule date of this message is empty');
    end if;

    --дата планирования не может быть меньше даты создания
    --достоверность проверки 4
    if :new."date_created" > :new."date_scheduled" then
       raise_application_error(-20103,
       'Date scheduled is older than date created');
    end if;

    --дата передачи не может быть меньше даты создания
    --достоверность проверки 5
    if :new."date_emailed" is not null and
       :new."date_created" > :new."date_emailed" then
          raise_application_error(-20104,
          'Date emailed is older than date created');
    end if;
end;
```

Резюме

1. Реализация итерации 3 направлена на проектирование данных, связанных с учебным примером управления электронной почтой. Шаблон PCMEF+ полностью сохранен и строго выполняется.
2. Итерация 3 вводит класс `MUnitOfWork` (класс «единица работы» пакета `mediator`), чтобы обеспечить поддержку транзакций для приложения.
3. Весь SQL-код перемещен на сервер БД в форме хранимых процедур/функций.
4. Все транзакции пользователя инициализируются в классе `MModerator` (класс «координатор» пакета `mediator`).
5. Права доступа применяются ко всем операциям, чтобы ограничить объем исходящих сообщений, которые служащие могут просматривать. Права доступа могут быть изменены только пользователем в роли администратора безопасности/БД.
6. Реализованы функциональные возможности создания ежедневных итоговых отчетов.
7. Исходящие сообщения могут быть изменены перед передачей по электронной почте.
8. Пакет `acquaintance` (знакомство) имеет дополнительный интерфейс по имени `IAReportEntry` (интерфейс «вхождение в отчет» пакета `acquaintance`), чтобы помочь в формировании ежедневных отчетов со статистикой передачи по электронной почте.
9. Пакет `presentation` (представление) имеет дополнительный класс `PTableWindow` (класс «окно таблицы» пакета `presentation`), который используется, чтобы показать права доступа и ежедневные отчеты.
10. Пакет `control` (управление) состоит из трех классов, как и в итерации 2. Итерация 3 лишь изменяет `CAdmin` (класс «администратор» пакета `control`), чтобы обеспечить средства удовлетворения дополнительных требований.
11. Реализация интерфейса `IAReportEntry` добавила класс `EReportEntry` (класс «вхождение в отчет» пакета `entity`) в пакет `entity` (сущность). Класс `EIdentityMap` (класс «коллекция идентичности объектов» пакета `entity`) был слегка изменен, чтобы обслужить этот дополнительный класс сущности.
12. `MUnitOfWork` — новый класс в пакете `mediator` (посредник), чтобы обращаться с транзакциями. Имеются изменения в классах пакета `mediator`, связанных с заменой SQL-команд, обеспечивающих доступ к БД, вызовами размещенных в БД хранимых процедур и функций.
13. Модификации в пакете `foundation` (основание) весьма незначительны. Большинство модификаций включает поддержку для транзакций. Дополнительные усовершенствования поддерживают вызовы хранимых процедур.

14. Итерация 3 перемещает бизнес-логику приложения в БД. Это завершается существенным числом хранимых процедур и функций, а также реализацией триггеров, чтобы обеспечить целостность БД.

Ключевые термины

Итерация 3. Вопросы и упражнения

Ниже приведен небольшой набор вопросов и упражнений в отношении кода итерации 3 подсистемы EM. Большее количество вопросов и упражнений имеется на Web-сайте книги.

1. Обратимся к интерфейсу `IAReportEntry` (интерфейс «вхождение в отчет» пакета `acquaintance`) на рис. 23.3. Как вы думаете, каковы причины того, что программист использовал `java.sql.Date`, чтобы представить даты отчета, вместо `java.util.Date`?
2. Обратимся к разделу 23.3.1 «Заполнение списка деловых партнеров в отчете». Укажите причину(ы) использования ключевого слова `synchronized` (синхронизировано) в листинге 23.1, строка 50.
3. Обратимся к разделу 23.3.1 «Окно отчета». Листинг 23.2 иллюстрирует один путь формирования окна отчета. Обсудите несколько альтернативных путей.
