O'REILLY®

Фундаментальный подход к программной архитектуре

паттерны, свойства, проверенные методы

Марк Ричардс и Нил Форд

Фундаментальный подход к программной архитектуре

паттерны, свойства, проверенные методы

Марк Ричардс и Нил Форд

ПИТЕР®

Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018-02
УДК 004.415
Р56

Ричардс Марк, Форд Нил

Р56 Фундаментальный подход к программной архитектуре: паттерны, свойства, проверенные методы. — СПб.: Питер, 2023. — 448 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1842-7

Архитекторы ПО стабильно входят в десятку самых высокооплачиваемых профессий. Но до сих пор не было реального руководства, которое позволило бы разработчикам стать архитекторами. И вот наконец появилась книга, в которой дается всеобъемлющий обзор разнообразных аспектов архитектуры программного обеспечения. Начинающие и уже состоявшиеся архитекторы найдут в ней паттерны архитектур, определения компонентов, приемы построения эволюционных архитектур и множество других тем.

Марк Ричардс и Нил Форд обладают бесценным практическим опытом, профессионально занимаются этой темой, уделяя особое внимание принципам построения архитектуры, применимым ко всем технологическим стекам. Они предлагают современный взгляд на архитектуру ПО с учетом всех нововведений последнего десятилетия.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018-02
УДК 004.415

ISBN 978-1492043454 англ.

ISBN 978-5-4461-1842-7

Краткое содержание

Часть III
ТЕХНИЧЕСКИЕ ПРИЕМЫ И ГИБКИЕ НАВЫКИ

Оглавление

ЧАСТЬ I. ОСНОВЫ

ЧАСТЬ III. ТЕХНИЧЕСКИЕ ПРИЕМЫ И ГИБКИЕ НАВЫКИ

Отзывы о книге «Фундаментальный подход к программной архитектуре»

Нил и Марк — не только выдающиеся архитекторы ПО, но и замечательные наставники. В книге «Фундаментальный подход к программной архитектуре» благодаря своему многолетнему опыту им удалось в сжатом виде изложить обширнейшую тему архитектуры ПО. Кем бы вы ни были, новичком или архитектором с многолетней практикой, эта книга поможет поднять ваш профессиональный уровень. Очень жаль, что она не появилась на заре моей карьеры.

Натаниэль Шутта (Nathaniel Schutta),
архитектор и девелопер-адвокат VMware, ntschutta.io

Марк и Нил поставили перед собой грандиозную цель — разъяснить многочисленные и многоуровневые фундаментальные принципы, без знания которых невозможно преуспеть в профессии архитектора ПО, — и у них это получилось. Сфера архитектуры ПО постоянно совершенствуется, поэтому профессия требует владения разнообразными навыками, широкого кругозора и глубины познаний. Эта книга послужит руководством для многих на пути их становления архитекторами программного обеспечения.

Ребекка Дж. Парсонс (Rebecca J. Parsons),
CTO, ThoughtWorks

Марк и Нил дают реальные практические советы, позволяющие специалистам достичь архитектурного совершенства. Они рассказывают о наиболее распространенных архитектурных свойствах и компромиссах, знание которых необходимо для успешной работы.

Кэсси Шум (Cassie Shum),
технический директор, ThoughtWorks

Предисловие. Развенчание аксиом

Аксиома

Утверждение или суждение, которое считается установленным, принятым или бесспорно истинным.

Математики выстраивают теории, основанные на аксиомах, то есть на бесспорно истинных предположениях. Архитекторы программного обеспечения (ПО) также выстраивают теории, но по сравнению с математикой мир ПО менее строгий: фундаментальные вещи, включая аксиомы, на которых мы основываем наши теории, продолжают меняться весьма быстрыми темпами.

Экосистема разработки ПО постоянно находится в состоянии динамического равновесия: несмотря на сбалансированность ее состояния в любой отдельно взятый момент времени, в долгосрочной перспективе она демонстрирует сугубо *динамическое* поведение. Отличным современным примером такой особенности является расцвет контейнеризации и сопутствующие этому изменения: десять лет назад таких инструментов, как Kubernetes[1], еще не было, а теперь проводятся целые конференции для его пользователей. Изменения программной экосистемы носят хаотичный характер: одно небольшое изменение вызывает другое небольшое изменение, и так, повторяясь сотни раз, этот процесс порождает новую экосистему.

На архитекторов возлагается ответственная задача: подвергать сомнению доставшиеся нам в наследство предположения и аксиомы. Многие книги об архитектуре ПО были написаны во времена, весьма слабо напоминающие современный мир. Авторы считают, что необходимо регулярно подвергать сомнению все аксиомы в областях улучшения инженерных практик, эксплуатационных экосистем, разработки программного обеспечения, то есть всех составляющих хаотичного динамического равновесия, в котором ежедневно приходится работать архитекторам и разработчикам.

Те, кто на протяжении длительного времени внимательно следил за развитием архитектуры ПО, стали свидетелями эволюции возможностей. Все инновации,

[1] https://kubernetes.io/

начиная с экстремального программирования[1], с последующим переходом к непрерывной доставке, DevOps-революции, микросервисам, контейнеризации и современным облачным ресурсам, приводили к новым возможностям и компромиссам. По мере изменения возможностей менялись и взгляды архитекторов на производство программных продуктов. В течение многих лет не без иронии говорилось, что архитектура — «это то, что изменить впоследствии весьма непросто». Затем появились микросервисы, где *изменение* является важнейшей особенностью дизайна.

Каждая новая эра требует новых технологических приемов, инструментов, метрик, паттернов и множества других изменений. В этой книге архитектура программного обеспечения рассматривается в современном восприятии, с учетом всех инноваций последнего десятилетия, а также некоторых новых показателей и метрик, соответствующих актуальным структурам и перспективам.

Разработчики давно хотели превратить разработку ПО из *ремесла*, в котором талантливые мастера могут создавать единичные проекты, в *инженерную* дисциплину, подразумевающую повторяемость, строгость и эффективный анализ. Хотя программная инженерия все еще ощутимо отстает от других инженерных дисциплин (справедливости ради следует отметить, что создание ПО — очень молодая дисциплина по сравнению с другими), архитекторы внесли в нее существенные улучшения, с которыми нам предстоит познакомиться. В частности, современные методы гибкой разработки с применением Agile-технологии позволили добиться больших успехов в разработке новых типов систем, проектируемых архитекторами.

Мы также рассмотрим чрезвычайно важный вопрос *анализа компромиссов*. Разработчику ПО легко увлечься определенной технологией или подходом. Но архитекторы всегда должны трезво оценивать все хорошее, плохое и уродливое в каждом варианте, ведь в реальном мире практически ничто не предлагает удобных двоичных вариантов выбора — все в нем соткано из компромиссов. С этой прагматической точки зрения мы стремимся исключить оценочные суждения о технологиях и сосредоточиться на анализе компромиссов, чтобы вооружить наших читателей аналитическим взглядом на выбор технологий.

Наша книга не сможет волшебным образом превратить вас в первоклассного специалиста по архитектуре ПО — это достаточно сложная область знаний с множеством нюансов. Мы хотим предложить действующим и активно развивающимся архитекторам полезный и актуальный обзор архитектуры ПО и ее

[1] http://www.extremeprogramming.org

многочисленных аспектов, от структуры до нужных архитектору навыков межличностного общения. Хотя в этой книге рассматриваются хорошо известные паттерны, мы применяем новый подход, опираясь на накопленный нами опыт, используемые инструменты, практику проектирования и другие материалы. Мы берем многие существующие в архитектуре программного обеспечения аксиомы и переосмысливаем их в свете текущего состояния экосистемы; мы проектируем архитектуры с учетом современного ландшафта.

Условные обозначения

В этой книге используются следующие условные обозначения:

Курсив

Курсивом выделены новые термины или важные понятия.

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена файлов и их расширения.

`Моноширинный полужирный шрифт`

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

`Моноширинный курсив`

Показывает текст, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок и других элементов интерфейса, каталогов.

Этот рисунок указывает на совет или предложение.

Использование исходного кода примеров

Вспомогательные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу http://fundamentalsofsoftwarearchitecture.com. Если у вас возникнут вопросы технического характера по использованию примеров кода, направляйте их по электронной почте на адрес bookquestions@oreilly.com.

В общем случае все примеры кода из книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

У этой книги есть веб-страница с перечнем замеченных опечаток и другой дополнительной информацией, доступ к которой можно получить по адресу https://oreil.ly/fundamentals-of-software-architecture.

Благодарности

Мы, Марк и Нил, благодарим всех, кто посещал наши занятия, семинары, серии конференций, собрания групп пользователей, а также всех людей, прослушавших версии этого материала и предоставивших свои бесценные отзывы. Также мы благодарим команду издательства O'Reilly, оказавшую максимально возможное содействие в написании книги. Спасибо директору No Stuff Just Fluff Джею Циммерману (Jay Zimmerman), организовавшему серию конференций, в результате которых удалось расширить техническое содержимое книги, и всем другим ораторам, чьи отзывы и мокрые от наших слез плечи мы бесконечно ценим. Мы также благодарны ряду случайно оставшихся оазисов здравомыслия, способных вдохновить на смелые замыслы, в виде групп с именами, похожими на Pasty Geeks и Hacker B&B.

Благодарности от Марка Ричардса

Вдобавок к предыдущим благодарностям хочу сказать спасибо своей любимой жене Ребекке. Взвалив на себя все домашние дела и пожертвовав написанием своей собственной книги, ты позволила мне проводить дополнительные консультации и выступать на большем количестве конференций и учебных курсов, что дало мне возможность попрактиковаться и отточить материал для этой книги. Ты лучшая.

Благодарности от Нила Форда

Хочу поблагодарить свою большую семью — коллектив ThoughtWorks, и в частности Ребекку Парсонс (Rebecca Parsons) и Мартина Фаулера (Martin Fowler). Компания ThoughtWorks — замечательная команда, способная создавать ценности для клиентов, внимательно отслеживая, как работают те или иные вещи, чтобы мы могли их улучшить. Компания ThoughtWorks поддерживала работу над этой книгой всевозможными способами и продолжает повышать профессиональный уровень своих сотрудников, ведущих повседневную борьбу с трудностями и вдохновляющих на победу над ними. Хочу также поблагодарить работников соседнего коктейль-клуба за возможность отдохнуть от рутинной работы. И наконец, хочу сказать спасибо своей жене Кэнди, чье терпеливое отношение к написанию мною книги и участию в различных конференциях воистину не знало границ. Не один десяток лет она поддерживала во мне реальный взгляд на вещи и здравомыслие, настраивая на плодотворный труд, и я надеюсь, что еще не одно десятилетие она будет любовью всей моей жизни.

От издательства

В книге вы встретитесь с многообразными паттернами (и антипаттернами) проектирования. Названия некоторых из них уже устоялись в русском языке (в этом случае в скобках для информации приводится английское название), другие пока встречаются в литературе только на английском (в этом случае в скобках приводится перевод). Некоторые обычно употребляются как имена собственные, другие — как нарицательные.

Для вашего удобства мы собрали в таблицу небольшой словарь паттернов, упоминаемых в книге.

СЛОВАРЬ ПАТТЕРНОВ И АНТИПАТТЕРНОВ ПРОЕКТИРОВАНИЯ

Bullet-Riddled Corpse	Тело, изрешеченное пулями
Covering Your Assets (антипаттерн)	Излишняя перестраховка
Email-Driven Architecture (антипаттерн)	Архитектура, управляемая имейлами
Groundhog Day (антипаттерн)	День сурка
Антипаттерн архитектурной воронки	Architecture sinkhole pattern
Антипаттерн бессистемной архитектуры	Accidental architecture antipattern
Антипаттерн подразумеваемой архитектуры	Architecture by implication antipattern
Башня из слоновой кости (антипаттерн)	Ivory Tower
Большой ком грязи (антипаттерн)	Big Ball of Mud
Душитель	Strangler
Замороженный троглодит (антипаттерн)	Frozen Caveman
Западня сущности (антипаттерн)	Entity trap
Локатор сервисов	Service Locator
Модель — Представление — Контроллер	Model-View-Controller
Паттерн Backends for Frontends (BFF)	Обычно не переводится

Паттерн sidecar	Обычно не переводится
Паттерн делегирования	Business delegate pattern
Паттерн саги	Saga pattern
Паттерн событий рабочего процесса	Workflow event pattern
Паттерн фронт-контроллера	Front controller pattern
Стратегия	Strategy
Шаблонный метод	Template Method

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1

Введение

Специальность «архитектор программного обеспечения» находится в верхних строчках многочисленных рейтингов вакансий по всему миру. Но когда читатели смотрят на *другие* специальности в этих списках (например, фельдшер или экономист), то видят вполне определенный путь профессионального роста. Так почему же развитие карьеры архитектора программного обеспечения мало кому понятно?

Во-первых, в самой профессиональной среде отсутствует четкое определение архитектуры программного обеспечения (ПО). При чтении установочных лекций студенты часто просят дать краткое описание того, чем занимается архитектор ПО, но мы отвечаем уклончиво. И не мы одни. В своей знаменитой статье «Who Needs an Architect?» («Кому нужен архитектор?»)[1] Мартин Фаулер (Martin Fowler), как известно, отказался от попытки дать четкое определение соответствующему понятию, вернувшись вместо этого к известной цитате:

> Архитектура — это о важном... что бы это ни было.
>
> *Ральф Джонсон (Ralph Johnson)*

Однажды нам все же пришлось создать ментальную карту (майндмэп), показанную на рис. 1.1, которая, к сожалению, не является исчерпывающей, но при этом показывает масштаб архитектуры ПО. Через некоторое время мы предложим свое определение этому понятию.