4. Логика печати отчетов помещена в класс `PTableWindow` (класс «окно таблицы» пакета `presentation`). Не лучше ли было вместо этого поместить такую логику в класс `PWindow` (класс «окно» пакета `presentation`)? Объясните.
5. Печать отчетов (раздел 23.3.1) реализована исключительно в `PTableWindow`. В этой реализации любые модификации механизма печати или выбора компонентов, которые будут напечатаны, должны быть помещены в `PTableWindow`. Какими были бы другие, более гибкие подходы?
6. Почему экземпляр `PWindow` в листинге 23.2, строка 499, объявлен как `final` (заключительный)? (Намек: попробуйте удалить ключевое слово `final` и повторно скомпилируйте. Посмотрите, что произойдет.)

7. Обратимся к разделу 23.3.1 «Окно отчета» и листингу 23.2. Анонимный внутренний класс в строках 502–511 использует свой собственный поток. Почему следующий анонимный внутренний класс в строках 512–516 не использует поток, чтобы снять бремя с потока изображения?
8. Обратимся к разделу 23.3.1 и листингу 23.11 (и другим листингам для `PMessageDetailWindow` — класс «окно деталей сообщения» пакета `presentation`). Было бы выгодно сделать рефакторинг для `PMessageDetailWindow`? Если это так, то что могло бы быть целью рефакторинга? Объясните.
9. Обратимся к разделеу 23.6.2 «Извлечение отчета в `MDataMapper`» и листингу 23.27. Подумайте, почему код в листинге 23.27 использует такую сложную проверку, чтобы увидеть, является ли параметр `day` (день) сегодняшней датой? Можно ли это сделать каким-либо другим путем?

Литература

1. Agile (2003) http://agilealliance.org/home (последнее обращение в июле 2003 г.).
2. Alur, D., Crupi, J. and Malks, D. (2003) *Core J2EE Patterns: Best Practices and Design Strategies*, 2nd edn. Englewood Cliffs, NJ: Prentice Hall.
3. Apache (2003) http://www.apache.org/ (последнее обращение в июле 2003 г.).
4. ArgoUML (2003) http://argouml.tigris.org/ (последнее обращение в августе 2003 г.).
5. Beck, K. (1997) *Smalltalk Best Practice Patterns*. Englewood Cliffs, NJ: Prentice Hall.
6. Beck, K. (1999) *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley [Имеется русский перевод книги: Бек К. Экстремальное программирование. Библиотека программиста. — СПб.: Питер, 2002].
7. Benson, S. and Standing, C. (2002) *Information Systems: A Business Approach*. Brisbane: John Wiley.
8. Boehm, B.W. (1981) *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall.
9. Boehm, B.W. (1984) Software Engineering Economics, *IEEE Tran. Soft. Eng.*, 1: 4–21.
10. Boehm, B.W. (1988) A spiral model of software development and enhancement, *Computer*, May: 61–72.
11. Boehm, B.W., Abts, C., Brown, A.W., Chulani, S., Clark, B.K., Horowitz, E., Madachy, R., Reifer, D. and Steece, B. (2000) *Software Cost Estimation with COCOMO II*. Englewood Cliffs, NJ: Prentice Hall.
12. Booch, G., Rumbaugh, J. and Jacobson, I. (1999) *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley [Имеется русский перевод книги: Буч Г., Рамбо Дж., Якобсон А. Язык UML: руководство пользователя. — СПб.: Питер, 2003].
13. Bonazzi, E. and Stokol, G. (2001) *Oracle® 8i and Java™. From Client/Server to E-Commerce*. Englewood Cliffs, NJ: Prentice Hall.
14. Borland (2003) http://www.borland.com/together/ (последнее посещение в сентябре 2003 г.).
15. Brooks, F.P. (1987) No silver bullet: essence and accidents of software engineering, *IEEE Software*, 4: 10–19; reprinted in C.F. Kemerer (ed.) (1997) *Software Project Management: Readings and Cases*. Chicago: Irwin, pp. 2–14.
16. Bugzilla (2003) http://gcc.gnu.org/bugs/management.html (последнее посещение в сентябре 2003 г.).

17. Buschmann, E., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996) *Pattern-Oriented Software Architecture: A System of Patterns*. New York: John Wiley & Sons.