Во-вторых, как показано на схеме, зона ответственности архитектора ПО весьма велика и постоянно расширяется. Лет десять назад архитекторы программного обеспечения занимались чисто техническими аспектами архитектуры: модульностью, компонентами и паттернами. Но благодаря появлению новых архитектурных стилей с более широким спектром возможностей (например,

[1] https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf

микросервисов, обязанности архитектора значительно увеличились. Множество пересечений архитектуры с остальными организационными задачами будет рассмотрено в следующем далее разделе «Пересечение архитектуры и...» (с. 37).

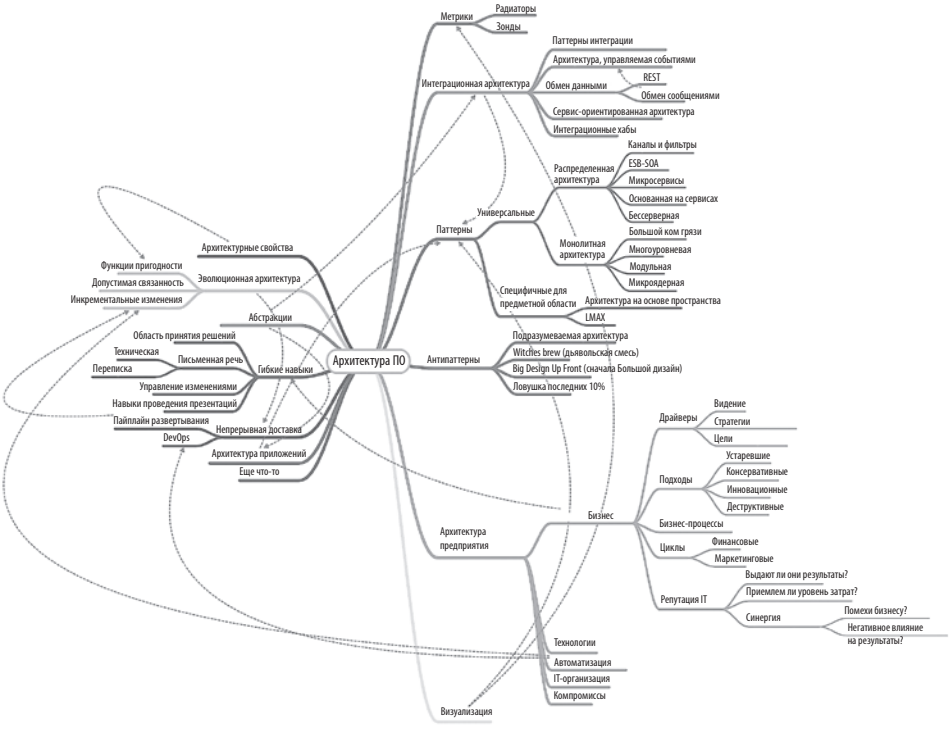


Рис. 1.1. Архитектор ПО должен обладать техническими знаниями, гибкими навыками (soft skills), а также понимать бизнес-процессы и множество других вещей

В-третьих, архитектура ПО — это постоянно меняющаяся концепция, потому что экосистема разработки программного обеспечения слишком быстро развивается. Любое принятое сегодня определение через несколько лет безнадежно устареет. Определение архитектуры ПО в Википедии дает вполне приемлемое общее представление, но многие положения уже неактуальны, например: «Архитектура программного обеспечения относится к выбору фундаментальных структурных решений, внесение изменений в которые после их реализации обходится слишком дорого». Но с тех пор специалисты разработали современные архитектурные стили (микросервисы) с идеей поэтапного встраивания, при котором внесение структурных изменений уже не обходится «слишком дорого». Конечно, эта возможность означает компромисс с другими вопросами,

например связанностью. Во многих книгах по программной архитектуре она представляется чисто статической, и считается, что после однократного внедрения ее можно спокойно игнорировать. Однако неизбежная динамическая природа архитектуры ПО, да и самого определения этого понятия прослеживается во всей книге.

В-четвертых, основной объем сведений об архитектуре ПО несет чисто историческую нагрузку. Читателям страницы Википедии, несомненно, бросится в глаза множество сокращений и перекрестных ссылок на целую вселенную знаний. Но многие из этих сокращений, по сути, устарели или не прижились. Даже те методы, которые несколько лет назад представлялись правильными, сейчас не работают, потому что изменился контекст. История развития архитектуры ПО насыщена принятыми решениями, которые в дальнейшем привели к разрушительным последствиям. Многие из этих уроков мы рассмотрим здесь. Так почему же эта книга по основам архитектуры программного обеспечения появилась именно сейчас? В нашем мире все постоянно меняется, и область архитектуры ПО не исключение. Новые технологии, методы, возможности... по сути, в последнем десятилетии проще найти что-то, что не изменилось, чем перечислить все изменения. И архитекторы программного обеспечения вынуждены принимать решения в этой постоянно меняющейся экосистеме. Поскольку изменения касаются практически всего, включая и сами основы принятия наших решений, архитекторы должны каждый раз пересматривать ряд аксиом, на которых базировались предыдущие работы. Например, в ранее вышедших книгах об архитектуре ПО не рассматривалось влияние методологии DevOps, поскольку на момент написания этих книг ее просто не было.

Изучая архитектуру, нужно осознавать, что, как и во многих других искусствах, ее премудрости можно усвоить только в контексте. Многие решения принимаются архитекторами на основе особенностей той среды, в которой они оказались. Например, основной целью архитектуры ПО конца XX века было достижение более эффективного использования общих ресурсов, поскольку вся инфраструктура того времени состояла из дорогих коммерческих продуктов: операционных систем, серверов приложений, серверов баз данных и т. д. Представьте, что вы приходите в дата-центр образца 2002 года и говорите начальнику отдела эксплуатации: «У меня есть отличная идея по внесению кардинальных изменений в сам стиль архитектуры: запуск каждого сервиса будет на отдельной, предназначенной только для него машине, с его собственной выделенной базой данных». (Это суть того, что нам сейчас известно как микросервисы.) «И для этого мне понадобятся 50 лицензий на Windows, 30 лицензий на сервер приложения и как минимум 50 лицензий на сервер базы данных». В 2002 году попытка создания архитектуры, похожей на микросервисы, обошлась бы слишком дорого. А вот

с появлением в последние годы открытого исходного кода в сочетании с новой революционной практикой разработки и сопровождения программных продуктов в рамках методологии DevOps построение именно такой архитектуры вполне оправданно. Читатели должны понимать, что все архитектуры являются продуктом конкретных обстоятельств.

Определение архитектуры программного обеспечения

Многие специалисты пытались дать точное определение понятию «архитектура программного обеспечения». Одни считали, что архитектура ПО является по своей сути неким *планом системы*, а другие определяли ее как *дорожную карту* разработки системы. Но эти общие определения не отвечали на вопрос о конкретном содержимом плана или карты. Например, что именно архитектор должен исследовать при анализе архитектуры?

На рис. 1.2 показан способ осмысления архитектуры ПО. Согласно этому определению, она состоит из *структуры* системы (показанной широкими серыми линиями, поддерживающими архитектуру) в сочетании со *свойствами архитектуры* (выражаемыми словами с окончанием *-ость*), которые должны поддерживаться системой, *архитектурными решениями* и, наконец, *принципами проектирования*.

Под *структурой* системы (рис. 1.3) понимается тип архитектурного стиля (или стилей), в котором реализована система (например, микросервисы, многоуровневая система или же микроядро). Составить полное представление об архитектуре исключительно по структуре невозможно. Предположим, к примеру, что архитектора попросили дать описание архитектуры, а он ответил: «Это архитектура микросервисов». В этом случае он рассказал только лишь о *структуре* системы, но не об ее *архитектуре*. Для полного представления об архитектуре системы нужны также ее свойства, архитектурные решения и принципы проектирования.

Еще одним элементом определения архитектуры ПО являются ее свойства (рис. 1.4). Свойства архитектуры определяют критерии ее успеха, не имеющие ничего общего с функциональными возможностями системы. Заметьте, что все перечисленные свойства не требуют никаких сведений о функциональности системы и в то же время нужны для ее корректной работы. Свойства архитектуры играют настолько важную роль, что их определению и пониманию посвящено несколько глав этой книги.

Рис. 1.2. Архитектура состоит из структуры в сочетании со свойствами (выражаемыми словами с окончанием *-ость*), архитектурными решениями и принципами проектирования

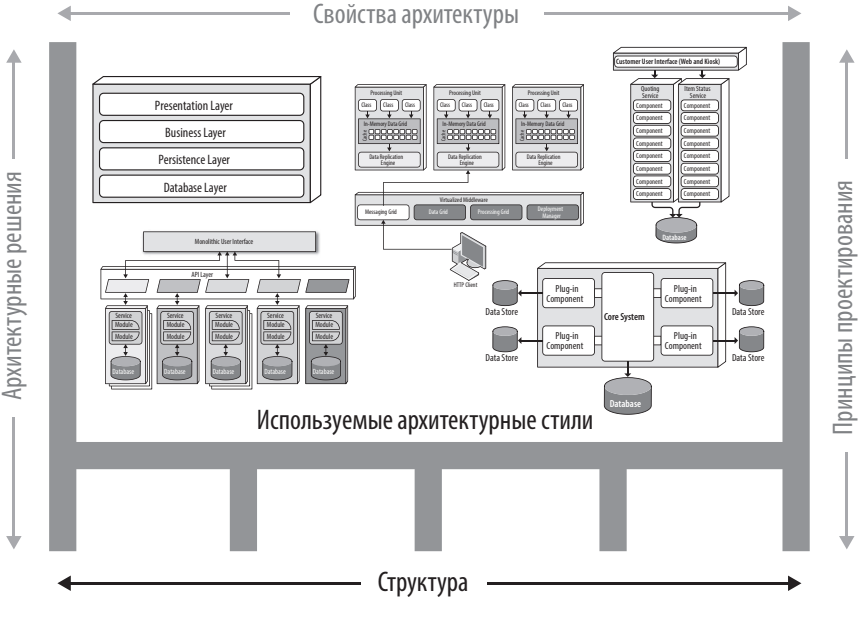


Рис. 1.3. Под структурой подразумевается тип архитектурных стилей, используемых в системе

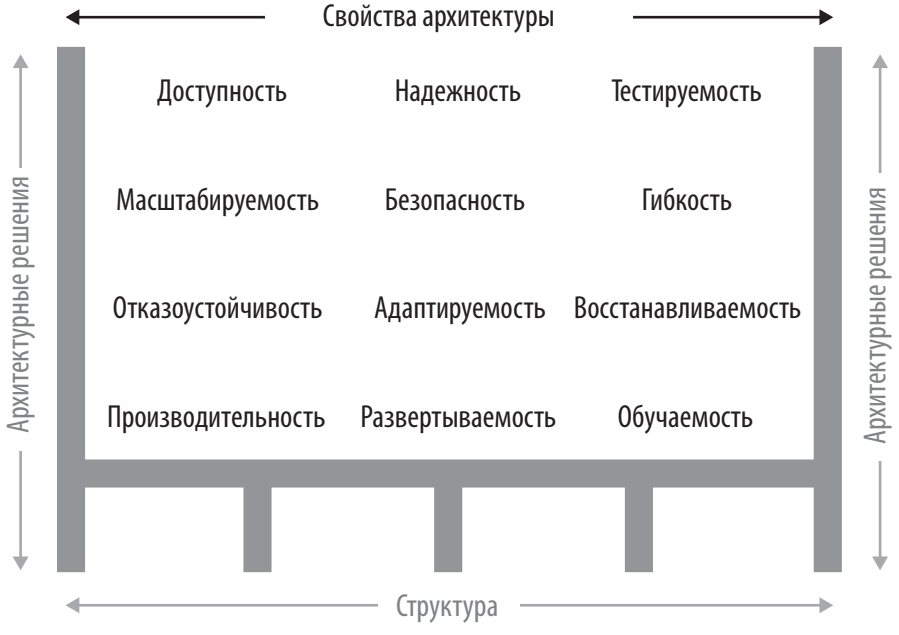


Рис. 1.4. Свойства архитектуры относятся к тому, что должно поддерживаться системой и описывается словами с окончанием *-ость*

Следующим фактором, определяющим архитектуру ПО, являются *архитектурные решения*. Ими определяются правила, по которым должна выстраиваться система. Например, архитектор должен решить, что доступ к базе данных (БД) в многоуровневой архитектуре должен быть только с бизнес-уровня и с уровня сервисов (рис. 1.5), а прямые обращения к ней с уровня представления должны быть запрещены. Архитектурные решения формируют ограничения системы и дают командам разработчиков понять, что разрешено, а что нет.

Если из-за какого-то условия или другого ограничения конкретное архитектурное решение не может быть реализовано в одной из частей системы, это решение (или правило) может быть нарушено посредством так называемого *отступления (variance)*. Модели отступлений, используемые наблюдательным советом за архитектурой (architecture review board, ARB) или главным архитектором, имеются в большинстве организаций. Эти модели определяют процесс поиска отступления от конкретного стандарта или архитектурного решения. Исключение из конкретного архитектурного решения анализируется советом ARB (или главным архитектором, если такого совета нет) и либо утверждается, либо отклоняется на основе обоснований и компромиссов.