18. Cambridge (2003): http://dictionary.cambridge.org/ (последнее посещение в сентябре 2003 г.).
19. Cattell, R.G.G. (1994) *Object Data Management: Object-Oriented and Extended Relational Database Systems*, rev. edn. Reading, MA: Addison-Wesley.
20. Charette, R.N. (1989) *Software Engineering Risk Analysis and Management*. Reading, MA: McGraw-Hill.
21. Chidamber, S.R. and Kemerer, C.F. (1994) A metrics suite for object oriented design, *IEEE Trans. Soft. Eng.*, 6: 476–93.
22. CMM (2003) http://www.sei.cmu.edu/cmm/cmm.html (последнее посещение в августе 2003 г.).
23. Codd, E.F. (1982) Relational database: a practical foundation for productivity, *Comm. ACM*, 2: 109–17.
24. De Millo, R.A., Lipton, R.J. and Perlis, A.J. (1979) Social processes and proofs of theorems and programs, *Comm. ACM*, 5: 271–80.
25. Eckel, B. (2000) *Thinking in Java*, 2nd edn. Englewood Cliffs, NJ: Prentice Hall, http://www.planetpdf.com/ (последнее посещение в апреле 2002 г.). [Имеется русский перевод книги: Эккель Б. Философия Java. Библиотека программиста. 3-е изд. — СПб.: Питер, 2003].
26. Eclipse (2003) http://www.eclipse.org/ (последнее посещение в сентябре 2003 г.).
27. Ege, R. (1992) *Programming in an Object-Oriented Environment*. New York: Academic Press.
28. eProject (2003) http://www.eproject.com/ (последнее посещение в июле 2003 г.).
29. eRoom (2003) http://www.eroom.net/eRoomNet/ (последнее посещение в июле 2003 г.).
30. Fowler, M. (1999) *Refactoring: Improving the Design of Existing Code*. Harlow: Addison-Wesley [Имеется русский перевод книги: Фаулер М. Рефакторинг: улучшение существующего кода. — СПб.: Символ-Плюс, 2002].
31. Fowler, M. (2003) *Patterns of Enterprise Application Architecture*. Harlow: Addison-Wesley.
32. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley [Имеется русский перевод книги: Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2006].
33. Gentleware (2003) http://www.gentleware.com/ (последнее посещение в августе 2003 г.).
34. Ghezzi, C., Jazayeri, M. and Mandrioli, D. (2003) *Fundamentals of Software Engineering.* Prentice Hall [Имеется русский перевод книги: Гецци К., Джазайери М., Мандриоли Д. Основы инженерии программного обеспечения. — СПб.: БХВ-Петербург, 2005].
35. Goodwill, J. (2002) *Mastering Jakarta Struts.* New York: John Wiley & Sons.

36. Hawryszkiewycz, I., Karagiannis, D., Maciaszek, L. and Teufel, B. (1994) Response — Requirements specific object model for workgroup computing, *Int. J. Intelligent & Cooperative Information Systems*, 3: 293–318.
37. Heldman, K. (2002) *PMP®: Project Management Professional. Study Guide*. Berkeley, CA: Sybex Inc. [Имеется русский перевод книги: Хелдман К. Профессиональное управление проектом. — М.: БИНОМ. Лаборатория знаний, 2005].
38. Henderson-Sellers, B., Constantine, L.L. and Graham, I.M. (1996) Coupling and Cohesion (Towards a Valid Metrics Suite for Object-Oriented Analysis and Design), *Object Oriented Systems*, 3: 143–58.