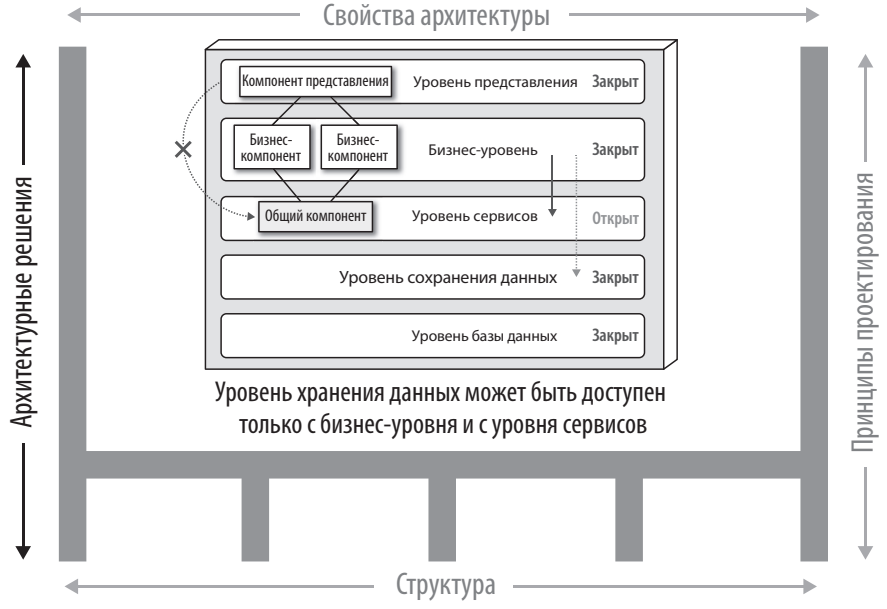


Рис. 1.5. Архитектурные решения являются правилами построения систем

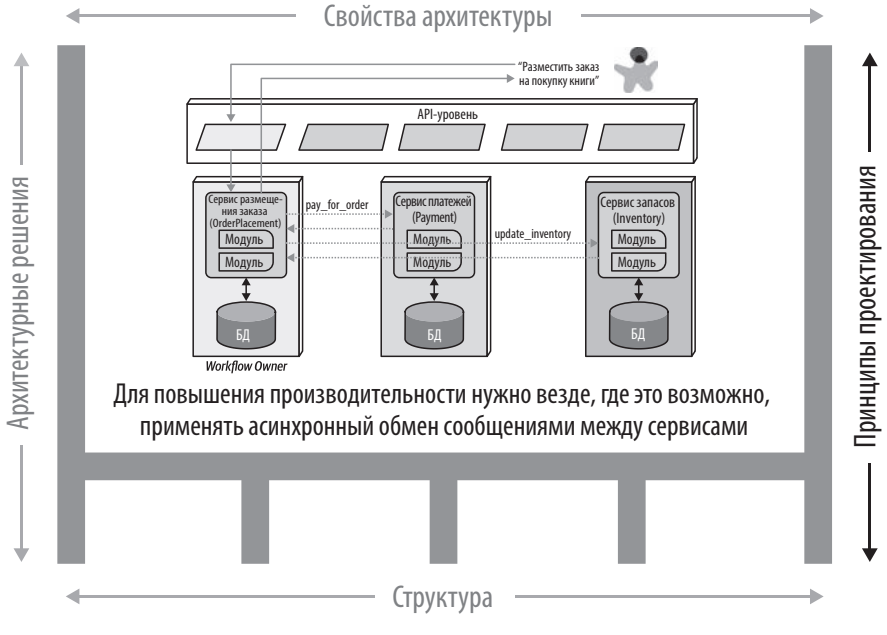


Рис. 1.6. Принципы проектирования являются руководящими установками выстраивания систем

Последним фактором, определяющим архитектуру, являются *принципы проектирования*. Принцип проектирования отличается от архитектурного решения тем, что он является *руководящей установкой*, а не строго определенным правилом. Например, в принципе проектирования, показанном на рис. 1.6, говорится, что для достижения высокой производительности в рамках архитектуры микросервисов команды разработчиков должны использовать асинхронный обмен сообщениями между микросервисами.

Архитектурное решение (правило) никогда не сможет охватить все условия и варианты обмена данными между сервисами, поэтому для обеспечения руководящей установки по предпочтительному методу (в данном случае это асинхронный обмен сообщениями) может использоваться принцип проектирования, позволяющий разработчику выбрать в конкретных обстоятельствах наиболее подходящий протокол обмена данными (например, REST или gRPC).

Ожидания от работы архитектора

Определить роль архитектора ПО ничуть не легче, чем дать определение самой архитектуре. Его обязанности могут варьироваться от задач опытного программиста до специалиста, формирующего стратегическое техническое направление для компании. Так что вместо пустой траты времени на попытку определения этой роли сконцентрируемся на *ожиданиях* от работы архитектора.

Независимо от каких-либо конкретных полномочий, титулов или должностных обязанностей, от архитектора ПО ожидается:

- принятие архитектурных решений;
- постоянный анализ архитектуры;
- своевременное следование последним тенденциям;
- контроль за выполнением принятых решений;
- обладание обширными знаниями и опытом;
- компетентность в нужной области бизнеса;
- владение навыками межличностного общения;
- четкое понимание политики компании.

Эффективная и плодотворная работа в должности архитектора программного обеспечения невозможна без понимания и осуществления каждого из этих ожиданий.

Принятие архитектурных решений

От архитектора ожидаются архитектурные решения и определение принципов проектирования, которые будут служить руководством для принятия технологических решений внутри команды, отдела или всего предприятия.

Ключевым здесь является слово *руководство*. Архитектор должен *выдавать установки*, а не *определять выбор* технологических решений. Например, архитектор может решить, что для разработки внешнего интерфейса нужно использовать библиотеку React.js. В этом случае он принимает техническое решение, а надо бы — архитектурное решение: определить принцип проектирования, помогающий команде разработчиков сделать свой выбор. Архитектор должен был бы рекомендовать команде разработчиков *использовать для веб-разработки внешнего интерфейса реактивно-ориентированную среду* и тем самым направить на выбор между Angular, Elm, React.js, Vue или любым другим фреймворком на реактивной основе.

Управлять выбором технологии через архитектурные решения и принципы проектирования довольно сложно. Ключом к принятию эффективных архитектурных решений является вопрос о том, помогает ли архитектурное решение *направлять* команды на выбор правильных технических решений или же такой выбор *делается* за них. И тем не менее бывает так, что для соблюдения конкретных архитектурных свойств, например масштабируемости, производительности или доступности, архитектору приходится принимать те или иные технологические решения. В таком случае это все равно будет считаться архитектурным решением, даже если им определяется конкретная технология. Архитекторы часто испытывают трудности с поиском правильного курса, поэтому архитектурным решениям целиком посвящена глава 19.

Постоянный анализ архитектуры

Архитектор должен постоянно анализировать архитектуру и текущую технологическую среду, а затем рекомендовать решения по их совершенствованию.

Это позволяет поддерживать жизнеспособность архитектуры, то есть оценивать, насколько архитектура, определенная года три (или более) назад, актуальна *на сегодняшний день* с учетом изменений как в бизнесе, так и в технологии. Исходя из нашего опыта, далеко не все архитекторы уделяют достаточно внимания постоянному анализу существующих архитектур. В результате большинство архитектур подвержено структурному распаду, происходящему из-за вноси-

мых разработчиками изменений в код или в проект, что влияет на требуемые свойства архитектуры, например на производительность, доступность или масштабируемость.

Другие аспекты, о которых часто забывают архитекторы, — среды тестирования и выпуска программного продукта. Возможность гибкой модификации кода дает очевидные преимущества, но если командам разработчиков требуются недели на тестирование продукта и месяцы на его релиз, значит, архитекторам не удалось достичь гибкости в плане общей архитектуры.

Чтобы убедиться в состоятельности архитектуры, архитектор должен проводить комплексный анализ изменений в технологиях и в областях решаемых задач. Хотя в объявлениях о вакансии подобное требование к уровню квалификации фигурирует довольно редко, архитектор все же должен соответствовать и этому ожиданию.

Своевременное следование последним тенденциям

Архитектор должен быть в курсе последних тенденций в технологии и в бизнес-секторе.

Чтобы сохранять свою востребованность (и не потерять работу!), разработчики должны быть в курсе технологических новинок, относящихся к их повседневной деятельности. К архитектору предъявляются еще более жесткие требования: он должен следить за текущими изменениями не только в технологии, но и в своем бизнес-секторе. Принимаемые архитектором решения носят долговременный характер и трудно поддаются изменениям. Четкое понимание основных тенденций помогает архитектору подготовиться к будущим вызовам и принимать правильные решения.

Отслеживание тенденций и следование им даются нелегко, особенно архитектору ПО. Различные технологии и ресурсы, позволяющие успешно решать эту задачу, будут рассмотрены в главе 24.

Контроль за выполнением принятых решений

Архитектор должен обеспечивать соответствие архитектурным решениям и принципам проектирования.

Контроль за выполнением принятых решений означает, что архитектор обязан постоянно проверять, что разработчики следуют принятым, задокументированным и доведенным до них архитектурным решениям и принципам

проектирования. Рассмотрим такую ситуацию. Архитектор принял решение, что в многоуровневой архитектуре доступ к базе данных возможен только из бизнес- и сервис-уровня (и невозможен с уровня представления). Следовательно, даже для самых простых обращений к базе данных уровень представления должен пройти через все уровни архитектуры. Разработчик пользовательского интерфейса может не согласиться с этим и с целью повышения производительности обращаться к базе данных (или к уровню постоянного хранения данных) напрямую. Но архитектор принял данное архитектурное решение по конкретной причине: контроль изменений. Закрытие уровней позволяет вносить изменения в базу данных, не затрагивая уровень представления. Если не обеспечить соблюдение архитектурных решений, могут быть допущены такие нарушения, при которых архитектура перестанет отвечать требуемым свойствам (выражаемым словами с окончанием *-ость*), а приложение или система не будут работать так, как ожидалось.

В главе 6 мы подробнее поговорим об оценке соблюдения решений и принципов с помощью автоматизированных функций пригодности и соответствующих автоматизированных инструментов.

Обладание обширными знаниями и опытом

Ожидается, что архитектор обладает опытом работы с многочисленными и разнообразными технологиями, средами разработки, платформами и средами окружения.

Данное требование не означает, что архитектор должен быть великим знатоком каждой среды разработки, платформы и языка, скорее он должен быть знаком с различными технологиями. Большинство современных сред имеет неоднородный характер, и архитектор должен по крайней мере знать возможности взаимодействия нескольких систем и сервисов независимо от языков, платформ и технологий, использованных при написании таких систем или сервисов.

Лучший способ соответствовать этому требованию — это постоянно выходить из зоны комфорта. Сосредоточенность на одной технологии или платформе — это не что иное, как тихая гавань. Успешный архитектор ПО должен активно изучать различные языки программирования, платформы и технологии. Для соответствия данному ожиданию лучше расширять, а не углублять свои технические знания. Технический кругозор включает то, что вам знакомо в общих чертах, в сочетании с тем, в чем вы хорошо разбираетесь. Например, для архитектора намного более ценно знать 10 различных средств кэширования и все их «за» и «против», чем быть узким специалистом по одному из них.

Компетентность в нужной области бизнеса

Ожидается, что архитектор достаточно хорошо ориентируется в нужной области бизнеса.

Успешные архитекторы программного обеспечения разбираются не только в технологии, но и в нужной области бизнеса применительно к решаемым задачам. Без знаний в бизнес-сфере трудно определить круг решаемых задач, цели и требования, а это затрудняет разработку оптимальной архитектуры, отвечающей потребностям бизнеса. Предположим, что архитектор в крупном финансовом учреждении не разбирается в общих финансовых понятиях: например, не знает, что такое средний индекс направленности, алеаторные сделки, рост цен или неприоритетный долг. Без этих знаний архитектор не сможет общаться со стейкхолдерами и бизнес-пользователями и быстро утратит доверие.

Наиболее успешные из известных нам архитекторов обладают практической технической эрудицией в сочетании с глубоким пониманием нужной бизнес-сферы. Эти архитекторы ПО способны вполне успешно общаться с руководителями высшего звена и с бизнес-пользователями, используя знания предметной области и язык, известный и понятный этим заинтересованным сторонам. В свою очередь, это придает им уверенности в том, что архитектор программного обеспечения знает, что делает, и компетентен в области создания эффективной и правильной архитектуры.

Владение навыками межличностного общения

Ожидается, что архитектор обладает навыками межличностного общения, включая работу в команде, содействие сотрудникам и лидерские качества.

Ожидать от большинства разработчиков и архитекторов наличия исключительных лидерских качеств и навыков межличностного общения довольно сложно. Обладая математическим мышлением, разработчики и архитекторы любят решать проблемы, связанные с техникой, а не с людьми. Однако, как говорил Джеральд Вайнберг (Gerald Weinberg): «Что бы они вам ни говорили, это всегда человеческая проблема». Предполагается, что архитектор не только обеспечит техническое руководство командой, но и будет руководить командой разработчиков в процессе реализации архитектуры. Лидерские качества — это как минимум половина того, чем должен обладать архитектор программного обеспечения независимо от его роли или должности.

В профессиональной среде много архитекторов ПО, которые сражаются за ограниченное число соответствующих вакансий. Наличие сильных лидерских

качеств и развитых навыков межличностного общения — это то, что позволит архитектору выделиться из общей массы себе подобных. Мы встречали многих специалистов, которые были превосходными техническими экспертами, но никудышными архитекторами из-за неспособности руководить командами, обучать и наставлять разработчиков, а также успешно внедрять идеи, архитектурные решения и принципы. Излишне говорить, что таким архитекторам сложно сохранять за собой должность и работу.

Четкое понимание политики компании

Ожидается, что архитектор разбирается в политическом климате компании и умеет в нем ориентироваться.

Может показаться, что в книге, посвященной архитектуре программного обеспечения, обсуждение темы переговоров и ориентации в офисной политике выглядит как-то странно. Чтобы продемонстрировать важность и необходимость навыка ведения переговоров, рассмотрим ситуацию, в которой разработчик решает применить паттерн Стратегии ради сокращения общей цикломатической сложности конкретного фрагмента довольно непростого кода. Кого это действительно волнует? Можно было бы, конечно, поаплодировать за использование подобного паттерна, но почти во всех случаях разработчик не нуждается в одобрении такого решения.

А теперь рассмотрим другую ситуацию. Архитектор, отвечающий за крупную систему управления взаимоотношениями с клиентами (CRM), испытывает проблемы с управлением доступом к базе данных из других систем, защитой конкретных данных клиентов и внесением изменений в схему базы данных, поскольку база данных CRM используется слишком большим количеством других систем. Исходя из этого, архитектор принимает решение по созданию так называемого *application silos*, то есть хранилища, принадлежащего исключительно конкретному приложению. Получается, что каждая прикладная база данных доступна только из приложения, которому она принадлежит. Принятие такого решения позволит архитектору лучше контролировать клиентские данные, их безопасность и возможность их изменения. Но, в отличие от предыдущей ситуации с разработчиком, против этого решения будут практически все остальные сотрудники компании (за исключением, конечно, команды разработчиков CRM-приложения), ведь в данных управления взаимоотношениями с клиентами нуждаются и другие приложения. Если они больше не смогут обращаться к базе данных напрямую, то запрашивать эти данные придется у CRM-системы, для чего понадобятся вызовы через REST, SOAP или какой-либо иной протокол удаленного доступа.

Главное здесь то, что *оспариваться будет практически каждое решение архитектора*. С архитектурными решениями будут спорить заказчики программного продукта, руководители проекта и стейкхолдеры бизнес-сферы по причине увеличения затрат или сроков выполнения заказа. Эти решения будут также оспариваться разработчиками, считающими свой подход более удачным. В любом случае архитектор должен ориентироваться в политике компании и применять основные навыки ведения переговоров, чтобы добиться одобрения своих решений. Данное утверждение может вызвать раздражение у архитектора ПО, поскольку большинство решений, которые принимались им в ранге разработчика, не требовали одобрения или даже согласования. Вопросы программирования, будь то структура кода, дизайн класса, выбор паттерна проектирования, а иногда даже и выбор языка, являются частью искусства программирования. Но архитектор, которому теперь наконец-то можно принимать более масштабные и важные решения, должен обосновывать каждое из них и бороться за их воплощение в жизнь. Навыки ведения переговоров, равно как и навыки лидерства, настолько важны и необходимы, что их приобретению посвящена целая глава 23.

Пересечение архитектуры и…

За последние десятилетия сфера применения архитектуры программного обеспечения расширилась и стала включать в себя больше ответственных задач и перспектив. Еще лет десять назад типичные отношения между архитектурой и функциональными аспектами носили договорной и формальный характер с множеством чисто бюрократических моментов. Большинство компаний, стараясь избежать сложностей, связанных с поддержкой собственных операций, зачастую прибегали к аутсорсингу. Они обращались к сторонним компаниям, у которых были договорные обязательства по соглашениям об уровне обслуживания, где прописывались время безотказной работы, масштаб, оперативность реагирования и масса других важных архитектурных свойств. Теперь такие архитектуры, как микросервисы, свободно используются для решения подобных эксплуатационных задач. Например, гибкое масштабирование когда-то встраивалось в архитектуры с великим трудом (см. главу 15), а микросервисы справляются с этим гораздо легче благодаря взаимодействию архитекторов и DevOps[1].

[1] DevOps — это особый подход к организации команд разработки. Его суть в том, что разработчики, тестировщики и администраторы работают в едином потоке — не отвечают каждый за свой этап, а вместе работают над выходом продукта и стараются автоматизировать задачи своих отделов, чтобы код переходил между этапами без задержек. В DevOps ответственность за результат распределяется между всей командой. — *Примеч. ред.*

ИСТОРИЯ: PETS.COM И ПОЧЕМУ У НАС ЕСТЬ ГИБКОЕ МАСШТАБИРОВАНИЕ

История разработки программного обеспечения насыщена как плохими, так и хорошими уроками. Мы думаем, что современные возможности (например, гибкое масштабирование) появились в один прекрасный день благодаря талантливому разработчику, но подобные идеи зачастую рождались из тяжелых уроков. История компании Pets.com — один из ранних примеров усвоения таких вот уроков. Pets.com появилась на заре интернета в надежде повторить успех Amazon.com в сфере продажи товаров для животных. Им повезло с отделом маркетинга, где собрались превосходные специалисты, придумавшие привлекательный талисман: надеваемую на руку куклу с микрофоном, говорившую всякие дерзости. Эта кукла стала суперзвездой, появляясь на публике на парадах и национальных спортивных мероприятиях.

К сожалению, руководство Pets.com, по-видимому, потратило все деньги на эту куклу, а не на инфраструктуру. Когда посыпались заказы, компания оказалась не готова к этому. Веб-сайт работал медленно, транзакции терялись, доставка задерживалась... то есть дела шли по наихудшему сценарию. Все было настолько плохо, что компания закрылась вскоре после рождественской торговой лихорадки, продав конкуренту единственный сохранивший ценность актив (ту самую куклу).

Компании явно не хватало гибкого масштабирования: возможности востребованного наращивания ресурсов. Поставщики облачных сервисов предоставляют масштабирование в виде стандартного свойства, но на заре интернета компаниям приходилось самим управлять своей инфраструктурой, и многие из них пали жертвой неслыханного ранее феномена: слишком большой успех способен убить бизнес. Трагедия Pets.com и другие подобные ужастики заставили специалистов разработать фреймворки, которыми по сей день пользуются архитекторы.

В следующих разделах рассматривается ряд недавно возникших пересечений работы архитектора и других отделов организации, которые привели к возникновению новых возможностей и обязанностей.

Практика проектирования

Согласно сложившемуся порядку вещей, архитектура ПО была отделена от процесса разработки программного продукта. Известны десятки популярных методик создания программных продуктов, включая Waterfall и многие раз-

новидности Agile (например, Scrum, Extreme Programming, Lean и Crystal), которые, по большому счету, не оказывают на архитектуру ПО никакого влияния.

Но в последние несколько лет прогресс в области разработки программных средств заставил задуматься о процессах выстраивания архитектуры ПО. Появились причины отделить *процесс* разработки программного продукта от *практики проектирования*. Под *процессом* подразумевается формирование команд и управление ими, порядок проведения совещаний и организация повседневной работы, то есть все, что относится к специфике объединения людей и их взаимодействия. А под практикой *проектирования* ПО подразумевается деятельность, не связанная с процессом, но приносящая наглядную пользу. В качестве примера можно привести непрерывную интеграцию, которая не зависит от какого-либо конкретного процесса.

Сосредоточенность на практике проектирования играет весьма важную роль. Во-первых, разработка программных продуктов лишена многих свойств более зрелых инженерных дисциплин. Например, инженеры-строители могут предсказывать структурные изменения с гораздо большей точностью, чем аналогичные важные аспекты структуры ПО. Во-вторых, довольно важной проблемой разработки программных продуктов остается вычисление оценочных показателей: сколько уйдет времени, ресурсов, денег? Это связано с устаревшими методами оценки, которые не учитывают нужные показатели, а также с тем, что мы не можем оценить из-за *неизвестных неизвестностей*.

> ...Есть известные известные — вещи, о которых мы знаем, что знаем их. Есть также известные неизвестные — вещи, о которых мы знаем, что не знаем. Но еще есть неизвестные неизвестные — это вещи, о которых мы не знаем, что не знаем их.
>
> *Бывший министр обороны США Дональд Рамсфельд (Donald Rumsfeld)*

Неизвестные неизвестности — это злейший враг программных систем. Многие проекты начинаются с перечня *известных неизвестностей*: тех особенностей, которые появятся в будущем в этой сфере и технологии и по этой причине должны быть изучены разработчиками. Но также проекты могут стать жертвами *неизвестных неизвестностей*, внезапного появления которых никто не мог предугадать. Именно из-за этого попытки разработать программное обеспечение «с большим заделом» («Big Design Up Front») оказываются неудачными: архитекторы не в состоянии проектировать с учетом неизвестных неизвестностей. Приведем цитату Марка (одного из авторов этой книги):

> Разработка архитектуры стала итеративным процессом из-за *неизвестных неизвестностей*, но Agile просто признает это и делает это быстрее.

ПУТЬ ОТ ЭКСТРЕМАЛЬНОГО ПРОГРАММИРОВАНИЯ К НЕПРЕРЫВНОЙ ДОСТАВКЕ

История возникновения экстремального программирования (Extreme Programming, XP)[1] может послужить хорошей иллюстрацией отличия технологического *процесса от проектирования*. В начале 1990-х годов группа опытных разработчиков программного обеспечения под руководством Кента Бека (Kent Beck) начала проверять эффективность множества различных популярных в то время процессов разработки. Был сделан вывод, что ни один из них не добивается стабильно удачных результатов. Один из основателей XP как-то сказал, что при выборе одного из существующих процессов «успех проекта гарантировался не более, чем при подбрасывании монетки». Группа решила переосмыслить приемы создания программных продуктов и в марте 1996 года запустила проект XP. Они отказались от традиционных представлений и сконцентрировались на *практических подходах*, которые в прошлом приносили успех проектам, подняв их до «экстремального» уровня. Логика команды основывалась на наблюдаемой в предыдущих проектах взаимосвязи большего количества тестов и более высокого качества продукта. Таким образом, XP-подход к тестированию вывел проектирование на новый уровень: до проведения разработки, при которой тестирование выводится на первый план, гарантируя, что весь код будет протестирован до его попадания в кодовую базу.

XP-программирование было включено в список популярных Agile-процессов, но при этом относилось к тем немногочисленным методикам, которые включали в себя автоматизацию, тестирование, непрерывную интеграцию и другие четко выраженные и основанные на практическом опыте методы. Активное развитие разработки программного обеспечения началось с выходом обновленной версии многих практических приемов XP в книге «Continuous Delivery» (издательство Addison-Wesley Professional) и привело к появлению DevOps. Во многих отношениях DevOps-революция произошла в тот момент, когда был принят подход XP-программирования: автоматизация, тестирование, объявление единого источника достоверных данных и т. д.

Мы всецело поддерживаем достигнутый в этой области прогресс, выстраивающий поэтапные шаги, которые в конечном счете переведут разработку программного обеспечения в разряд полноценной проектной дисциплины.

[1] www.extremeprogramming.org

Получается, что при практической отделенности процесса от архитектуры итеративный процесс лучше приспособлен к природе архитектуры ПО. Команды, которые стремятся к созданию современной системы, например микросервисов, но пользуются при этом устаревшими процессами вроде каскадной разработки (Waterfall), столкнутся с большими трудностями из-за того, что устаревший процесс игнорирует новые правила сборки ПО.

Зачастую архитектор ПО также является техническим лидером разработчиков проекта и поэтому определяет стратегию проектирования, которой должна придерживаться команда. Это похоже на ситуацию, при которой архитектор строительства тщательно изучает проблемный георегион, прежде чем выбрать архитектурный стиль зданий. Он также следит за тем, чтобы архитектурный стиль и инженерные решения образовывали единую гармоничную систему. К примеру, архитектура микросервисов предполагает автоматизированное предоставление аппаратных ресурсов, автоматизированное тестирование и развертывание, а также целый ряд других возможностей. Попытка выстроить одну из таких архитектур с устаревшей структурой, ручными процессами и небольшим количеством тестов приводит к серьезным противоречиям и становится препятствием на пути к успеху. Методики проектирования должны быть подобраны в соответствии с правилами архитектуры, подобно тому как в проблемных регионах при строительстве зданий учитываются особенности окружающей среды.

Развитие мысли, позволившее пройти путь от экстремального программирования (XP) до непрерывной разработки (continuous delivery), продолжается. Последние достижения в практике проектирования открывают новые возможности в архитектуре. В книге Нила «Building Evolutionary Architectures»[1] (издательство O'Reilly)[2] раскрываются новые взгляды на пересечение методов проектирования и архитектуры, позволяющие улучшить автоматизацию управления архитектурой. Мы не будем приводить здесь краткое содержание этой книги, но отметим, что в ней представлен инновационный взгляд на концепцию свойств архитектуры, которые будут рассмотрены в оставшейся части нашей книги.

В книге Нила дается описание методов построения постепенно меняющихся архитектур. В главе 4 будет дано описание архитектуры в виде сочетания требований и дополнительных соображений, как показано на рис. 1.7.

---

[1] Форд Н., Парсонс Р., Куа П. «Эволюционная архитектура. Поддержка непрерывных изменений». СПб., издательство «Питер».

[2] shop.oreilly.com/product/0636920080237.do

Рис. 1.7. Архитектура программной системы содержит как требования, так и свойства архитектуры

Опираясь на весь накопленный в мире опыт разработки ПО, можно с уверенностью утверждать, что ничто не останется постоянным. Архитекторы могут спроектировать систему, отвечающую определенным критериям, но этот проект должен пережить как реализацию (у архитекторов должна быть возможность убедиться в правильности реализации их проекта), так и неизбежные изменения, обусловленные экосистемой разработки программного продукта. Нам нужна *эволюционирующая архитектура*.