39. IBM (2003) http://www-3.ibm.com/software/awdtools/studioappdev/ (последнее посещение в сентябре 2003 г.).
40. ICE (2003) http://www.iceincusa.com/products_tools.htm (последнее посещение в июле 2003 г.).
41. ISO (2003) http://www.iso-9000-2000.com/ (последнее посещение в августе 2003 г.).
42. Israel, M. and Jones, J.S. (2001) *MCSE: SQL Server™ 2000. Design Study Guide.* Berkeley, CA: Sybex Inc.
43. Jacobson, I. (1992) *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley.
44. Johnson, R. (2002) *Expert One-on-One J2EE Design and Development*. Birmingham: Wrox Press.
45. Java (2003) http://java.sun.com/products/ (последнее посещение в августе 2003 г.).
46. JSF (2003) http://java.sun.com/j2ee/javaserverfaces/ (последнее посещение в июле 2003 г.).
47. JTA (2003) http://java.sun.com/products/jta/ (последнее посещение в августе 2003 г.).
48. JTS (2003) http://java.sun.com/products/jts/ (последнее посещение в августе 2003 г.).
49. JUnit (2003) www.junit.org (последнее посещение в марте 2003 г.).
50. Kim, W. (1990) *Introduction to Object-Oriented Databases*. Boston, MA: MIT Press.
51. Kleppe, A., Warmer, J. and Bast, W. (2003) *MDA Explained: The Model Driven Architecture™: Practice and Promise.* Reading, MA: Addison-Wesley.
52. Krasner, G.E. and Pope, S.T. (1988) A cookbook for using the model view controller user interface paradigm in Smalltalk-80, *J. Object-Oriented Prog.*, Aug/Sept: 26–49.
53. Kruchten, P. (1999) *The Rational Unified Process*. Reading, MA: Addison-Wesley.
54. Lakshman, B. (2002) *Oracle and Java Development*. Boston, MA: SAMS.
55. Larman, C. (2002) *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd edn. Englewood Cliffs, NJ: Prentice Hall [Имеется русский перевод книги: Ларман К. Применение UML и шаблонов проектирования. 2-е издание. — СПб.: Вильямс, 2002].

56. Lee, R.C. and Tepfenhart, W.M. (2002) *Practical Object-Oriented Development with UML and Java*. Englewood Cliffs, NJ: Prentice Hall.
57. Lethbridge, T.C. and Laganiere, R. (2001) *Object-Oriented Software Engineering. Practical Software Development Using UML and Java*. New York: McGraw-Hill.
58. Lieberherr, K.J. and Holland, I.M. (1989) Assuring good style for object-oriented programs, *IEEE Soft.*, 9: 38–48.
59. Lientz, B.P. and Rea, K.P. (2002) *Project Management for the 21st Century*, 3rd edn. New York: Academic Press.
60. Maciaszek, L.A. (1990) *Database Design and Implementation*. London: Prentice Hall.
61. Maciaszek, L.A. (2001) *Requirements Analysis and System Design: Developing Information Systems with UML*. Harlow: Addison-Wesley [Имеется русский перевод книги: Лешек А. Мацяшек. Анализ требований и проектирование систем. Разработка информационных систем с использованием UML. — СПб.: Вильямс, 2002].
62. Maciaszek, L.A. (2002) Process model for round-trip engineering with relational database, in S. Valenti (ed.) *Software Reengineering*. Hershey, PA: IRM Press, pp. 76–91.
63. Maciaszek, L.A., De Troyer, O.M.F., Getta, J.R. and Bosdriesz, J. (1996) Generalization versus aggregation in object application development — the 'AD HOC' Approach, *Proc. 7th Australasian Conf. on Information Systems ACIS'96*, Vol. 2, Hobart, Tasmania, Australia, pp. 431–42.
64. Maciaszek, L.A. and Owoc, M.L. (2001) Designing application authorizations, in E. Boyd, E. Cohen, and A.J. Zaliwski (eds) *Proc. 2001 Informing Science Conference*, Krakow, Poland. Записано на CD (ISSN 1535-0703) и на сайте http://ecommerce.lebow.drexel.edu/eli/.
65. ManagePro (2003) http://www.managepro.net/ (последнее посещение в июле 2003 г.).
66. Martin, R.C. (2003) *Agile Software Development: Principles, Patterns, and Practices*. Englewood Cliffs, NJ: Prentice Hall [Имеется русский перевод книги: Мартин Р. Быстрая разработка программ: принципы, примеры, практика. — СПб.: Вильямс, 2003].