В правилах *построения эволюционных архитектур* есть понятие *функций пригодности (fitness functions)*. Они служат для защиты архитектурных свойств (и управления ими), когда с течением времени происходят различные изменения. Идея позаимствована из эволюционных вычислительных методов. Во время создания генетического алгоритма разработчики могут применять множество методов изменения (мутации) решения, итеративно подбирая новые варианты. При построении такого алгоритма для конкретной цели необходимо оценить итоговый результат, чтобы увидеть, насколько он близок или далек от оптимального решения; такая оценка и является функцией пригодности. Например, если разработчики создали наследственный алгоритм для решения известной задачи коммивояжера (нахождение кратчайшего маршрута между различными городами), то функция пригодности оценит протяженность маршрута.

В *эволюционных архитектурах* эту методику используют и для создания *архитектурных функций пригодности*, дающих объективную оценку целостности ряда архитектурных свойств. Эта оценка может включать в себя различные механизмы: например, системы показателей, юнит-тесты, системы отслеживания и хаос-инжиниринг. Скажем, в качестве важного архитектурного свойства архитектор задает время загрузки страницы. Чтобы позволить системе вносить изменения в страницу без ущерба для производительности, архитектурой создается функция пригодности в виде теста, замеряющего время загрузки для каждой страницы, а затем этот тест запускается для проекта как часть непрерывной интеграции. Благодаря этому архитекторы всегда в курсе состояния критически важных частей архитектуры, поскольку у них имеется механизм проверки для каждой такой части в форме функции пригодности.

Мы здесь не будем вдаваться в подробности функций пригодности, но будем указывать на возможности и примеры такого подхода там, где это уместно. Обратите внимание на взаимосвязь между частотой выполнения функций пригодности и обратной связью, которую они обеспечивают. Вы поймете, что практика проектирования с использованием таких Agile-технологий, как непрерывная интеграция, автоматизированное предоставление аппаратных ресурсов и т. д., существенно упрощает построение гибких архитектур. К тому же при этом наглядно демонстрируется, насколько тесно архитектура переплетается с методами проектирования.

Использование систем сопровождения или DevOps

С появлением методологии DevOps произошло наиболее очевидное пересечение архитектуры и смежных областей, вызванное некоторым переосмыслением архитектурных аксиом. Долгое время множество компаний рассматривали сопровождение готовой программы как отдельную процедуру, не имеющую ничего общего с разработкой ПО, поэтому часто в целях экономии средств сопровождение передавалось другой компании. Многие архитектуры, разработанные в 1990-е и 2000-е годы, выстраивались с предположением, что архитекторы не в состоянии контролировать процесс эксплуатации, и были созданы в расчете на строгую защиту данного ограничения (здесь хорошим примером может послужить пространственная архитектура Space-Based Architecture, которую мы рассмотрим в главе 15).

Но несколько лет назад ряд компаний начали экспериментировать с новыми формами архитектуры, где задачи поддержки были встроены в архитектуру. Например, в архитектурах старого стиля, таких как ESB-driven SOA, сама архитектура выстраивалась для гибкого масштабирования, что существенно усложняло ее. Архитекторы были вынуждены обходить ограничения, связанные с экономией средств при передаче сопровождения на ауторсинг. Поэтому создавались архитектуры, способные самостоятельно справляться с масштабированием, поддержкой высокой производительности и гибкости, а также обеспечивать ряд других возможностей. Но побочным эффектом такого подхода была сильно усложненная архитектура.

Создатели архитектуры в стиле микросервисов поняли, что эксплуатационные проблемы лучше решать в процессе самой эксплуатации. Создавая связь между архитектурой и эксплуатацией, архитекторы могут упростить дизайн и возложить на процесс эксплуатации все то, с чем он хорошо справляется. Осознание того, что нерациональное использование ресурсов приводит к непреднамеренной сложности, объединило усилия архитекторов и эксплуатационников и привело к созданию микросервисов, подробно рассматриваемых в главе 17.

Процесс разработки

Еще одна аксиома гласит, что архитектура ПО по большей части не зависит от процесса разработки программ: способ создания программных продуктов (*процесс*) практически не влияет на архитектуру ПО (*структуру*). Это убеждение сохраняется даже в случае, если стиль проектирования, который выбрала команда, каким-то образом воздействует на архитектуру (особенно в части практики проектирования). Во многих книгах, посвященных архитектуре ПО, игнорируется сам процесс разработки, необоснованно считающийся предсказуемым. Однако процесс, с помощью которого команды разработчиков создают программное обеспечение, оказывает влияние на многие аспекты архитектуры. Например, за последние десятилетия множество компаний приняли методику разработки Agile из-за самой природы ПО. Архитекторы в Agile-проектах могут допустить поэтапную разработку и, следовательно, ускоренный цикл обратной связи для принятия решений. А это, в свою очередь, позволит архитекторам проявить настойчивость в проведении экспериментов и получении ценной информации, основанной на этой обратной связи.

Из предыдущей цитаты Марка следует, что рано или поздно к итеративной разработке придут все архитекторы. В связи с этим мы будем опираться на методологию Agile и указывать на исключения там, где это понадобится. Например, в отношении монолитных архитектур по-прежнему принято использовать устаревшие процессы разработки, что обусловливается их возрастом, политикой или иными факторами, не связанными с программным обеспечением.

Один из важнейших аспектов архитектуры, где наиболее ярко проявляются все преимущества Agile-методологии, — реструктуризация. У команд довольно часто возникают потребности в переносе своей архитектуры с одного паттерна на другой. К примеру, команда исходно полагалась на монолитную архитектуру, поскольку она проще и с нее быстрее начать, но по мере развития проекта им требуется переходить на более современную архитектуру. Такие изменения лучше поддерживаются не жестко распланированными процессами, а Agile-методами, которые обеспечивают сжатый цикл обратной связи и стимулируют применение таких технологий, как паттерн Душитель (Strangler Pattern) и переключатели функциональности.

Данные

Большая часть серьезных приложений не обходится без внешнего хранилища данных, зачастую в виде реляционной (или, что все чаще случается, NoSQL) базы

данных. Но многие книги по архитектуре ПО содержат весьма скудное описание этого важного аспекта архитектуры. У кода и данных имеется ярко выраженный симбиоз: один компонент абсолютно бесполезен без другого.

Администраторы баз данных часто работают вместе с архитекторами над созданием архитектуры данных для сложных систем. Задача — проанализировать, как схема базы данных и многократное использование влияют на портфель приложений. Мы не будем углубляться в такие специализированные детали в этой книге. Но в то же время не станем игнорировать существование внешних хранилищ и зависимость от них. В частности, когда речь пойдет об эксплуатационных аспектах архитектуры и об *архитектурных квантах* (см. главу 3), мы обязательно коснемся таких важных внешних факторов, как базы данных.

Законы архитектуры программного обеспечения

При всей необъятности сферы применения архитектуры ПО в ней существуют единые общие принципы. Прежде всего авторы усвоили *Первый закон архитектуры программного обеспечения*, постоянно сталкиваясь с ним:

> Все в архитектуре программного обеспечения соткано из компромиссов.
>
> *Первый закон архитектуры программного обеспечения*

Для архитекторов ПО не существует ничего заведомо удобного и понятного. Каждое решение исходит из учета множества противоречащих друг другу факторов.

> Если архитектор полагает, что обнаружил нечто бескомпромиссное, то, скорее всего, компромисс еще просто *не выявлен*.
>
> *Следствие 1*

Архитектура ПО определяется нами в понятиях, выходящих за рамки структурных конструкций, внедрения принципов, свойств и т. д. Архитектура — это нечто большее, чем простое сочетание структурных элементов, что находит отражение во *Втором законе архитектуры программного обеспечения:*

> *Почему* важнее, чем *как*.
>
> *Второй закон архитектуры программного обеспечения*

Важность именно такого взгляда на вещи обнаружилась авторами при попытке сохранить результаты упражнений, выполненных студентами во время практического курса по принятию архитектурных решений. Поскольку на упражнения отводилось ограниченное время, мы сохранили только схемы, показывающие топологию. Иными словами, мы зафиксировали то, *как* решалась задача, но не *почему* командой выбирался тот или иной вариант. Архитектор может посмотреть на существующую систему, о которой он ничего не знает, и понять, как работает эта структура, но ему будет трудно объяснить, почему были выбраны эти, а не другие варианты.

В книге везде акцентируется внимание на том, *почему* архитекторы принимают те или иные решения наряду с компромиссами. Кроме того, в разделе главы 19 «Запись архитектурных решений» (на с. 332) мы обращаем внимание на полезные приемы фиксирования важных решений.

ЧАСТЬ I

Основы

Чтобы разобраться в важных компромиссах архитектуры, разработчики должны усвоить ряд основных понятий и терминов, относящихся к компонентам, модульности, связанности и коннасценции.

ГЛАВА 2

Архитектурное мышление

Архитектор смотрит на вещи совсем не так, как разработчик, аналогично тому как метеоролог смотрит на облака не так, как художник. Взгляд архитектора обусловлен *архитектурным мышлением*. К сожалению, слишком многие архитекторы считают, что архитектурное мышление — это просто «размышления об архитектуре» (рис. 2.1).

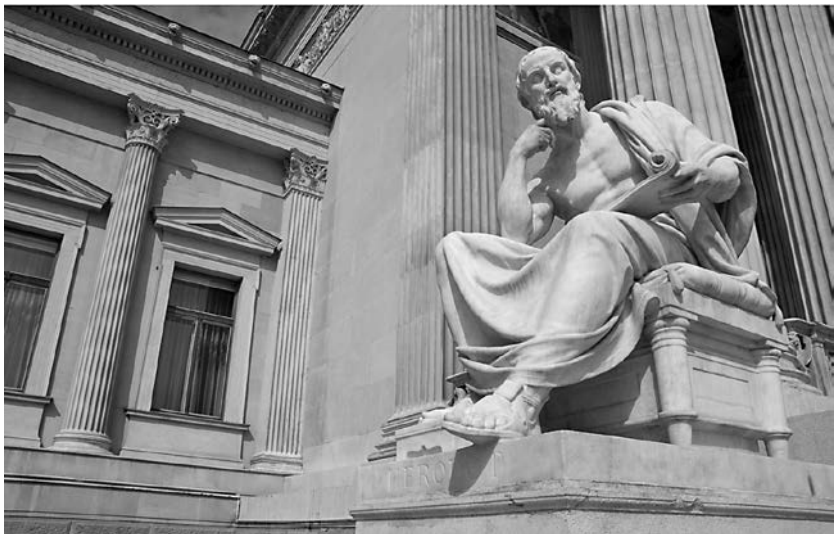


Рис. 2.1. Архитектурное мышление (iStockPhoto)

Но понятие архитектурного мышления гораздо шире. Это архитектурный взгляд на вещи или, говоря иначе, архитектурная точка зрения. Архитектурное мышление характеризуется четырьмя основными особенностями. Во-первых, это понимание разницы между архитектурой и проектированием и умение

сотрудничать с командами разработчиков, добиваясь работоспособности архитектуры. Во-вторых, это широкий спектр технических знаний при сохранении достаточного уровня (глубины) технической компетенции, что позволяет архитектору видеть решения и возможности, которые могут не заметить остальные участники проекта. В-третьих, это осознание, анализ и выдерживание баланса компромиссов при принятии различных решений и выборе технологий. И наконец, в-четвертых, это понимание важности бизнес-факторов и того, как они влияют на решение архитектурных задач.

В этой главе будут исследованы все четыре особенности архитектурного мышления и взгляд на вещи глазами архитектора.

Архитектура и проектирование

Поиск границы между архитектурой и проектированием часто приводит к разногласиям. Где заканчивается выстраивание архитектуры и начинается проектирование? Каковы обязанности архитектора и что возлагается на разработчика? Архитектурное мышление предполагает осознание разницы между выстраиванием архитектуры и проектированием и понимание, насколько тесно они взаимодействуют как при решении задач бизнеса, так и при принятии технических решений.

Рассмотрим изображенное на рис. 2.2 традиционное сравнение обязанностей архитектора с обязанностями разработчика. Согласно схеме, архитектор отвечает за анализ бизнес-требований, чтобы выявить и определить архитектурные свойства (выражаемые словами с окончанием на *-ость*); за выбор конкретных архитектурных паттернов и стилей, соответствующих задачам предметной области; и, наконец, за создание компонентов (строительных блоков системы). Затем все эти решения передаются команде разработчиков, отвечающей за создание диаграммы классов для каждого компонента, экранов пользовательского интерфейса, а также за разработку и тестирование исходного кода.

В традиционной модели ответственности, показанной на рис. 2.2, имеется целый ряд проблем. По сути, именно здесь и показано, почему архитектура редко работает так, как следует. Главной причиной всех проблем, связанных с архитектурой, является однонаправленная стрелка, что проходит через виртуальные и физические барьеры, отделяющие архитектора от разработчика. Решения, принимаемые архитектором, порой вообще не доходят до команд разработчиков, а решения команд разработчиков, изменяющие архитектуру, редко попадают в поле зрения архитектора. В этой модели архитектор не связан с командами разработчиков, поэтому архитектура редко достигает цели, ради которой она была создана.

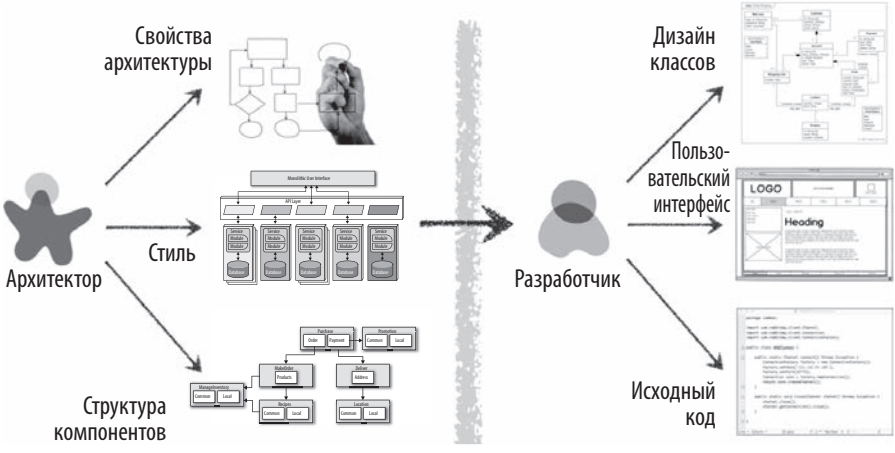


Рис. 2.2. Традиционное сравнение выстраивания архитектуры и проектирования

Чтобы получить работоспособную архитектуру, следует разрушить все существующие физические и виртуальные барьеры между архитекторами и разработчиками, сформировав, таким образом, двунаправленные отношения. Рабочие отношения предполагают объединение архитектора и разработчика в одну виртуальную команду (рис. 2.3). Такая модель не только поможет созданию прочной двунаправленной связи между выстраиванием архитектуры и проектированием, но и позволит архитектору направлять и инструктировать команду разработчиков.

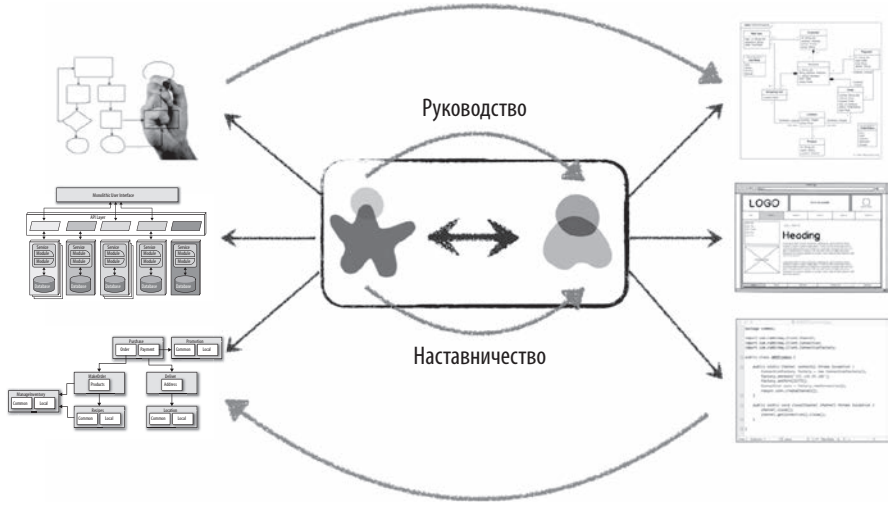


Рис. 2.3. Обеспечение работоспособности архитектуры за счет сотрудничества

В отличие от каскадных подходов старой школы, предполагающих статичную и неизменную архитектуру ПО, архитектура современных систем изменяется и совершенствуется с каждым этапом или фазой проекта. Тесное сотрудничество архитектора и команды разработчиков является залогом успеха любого программного проекта. А где же тогда заканчивается выстраивание архитектуры и начинается проектирование? Да нигде. И то и другое является частью жизненного цикла программного проекта, и для достижения успеха оба процесса должны всегда выполняться синхронно.

Широта технических взглядов

У разработчиков и архитекторов разные сферы охвата технологических деталей. В отличие от разработчика, который для выполнения своих задач обязан иметь весьма существенный объем глубоких технических знаний (*техническую глубину*), архитектор ПО должен обладать широтой технических взглядов (*технической широтой*, или *техническим кругозором*), чтобы мыслить как архитектор и смотреть на вещи с точки зрения архитектуры. Эти понятия иллюстрируются пирамидой, показанной на рис. 2.4, в которой заключены все технические знания в мире. Оказывается, вид информации, представляющей ценность для технического специалиста, меняется по мере его профессионального роста.

Рис. 2.4. Пирамида, представляющая весь объем знаний

На рис. 2.4 показано, что любой человек может разделить все свои знания на три части: *на известные ему вещи*, на *неизвестные* и на *те вещи, о которых он не знает, что не знает их*.

К *известным* знаниям можно отнести технологии, рабочие среды, языки и инструменты, повседневно используемые техническим специалистом для выполнения своей работы: например, Java-программист знает язык Java. К *неизвестным* знаниям относится то, о чем технический специалист немного осведомлен или же наслышан, но не имеет опыта работы в данной области. Подходящим примером этого уровня знания может послужить язык программирования Clojure. Большинство технических специалистов *слышали* о Clojure и знают, что это язык программирования, созданный на основе Lisp, но они не умеют на нем программировать. *Вещи, о которых мы не знаем, что не знаем их,* занимают самую большую часть пирамиды знаний и включают в себя всю совокупность тех технологий, инструментов, рабочих сред и языков, что могли бы идеально подойти для решения поставленной задачи, но о существовании которых технический специалист даже не догадывается.

В начале своей карьеры разработчики сосредоточены на расширении вершины пирамиды, набираются опыта и повышают квалификацию. На ранней стадии это считается идеальным подходом, поскольку разработчикам нужно расширять перспективы, нарабатывать навыки и приобретать практический опыт. Расширение верхней части не обходится и без расширения средней: разработчикам все чаще попадаются технологии и связанные с ними готовые продукты, которыми они пополняют свой арсенал *неусвоенных знаний*.

Расширение верхней части пирамиды выгодно, потому что опыт оценивается высоко (рис. 2.5). Но *усвоенные знания* также относятся к *знаниям, требующим постоянной поддержки их актуальности*, поскольку в мире программного обеспечения нет ничего статичного. Если разработчик становится знатоком Ruby on Rails, его квалификация будет утрачена, стоит только ему забросить Ruby on Rails на год-два. Нужно уделять достаточно времени, чтобы поддерживать актуальность знаний, показанных в верхней части пирамиды. По сути, размер верхней части чьей-то конкретной пирамиды отображает *техническую глубину* знаний данного специалиста.

Но как только разработчик становится архитектором, природа его знаний меняется. Основную ценность для архитектора представляет *широкий* технический кругозор и понимание, как технологии могут использоваться для решения конкретных задач. Например, архитектору куда полезнее знать, что для решения конкретной задачи существует пять вариантов, чем обладать высокой квалификацией только в одном из них. Для архитектора наиболее важными являются две части пирамиды: ее верхняя часть *и* середина; то, насколько сильно средняя часть вторгается в нижнюю, дает представление о *широте* его технического кругозора (рис. 2.6).

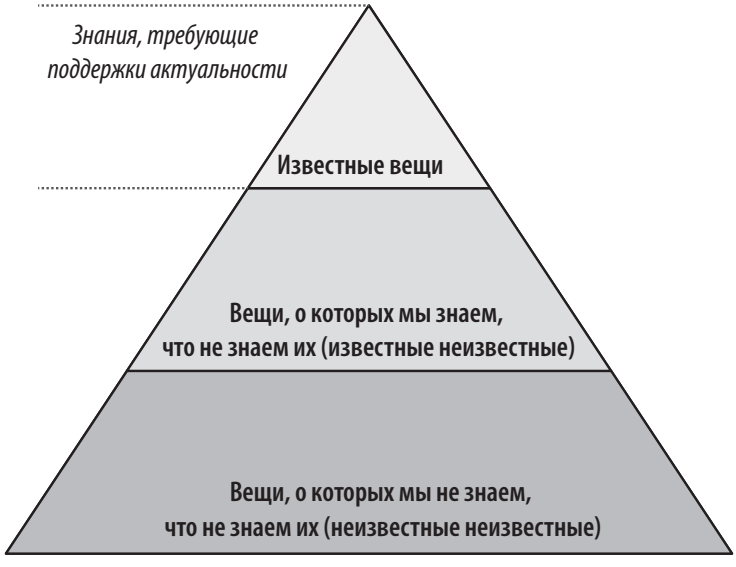


Рис. 2.5. Для сохранения квалификации разработчики должны поддерживать актуальность своих знаний

Рис. 2.6. Объем усвоенных знаний представляет техническую глубину, а эрудиция — техническую широту

Для архитектора *широта* важнее *глубины*. Поскольку архитекторам приходится принимать решения с учетом технических ограничений, для них наиболее ценен кругозор, позволяющий видеть широкий спектр возможных решений. Получается, что архитектору разумнее будет пожертвовать некоторым опытом, полученным с таким трудом, и потратить время не на его поддержание, а на расширение своего кругозора (рис. 2.7). На диаграмме показано, что часть квалификации (возможно, в наиболее привлекательных областях) сохранится, а часть утратится с пользой для дела.

Рис. 2.7. Рост диапазона и уменьшение глубины знаний для соответствия роли архитектора

Наша пирамида знаний показывает принципиальные отличия специальности *архитектора* от специальности *разработчика*. Разработчики на протяжении всей своей профессиональной деятельности оттачивают мастерство. Переход к роли архитектора подразумевает пересмотр этой парадигмы, что у многих вызывает затруднения. А это, в свою очередь, приводит к двум весьма распространенным вариантам неразумных действий: во-первых, архитектор пытается сохранять квалификацию в самых различных областях, не преуспев ни в одной из них, и изматывает себя в процессе работы. Во-вторых, это проявляется в виде внедрения *устаревшего опыта*: у архитектора возникает ошибочное ощущение, что устаревшая информация все еще актуальна. Такая картина зачастую наблюдается в крупных компаниях, где разработчики, стоявшие у ее истоков, становятся руко-

водителями и по-прежнему принимают технологические решения на основе устаревших критериев (см. ниже врезку «Антипаттерн Замороженный троглодит»).

Архитекторам следует сосредоточиться на технической широте, чтобы у них был большой колчан, из которого можно будет извлекать стрелы. Разработчикам, выбившимся в архитекторы, возможно, придется изменить свое представление о получении знаний. На протяжении всей своей профессиональной деятельности каждый разработчик должен выдерживать тонко выверенный баланс глубины и широты своих знаний.

АНТИПАТТЕРН ЗАМОРОЖЕННЫЙ ТРОГЛОДИТ

Поведенческий *антипаттерн Замороженный троглодит (Frozen Caveman)*, часто встречающийся на практике, описывает архитектора, который в каждой архитектуре неизменно возвращается к своим излюбленным, но уже ничем не обоснованным принципам. Например, один из коллег Нила разрабатывал систему с централизованной архитектурой. И неизменно при каждом представлении проекта архитекторы заказчика задавали вопрос: «А мы не потеряем Италию?». Несколько лет назад при весьма странных обстоятельствах пропала связь штаб-квартиры со своими магазинами в Италии, что вызвало ряд серьезных неудобств. Шансы на повторение инцидента были крайне малы, но архитекторы зациклились именно на этом свойстве архитектуры.

Обычно этим грешат архитекторы, которые в прошлом обожглись на неверных решениях или неожиданных происшествиях, заставивших их впоследствии проявлять излишнюю осторожность. Оценка важности рисков необходима, но она должна быть объективной. Умение определять разницу между реальным и предполагаемым техническим риском является частью непрерывного процесса обучения архитектора. Архитектурное мышление требует преодоления негативных установок и опыта «замороженного троглодита», а также поиска других решений и постановки более актуальных вопросов.

Анализ компромиссов

Мыслить как архитектор — это значит видеть компромиссы в каждом решении, как технического, так и любого другого плана, и анализировать эти компромиссы, чтобы выбрать лучшее. Процитируем Марка (одного из авторов книги):

> Архитектура — это то, что невозможно загуглить.

В архитектуре *все* соткано из компромиссов, поэтому известный ответ на каждый касающийся архитектуры вопрос будет звучать так: «Это с какой стороны посмотреть». Многих такой ответ раздражает, но, к сожалению, так оно и есть. В поисковике Google не найти ответа на вопросы: что будет лучше, REST или обмен сообщениями, или будут ли микросервисы верным архитектурным стилем. Все зависит от конкретных обстоятельств. От среды развертывания, бизнес-факторов, сложившейся в компании технической культуры, выделенного бюджета, установленных сроков, мастерства разработчиков и десятка других обстоятельств. У всех разные окружения, ситуации и задачи, именно поэтому выстраивание архитектуры — дело крайне трудное. Процитируем Нила (еще одного автора книги):

> В архитектуре нет верных или неверных ответов, есть только компромиссы.

Рассмотрим, к примеру, систему аукционных торгов, показанную на рис. 2.8, где все предлагают свою цену за лот, выставленный на аукцион.

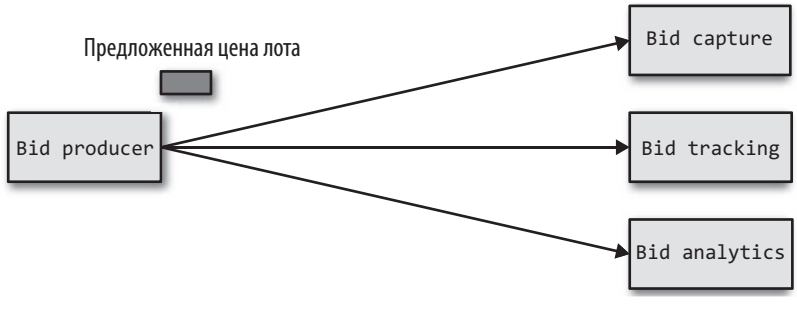


Рис. 2.8. Пример компромисса в аукционной системе: выбрать очереди или темы?