67. MDA (2003) http://www.omg.org/mda/ (последнее посещение в июле 2003 г.).
68. Microsoft (2003) http://msdn.microsoft.com/scripting/default.htm (последнее посещение в июле 2003 г.).
69. Microsoft (2003) http://msdn.microsoft.com/ssafe/default.asp (последнее посещение в сентябре 2003 г.).
70. Microsoft (2003) http://www.microsoft.com/office/project/default.asp (последнее посещение в июле 2003 г.).
71. Newkirk, J. and Martin, R. (2001) *Extreme Programming in Practice*. Reading, MA: Addison-Wesley.
72. Netscape (2003) http://wp.netscape.com/eng/javascript/ (последнее посещение в июле 2003 г.).
73. Nomagic (2003) http://www.magicdraw.com/ (последнее посещение в августе 2003 г.).

74. Oracle (2003) http://otn.oracle.com/products/jdev/content.html (последнее посещение в сентябре 2003 г.).
75. Palisade (2003) http://www.palisade-europe.com/ (последнее посещение в июле 2003 г.).
76. Paulk, M.C., Weber, C.V., Curtis, B. and Chrissis, M.B. (1995) *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, MA: Addison-Wesley.
77. Perforce (2003) http://www.perforce.com/ (последнее посещение в сентябре 2003 г.).
78. Pfleeger, S.L. (1998) *Software Engineering. Theory and Practice*. Englewood Cliffs, NJ: Prentice Hall.
79. PMC (2003) http://www.infogoal.com/pmc/pmcswr.htm (последнее посещение в июле 2003 г.).
80. Porter, M. (1985) *Competitive Advantage: Creating and Sustaining Superior Performance.* New York: Free Press.
81. Pressman, R.S. (2001) *Software Engineering: A Practitioner's Approach*, 5th edn. New York: McGraw-Hill.
82. Primavera (2003) http://www.primavera.com (последнее посещение в июле 2003 г.).
83. Quatrani, T. (2000) *Visual Modeling with Rational Rose2000 and UML*. Reading, MA: Addison-Wesley [Имеется русский перевод книги: Кватрани Т. Визуальное моделирование с помощью Rational Rose2000 и UML. — СПб.: Вильямс, 2003].
84. Ramakrishnan, R. and Gehrke, J. (2000) *Database Management Systems*. New York: McGraw-Hill.
85. Rational (2002) *Rational Suite Tutorial*, Version 2002.05.00. Rational Software Corporation.
86. Rational (2003) http://www.rational.com (последнее посещение в августе 2003 г.).
87. Responsive (2002) http://www.responsivesoftware.com (последнее посещение в октябре 2002 г.).
88. Riel, A.J. (1996) *Object-Oriented Design Heuristics*. Reading, MA: Addison-Wesley.
89. Roman, E., Ambler, S. and Jewell, T. (2002) *Mastering Enterprise Java Beans™*, 2nd edn. New York: John Wiley & Sons.
90. Rumbaugh, J., Jacobson, I. and Booch, G. (1999) *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley.
91. RUP (2003) http://www.rational.com/products/rup/ (последнее посещение в июле 2003 г.).
92. Schach, S. (2002) *Object-Oriented and Classical Software Engineering*, 5th edn. New York: McGraw-Hill.
93. Silberschatz, A., Korth, H.F. and Sudershan, S. (2002) *Database System Concepts*, 4th edn. New York: McGraw-Hill.
94. Singh, I., Stearns, B., Johnson, M. and Enterprise Team (2002) *Designing Enterprise Applications with the J2EE™ Platform*, 2nd edn. Reading, MA: Addison-Wesley.

95. SmallWorlds (2003) http://www.thesmallworlds.com/ (последнее посещение в июне 2003 г.).
96. Sommerville, I. (2001) *Software Engineering*, 6th edn. Reading, MA: Addison-Wesley.
97. Sparx (2003) http://www.sparxsystems.com.au/ (последнее посещение в вгусте 2003 г.).
98. Standish (2003) http://www.standishgroup.com/ (последнее посещение в июле 2003 г.).
99. Sun (2002) *J2EE BluePrints*, доступно н с йте http://java.sun.com (последнее посещение в сентябре 2003 г.).
100. Sun (2002) *J2EE Platform Specification*, доступно н с йте http://java.sun.com (последнее посещение в сентябре 2003 г.).