Сервис `Bid Producer` генерирует предложение цены от участника аукциона, после чего отправляет ее значение сервисам `Bid Capture`, `Bid Tracking` и `Bid Analytics`. Это можно сделать с помощью очередей в стиле обмена сообщениями «точка — точка» (point-to-point) или же воспользоваться темой (топиком, topic) в формате «публикация и подписка» (publish-and-subscribe). Каким вариантом воспользоваться архитектору? В Google ответа нет. Архитектурное мышление требует проанализировать компромиссы, связанные с каждым вариантом, и выбрать самый подходящий для конкретной ситуации.

На рис. 2.9 и 2.10 показаны два варианта обмена сообщениями для системы аукционных торгов. На первом рисунке — использование топика в модели «пуб-

ликация и подписка», а на втором — использование очередей в модели обмена сообщений «точка — точка».

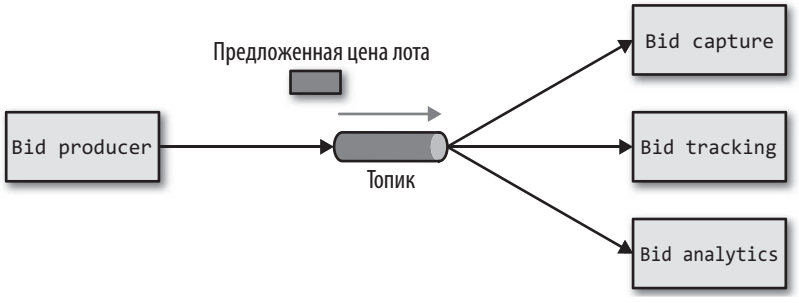


Рис. 2.9. Использование топика для обмена сообщениями между сервисами

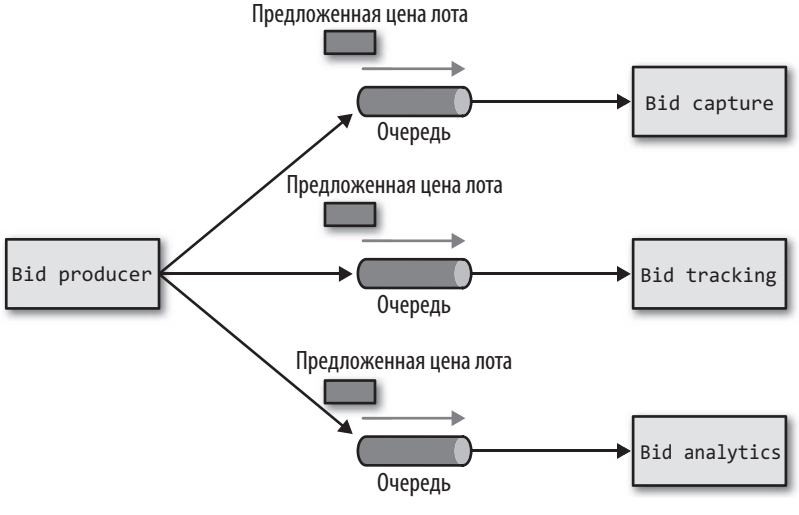


Рис. 2.10. Использование очереди для обмена сообщениями между сервисами

Явное преимущество варианта на рис. 2.9 (и, казалось бы, очевидное решение данной задачи) заключается в *возможности расширения архитектуры*. Сервису `Bid Producer` требуется только одно подключение к топику, в отличие от решения на рис. 2.10, где `Bid Producer` нуждается в подключении к трем различным очередям. Если в эту систему будет добавлена новая служба под названием `Bid History` (история заявок) в связи с требованием предоставить каждому участнику торгов историю всех ставок, сделанных им на каждом аукционе, то

в существующую систему не придется вносить никаких изменений. Когда новый сервис `Bid History` будет создан, он сможет просто подписаться на топик, уже содержащий информацию о предложенной цене. А вот в варианте использования очередей, показанном на рис. 2.10, для сервиса `Bid History` понадобится новая очередь, и придется внести изменения в сервис `Bid Producer` с добавлением еще одного подключения к новой очереди. Дело в том, что использование очередей требует значительных изменений в системе при добавлении новой функциональности, а в случае использования топика существующая инфраструктура остается неизменной. Также следует отметить, что `Bid Producer` в варианте с топиками получается менее связанным: ему неизвестно, как будет использоваться информация о предложенной цене или какому сервису она понадобится. А в варианте с использованием очередей `Bid Producer` получает точные сведения о порядке применения информации о торгах (и кем она будет использоваться), следовательно, он более связан с системой.

При таком анализе кажется очевидным, что подход с использованием топика и модели обмена сообщениями «публикация и подписка» является наилучшим выбором. Но приведем цитату Рича Хикки (Rich Hickey), создателя языка программирования Clojure:

> Программисты знают о преимуществах чего угодно, но не разбираются ни в каких компромиссах. Архитекторам нужно понимать и то и другое.

Архитектурное мышление предполагает изучение преимуществ конкретного решения, а также анализ его негативных сторон или связанных с ним компромиссов. В примере с аукционными торгами архитектор ПО проанализирует негативные стороны решения с использованием топика. Обратите внимание, что на рис. 2.9 *любой* может получить доступ к информации торгов, что создает проблемы с безопасностью. В модели, применяющей очередь и показанной на рис. 2.10, доступ к данным в очереди может получить *только* конкретный потребитель, которому адресуется это сообщение. Если вредоносный сервис перехватит очередь, предложенные цены не будут получены ожидающим их сервисом, и тут же появится уведомление о потере данных (а следовательно, и о возможной бреши в системе безопасности). Иными словами, проще перехватить топик, а не очередь.

В дополнение к проблеме безопасности решение с топиком, показанное на рис. 2.9, поддерживает только однотипные заявки. Все сервисы, получающие данные о предложенных ценах, должны следовать условиям одной и той же заявки и оперировать набором одних и тех же данных об этих ценах. В варианте с очередями, показанном на рис. 2.10, каждый потребитель может иметь свой собственный запрос с нужными ему данными. Предположим, к примеру, что

новому сервису `Bid History` вместе с предложенной ценой требуется текущая запрошенная цена, но эта информация не нужна другим сервисам. В таком случае необходимо изменить заявку, что повлияет на все остальные сервисы, использующие эти данные. В модели с использованием очередей это будет отдельно взятый канал, следовательно, и отдельный запрос, не оказывающий влияния ни на какой другой сервис.

Еще одним недостатком модели с использованием топика, показанной на рис. 2.9, является отсутствие отслеживания количества сообщений в топике, а следовательно, и возможности автоматического масштабирования. При использовании варианта с очередями на рис. 2.10 каждая очередь отслеживается в индивидуальном порядке, а программное распределение нагрузки применяется к каждому потребителю предложенной цены, то есть каждый из них может автоматически масштабироваться независимо друг от друга. Обратите внимание, что этот компромисс зависит от стратегии, так как расширенный протокол организации очереди сообщений (Advanced Message Queuing Protocol, AMQP)[1] способен поддерживать программное распределение нагрузки и мониторинг благодаря разделению между обменом (то, что отправляет производитель) и очередью (то, что слушает потребитель).

И каким же, исходя из проведенного анализа, будет самый подходящий вариант? Как ответить? Это зависит от обстоятельств! Все компромиссы сведены в табл. 2.1.

Таблица 2.1. Компромиссы между топиками и очередями

Преимущества топика	Недостатки топика
Архитектурная расширяемость	Проблемы с доступом к данным и их безопасностью
Отсутствие связанности сервисов	Нет разнородных контрактов
	Нет отслеживания и программной масштабируемости

Суть здесь в том, что *все* в архитектуре ПО выстраивается на основе компромиссов: сочетания преимуществ и недостатков. Мыслить как архитектор — это значит анализировать эти компромиссы, а затем задавать вопрос: «Что важнее: расширяемость или безопасность?». Решение о конкретном выборе из множества различных вариантов всегда будет зависеть от бизнес-факторов, рабочей среды и множества других обстоятельств.

[1] www.amqp.org

Понимание бизнес-факторов

Архитектурное мышление предполагает понимание бизнес-факторов, необходимых для успешной работы системы, и преобразование требований в свойства архитектуры (например, масштабируемость, производительность и доступность). Это сложная задача, которая требует от архитектора определенного уровня знаний в предметной области и налаженных отношений с ключевыми стейкхолдерами. Этой теме посвящено сразу несколько глав данной книги. В главе 4 мы даем определение различным свойствам архитектуры. В главе 5 описываем способы выявления и классификации свойств архитектуры. А в главе 6 объясняем, как измерить каждое из этих свойств, чтобы убедиться, что они соответствуют требованиям бизнеса.

Баланс между архитектурой и кодированием

Одной из наиболее сложных задач, стоящих перед архитектором, является поиск баланса между разработкой ПО (собственно кодированием) и архитектурой. Мы абсолютно уверены, что каждый архитектор должен обладать навыками программирования и поддерживать достаточный уровень своих технических знаний (см. предыдущий раздел «Широта технических взглядов» на с. 51). Это не кажется сложным, но иногда может быть весьма затруднительным.

Первый совет для поддержания баланса между кодированием и деятельностью архитектора — избегать ловушек узких мест. Такая уязвимость возникает, когда архитектор сам пишет код на важном этапе проектирования (обычно это код фреймворка) и начинает тормозить работу команды. Дело в том, что архитектор не является штатным разработчиком и вынужден балансировать между ролью разработчика (создание и тестирование исходного кода) и ролью архитектора (рисование диаграмм, участие в совещаниях, а также посещение большого количества мероприятий).

Один из способов избежать ловушки узкого места и остаться успешным архитектором — делегировать кодирование критического пути и фреймворка команде разработчиков, а самому сосредоточиться на программировании бизнес-функций (сервиса или системы вывода на экран). Такой подход имеет три позитивных момента. Во-первых, архитектор набирается практического опыта написания рабочего кода и не тормозит работу команды. Во-вторых, критический путь кода и фреймворк поручается команде разработчиков (как это и должно быть), и им передается владение кодом и более глубокое понимание наиболее сложных частей системы. В-третьих, что, наверное, важнее всего, архитектор пишет тот же исходный код, относящийся к бизнес-сфере, что и команда разработчиков, и поэтому может лучше понять все сложности, с которыми они могут столкнуться.

Но давайте предположим, что архитектор не создает код вместе с командой. Как он сможет сохранить квалификацию и поддерживать нужный уровень своих технических знаний? Существует четыре основных способа не терять практических навыков, не программируя при этом в нерабочее время (хотя мы рекомендуем не забрасывать программирование и в домашней обстановке).

Первый из них заключается в частой проверке концепций (proof-of-concepts, POCs). Эта практика не только требует от архитектора знания исходного кода, но и помогает подтвердить принятое архитектурное решение за счет дополнительного внимания к деталям реализации. Например, если архитектор затрудняется в выборе одного из двух решений по кэшированию данных, то создание рабочего примера по каждому средству кэширования и сравнение результатов поможет сделать правильный выбор. При этом архитектор получит представление о деталях реализации и объеме работы по созданию полноценного решения. Кроме того, он сможет провести более качественное сравнение таких архитектурных свойств, как масштабируемость, производительность или общая отказоустойчивость различных решений по кэшированию данных.

Наш совет: при проверке концепций архитектор должен писать код, максимально приближенный к бизнес-требованиям. Мы рекомендуем такую практику по двум причинам. Во-первых, довольно часто отработанный код PoC попадает в репозиторий исходного кода и становится образцом или примером, которому следуют другие разработчики. Вряд ли вам хотелось бы, чтобы черновой, небрежный код стали считать примером вашей работы. Во-вторых, создание PoC-кода с качеством на уровне конечного продукта повышает квалификацию архитектора в разработке добротного, хорошо структурированного кода и позволяет не растерять ее.

Второй способ, помогающий архитектору сохранять навыки программирования, заключается в том, чтобы браться за устранение технических недоработок (технического долга) или архитектурных шероховатостей исходного кода, освобождая команду для работы над критически важными пользовательскими функциями. Ликвидация шероховатостей обычно относится к низкоприоритетным задачам, поэтому если у архитектора на текущем этапе не получается избавиться от технического долга или от мелких огрехов в архитектуре, то конец света не наступит, и это, как правило, не повлияет на успешное завершение этапа.

Аналогичный способ, позволяющий архитектору сохранять квалификацию программиста и оказывать при этом помощь команде разработчиков, — исправление багов в ходе текущего этапа разработки. Конечно, это не слишком привлекательно, но данный метод позволяет выявлять возможные проблемы и слабые стороны кодовой базы, а может быть, и самой архитектуры.

Следующий отличный способ — создание простых инструментов, запускаемых из командной строки, и анализаторов, помогающих команде разработчиков в решении их повседневных задач и повышающих ее эффективность. Нужно найти повторяющиеся задачи, выполняемые командой, и автоматизировать процесс. Команда будет вам благодарна! В качестве примеров можно привести отсутствующие в других линтерах автоматизированные средства проверки исходного кода на соответствие конкретным стандартам программирования, а также автоматизацию повторяющихся ручных приемов рефакторинга кода.