101. Sun (2003) http://developer.java.sun.com/developer/infodocs/ (последнее посещение в июле 2003 г.).
102. Sun ONE (2003) http://wwws.sun.com/software/sundev/jde/ (последнее посещение в вгусте 2003 г.).
103. Sybase (2003) http://www.sybase.com/products/enterprisemodeling (последнее посещение в вгусте 2003 г.).
104. Telelogic (2003) http://www.telelogic.com/ (последнее посещение в вгусте 2003 г.).
105. Tomcat (2003) http://jakarta.apache.org/tomcat (последнее посещение в дек бре 2003 г.).
106. UML (2002) *Unified Modeling Language: Superstructure, Version 2 beta R1 (draft)*. U2 Partners.
107. UML (2003) http://www.rational.com/uml/resources/documentation/index.jsp (последнее посещение в июле 2003 г.).
108. UML (2003) *OMG Unified Modeling Language Specification, Version 1.5*. Needham, MA: OMG.
109. Unhelkar, B. (2003) *Process Quality Assurance for UML-Based Projects*. Reading, MA: Addison-Wesley.
110. W3C (2003) http://www.w3.org/ (последнее посещение в июле 2003 г.).
111. White, S., Fisher, M., Cattell, R., Hamilton, G. and Hapner, M. (1999) *JDBC™ API Tutorial and Reference,* 2nd edn, *Universal Data Access for the Java™ 2 Platform*. Reading, MA: Addison-Wesley.
112. yWorks (2003) http://www.yworks.de/index.htm (последнее посещение в сентябре 2003 г.).

Предметный указатель

И

К

Р

Т

Минимальные системные требования определяются соответствующими требованиями программ Adobe Reader версии не ниже 11-й либо Adobe Digital Editions версии не ниже 4.5 для платформ Windows, Mac OS, Android и iOS; экран 10"

Научное электронное издание

Серия: «Программисту»

Мацяшек Лешек А.
Лионг Брюс Ли

ПРАКТИЧЕСКАЯ ПРОГРАММНАЯ ИНЖЕНЕРИЯ НА ОСНОВЕ УЧЕБНОГО ПРИМЕРА

Ведущий редактор *Б. И. Копылов*
Художник *С. Инфантэ*
Компьютерная верстка: *В. А. Носенко*

Подписано к использованию 26.08.19.
Формат 160×220 мм

Издательство «Лаборатория знаний»
125167, Москва, проезд Аэропорта, д. 3
Телефон: (499) 157-5272
e-mail: info@pilotLZ.ru, http://www.pilotLZ.ru

Лешек Мацяшек — адъюнкт-профессор университета Маккуэри (Сидней, Австралия), специализирующийся в области компьютерных технологий. Он является научным консультантом, главным образом, по разработке и внедрению прикладного программного обеспечения предприятий, базам данных и объектно-ориентированной технологии. Ранее в переводе на русский язык была издана книга Л. Мацяшека «Анализ требований и проектирование систем. Разработка информационных систем с использованием UML» (СПб.: Вильямс, 2002).

Брюс Ли Лионг также является сотрудником университета Маккуэри. В последнее время он занимался разработкой объектно-ориентированных приложений. Специализируется в области интеграции прикладных систем предприятия, связи клиентских приложений с сетью Интернет и серверных приложений с базами данных.

Стивен Биллс — директор отделения мультимедийных систем исследовательского центра Нильсена в Сиднее, Австралия. Более 30 лет он занимался разработкой систем ПО и являлся консультантом крупных корпораций и государственных предприятий. Участвовал в создании данной книги.

Как реализуется на практике программная инженерия? Более конкретно — как следует разрабатывать современное программное обеспечение предприятия? Данная книга, основанная на учебном примере, дает ответы на эти вопросы. Она изменит представление студентов о программной инженерии и поможет профессионалам в области информационных технологий в их практической деятельности. Она также послужит источником новых идей и направлений в разработке систем ПО.

Для разработчиков сложного программного обеспечения, а также для студентов вузов, специализирующихся в вопросах создания современного ПО.