Автоматизация может также выполняться с помощью средств архитектурного анализа и функций пригодности, подтверждающих жизнеспособность и соответствие архитектуры предъявляемым требованиям (compliance). Например, архитектор может написать программу в ArchUnit[1] на языке Java и на Java-платформе, чтобы автоматизировать проверку соответствия требованиям, или же создать для этого специализированные функции пригодности[2], а заодно и набраться практического опыта программирования. Методика такой работы будет рассмотрена в главе 6.

Наконец, способ, позволяющий архитектору сохранять высокую квалификацию программиста, — частые код-ревью. Хотя при этом архитектор фактически не занимается программированием, но все же он не остается в стороне от создания программного кода. Кроме того, код-ревью дает дополнительные преимущества: проверка соответствия кода заданной архитектуре, а также поиск возможностей для менторства и коучинга команды.

[1] www.archunit.org

[2] evolutionaryarchitecture.com

ГЛАВА 3

Модульность

Давайте разберемся с некоторыми ключевыми терминами, которыми наполнены и даже переполнены рассуждения об архитектуре, связанные с модульным принципом построения (модульностью), и дадим определения, которые будем далее использовать в книге.

> Девяносто пять процентов слов [об архитектуре программного обеспечения] посвящено преимуществам «модульности», но при этом почти ничего говорится о том, как ее можно достичь.
>
> *Гленфорд Дж. Майерс (Glenford J. Myers), 1978 год*

Различные платформы предлагают разные механизмы повторного использования кода, но все они поддерживают тот или иной способ объединения связанного кода в *модули*. Хотя сама концепция универсальна для архитектуры ПО, она все же трудно поддается определению. Простой поиск в интернете выдает десятки несогласованных (а иногда и противоречивых) определений. Как можно увидеть из приведенной выше цитаты Майерса, это давняя проблема. Но поскольку общепризнанного определения не существует, мы должны ввязаться в спор и представить наши собственные решения — ради последовательности и согласованности нашей книги.

Для архитекторов ПО очень важно понимать, что такое модульность и как она реализуется на выбранной платформе разработки. Многие инструменты, которые мы используем для анализа архитектуры (метрики, функции пригодности, средства визуализации), зависят от принятых концепций модульности. Модульность — это принцип организации работы. Если архитектор ПО занимается проектированием, не обращая внимания на взаимосвязанность элементов, то в конечном итоге он получает систему с массой проблем. По аналогии с физикой, программное обеспечение моделирует сложные системы, склонные к энтропии (беспорядку). Для сохранения порядка к физической си-

стеме должна быть приложена определенная энергия. То же самое применимо и к программным системам: архитекторы должны постоянно тратить энергию на обеспечение прочности (основательности) структуры, которая сама по себе ниоткуда не возьмется.

Сохранение высокой модульности — это один из примеров для нашего определения *неявных* свойств архитектуры: практически ни один из проектов не содержит требований к архитектору по обеспечению модульности и обмена данными, но чтобы кодовая база была устойчивой, она требует порядка и согласованности.

Определение

В словаре понятию *модуль* дается такое определение: «Каждый элемент набора стандартизированных частей или независимых элементов, который может использоваться для составления более сложной структуры». Мы используем понятие *модульность* для описания логической компоновки взаимосвязанного кода, в качестве которой может выступать группа классов в объектно-ориентированном языке либо функций в структурном или функциональном языке. В большинстве языков есть механизмы модульных построений (пакеты `package` в Java, пространства имен `namespace` в .NET-среде и т. д.). Обычно разработчики используют модули, чтобы сгруппировать взаимосвязанный код. Например, пакет `com.mycompany.customer` в Java должен содержать то, что относится к клиентам.

В настоящее время существует большое разнообразие механизмов компоновки, поэтому разработчику порой бывает трудно выбрать наиболее подходящий. Например, во многих современных языках программирования есть возможность задавать поведение в функциях/методах, в классах или же в пакетах/пространствах имен, и все они имеют свою область видимости и правила масштабирования. Другие языки усложняют ситуацию добавлением таких программных конструкций, как протокол метаобъектов[1], чтобы предоставить разработчикам еще больше механизмов расширения.

Архитекторы должны знать, как разработчики формируют пакеты, поскольку это оказывает существенное влияние на архитектуру. Например, если несколько пакетов тесно связаны, то повторное использование одного из них для другой работы оказывается более сложным.

[1] oreil.ly/9Zw-J

МОДУЛЬНОСТЬ ДО ПОЯВЛЕНИЯ КЛАССОВ

Программисты, работавшие еще до появления объектно-ориентированных языков, могут недоумевать, почему существует так много различных схем разделения. Часто это связано не с совместимостью кода, а с представлением разработчиков о том, какая должна быть совместимость. В марте 1968 года Эдсгер Дейкстра (Edsger Dijkstra) опубликовал статью в журнале «Communications of the ACM», озаглавленную «Go To Statement Considered Harmful» («О вреде оператора Go To»). Он осуждал широкое применение оператора `GOTO`, который позволял совершать нелинейные переходы по всему программному коду, затрудняя его отладку и отслеживание логики программы.

Эта статья помогла переходу в эру *структурных* языков программирования, примерами которых могут послужить Pascal и C, побуждающие задумываться о взаимосвязи элементов. Вскоре разработчики обнаружили отсутствие подходящего способа логической компоновки взаимосвязанного кода. Так наступила недолгая эра *модульных* языков, таких как Modula (еще один язык, разработанный создателем Pascal Никлаусом Виртом (Niklaus Wirth)) и Ada. В них имелась программная конструкция *модуля*, отвечающая современным представлениям о пакетах или пространствах имен (но без классов).

Эра модульного программирования была недолгой. Популярность завоевали объектно-ориентированные языки, поскольку они предлагали новые способы инкапсуляции и повторного использования кода. Тем не менее разработчики осознали эффективность модулей и теперь используют их в виде пакетов, пространств имен и т. д. Существует множество странных функций совместимости для поддержки этих различных систем. Например, Java поддерживает модульную (с помощью пакетов и пакетной инициализации с использованием статических инициализаторов), объектно-ориентированную и функциональную парадигму; каждый из этих стилей программирования имеет свои правила и особенности.

При обсуждении архитектуры мы используем понятие «модульность» как общий термин для обозначения компоновки взаимосвязанного кода: классов, функций или любого другого группирования. Здесь подразумевается не физическое, а логическое разделение, и очень важно понимать, чем они отличаются. Например, объединение большого количества классов в монолитное приложение может быть целесообразным с точки зрения удобства. Но когда придет время перестраивать архитектуру, связанность, основанная на произвольном разбие-

нии на разделы, становится препятствием для разделения монолита на части. Таким образом, о модульности лучше говорить как о понятии, не имеющем ничего общего с физическим разделением, навязанным или же подразумеваемым конкретной платформой.

ЯЗЫК БЕЗ КОНФЛИКТОВ ИМЕН: JAVA 1.0

Первые Java-разработчики обладали солидным опытом работы с конфликтами имен и коллизиями на различных платформах программирования. В первой версии Java использовался хитрый прием, позволявший обходить неоднозначность при наличии двух классов с одним и тем же именем. Например, что делать, если ваша предметная область включает заказ по каталогу (catalog *order*) и порядок установки (installation *order*)? И там и там фигурирует имя *order*, но с совершенно разным назначением (и классами). Решение, принятое в Java, заключалось в создании механизма пространства имен `package` наряду с требованием точного соответствия физической структуры каталогов и имен пакетов. Поскольку файловая система не разрешает иметь одинаковые имена в одном и том же каталоге, разработчики для исключения случаев неоднозначности воспользовались внутренними возможностями операционной системы. Получилось, что в Java исходный путь к классам (`classpath`) содержит только каталоги, что позволяет избежать возникновения конфликтов имен.

Через некоторое время разработчики языка поняли, что вынужденное присутствие в каждом проекте целиком сформированной структуры каталогов неудобно, особенно когда проект разрастался. Кроме того, усложнялось создание повторно используемых ресурсов: фреймворки и библиотеки должны были «вклиниваться» в структуру каталогов. Во втором основном релизе Java (1.2, который был назван Java 2) разработчики добавили механизм `jar`, позволяющий архивным файлам выполнять в classpath роль структуры каталогов. В течение следующего десятилетия разработчики Java пытались получить `classpath` точно таким же образом, в виде сочетания каталогов и JAR-файлов. И конечно же, из этой затеи так ничего и не вышло: теперь конфликтующие имена в classpath могли получаться из двух JAR-файлов, что приводило к многочисленным битвам по отладке загрузчиков классов.

Здесь стоит отметить общую концепцию *пространства имен*, которая отделена от технической реализации .NET-платформы. Разработчикам зачастую нужны конкретные, четко обозначенные названия функциональных средств програм-

мы, чтобы различать различные программные средства (компоненты, классы и т. д.) между собой. Наиболее наглядным примером повседневного пользования является интернет, в котором применяются уникальные, глобальные идентификаторы, привязанные к IP-адресам. В большинстве языков имеется какой-либо механизм модульности, который служит также пространством имен для организации компонентов кода: переменных, функций и/или методов. Иногда модульная структура имеет отражение и в физическом плане. Например, в Java требуется, чтобы имеющаяся в этом языке пакетная структура являлась отражением структуры каталогов физических файлов классов.

Измерение модульности

Учитывая важность модульности для архитекторов, необходимы инструменты для ее измерения. К счастью, исследователи создали разнообразные метрики, не зависящие от конкретного языка, которые помогают архитекторам оценить степень модульности. Мы сосредоточимся на трех основных концепциях: *связности,* или *сцепления (cohesion)*, *связанности (coupling)* и *коннасценции (connascence)*.

Связность

Связность (cohesion) говорит о том, содержатся ли все взаимосвязанные элементы внутри одного модуля. Другими словами, это степень взаимосвязанности элементов модуля между собой. В идеальном модуле, обладающем абсолютной связностью, все элементы собраны в один пакет, поскольку разбиение на более мелкие составляющие потребует связанности, или зацепления (coupling) этих элементов с помощью вызовов между модулями.

> Попытка разбиения на части модуля, обладающего связностью, приведет лишь к увеличению степени связанности кода и снизит его удобочитаемость.
>
> *Ларри Константин (Larry Constantine)*

Специалисты computer science определили несколько уровней связности, перечисленных далее от наиболее сильной до наиболее слабой:

Функциональная связность

Все части модуля взаимосвязаны, а в модуле содержится все необходимое для его функционирования.

Последовательная связность

Взаимодействие двух модулей, где выходные данные одного становятся входными данными другого.

Коммуникационная связность

Два модуля формируют коммуникационную цепочку, где каждый производит действия над информацией и/или вносит свой вклад в некий результат. Например, добавляет запись в базу данных и создает сообщение электронной почты на основе этой информации.

Процедурная связность

Два модуля выполняют код в определенном порядке.

Временная связность

Связность модулей возникает на основе временных зависимостей. Например, у многих систем имеется перечень, казалось бы, не связанных между собой составляющих, которые должны быть проинициализированы в ходе начального запуска системы; эти разные задачи имеют временную связность.

Логическая связность

Данные внутри модулей связаны не функционально, а логически. Рассмотрим, к примеру, модуль, занимающийся преобразованием информации из текста, сериализованных объектов или потоков. Операции связаны, но функционально они совершенно разные. Типичный пример этого типа связности присутствует практически в каждом проекте Java в виде пакета `StringUtils`, представляющего собой группу статических методов, работающих с данными типа `String`, но в остальном совершенно не связанных друг с другом.

Случайная связность

Элементы в модуле ничем не связаны, кроме того что находятся в одном и том же исходном файле. Это наименее желательная форма связности.

Несмотря на то что перечислено целых семь вариантов *связности*, более точным показателем все же является *связанность (coupling)*. Часто степень связности конкретного модуля определяется только желанием архитектора. Рассмотрим, к примеру, следующее определение модуля:

`Customer Maintenance` (Обслуживание клиентов)

- `add customer` (добавление клиента)
- `update customer` (обновление данных о клиенте)

- `get customer` (получение данных о клиенте)
- `notify customer` (уведомление клиента)
- `get customer orders` (получение заказов от клиента)
- `cancel customer orders` (отмена заказов клиента)

Должны ли два последних пункта быть в этом модуле или разработчику нужно создать два отдельных модуля?

`Customer Maintenance` (Обслуживание клиентов)

- `add customer` (добавление клиента)
- `update customer` (обновление данных о клиенте)
- `get customer` (получение данных о клиенте)
- `notify customer` (уведомление клиента)

`Order Maintenance` (Обслуживание заказов)

- `get customer orders` (получение заказов от клиента)
- `cancel customer orders` (отмена заказов клиента)

Какая из структур правильная? Как всегда, ответ зависит от обстоятельств:

- Ограничивается ли `Order Maintenance` только двумя операциями? Если да, то возможно, есть смысл вернуть эти операции в `Customer Maintenance`.
- Ожидается ли расширение `Customer Maintenance`, что мотивирует разработчиков искать возможности по извлечению из него какой-то части поведения?
- Требуется ли в `Order Maintenance` такая подробная информация о клиенте `Customer`, что при разделении этих двух модулей понадобится высокая степень их связанности между собой, иначе невозможно будет достичь нужной функциональности? Это отсылает нас к цитате Ларри Константина (см. выше).

Эти вопросы являются разновидностью анализа компромиссов, положенного в основу работы архитектора ПО.

Как ни удивительно, но несмотря на субъективность связности, компьютерные специалисты разработали неплохую структурную метрику ее наличия (или, точнее говоря, ее отсутствия). Для оценки конкретных аспектов объектно-ориентированных систем был разработан широко известный набор показателей

Чидамбера и Кемерера — Chidamber and Kemerer Object-oriented metrics suite. В этот набор включено множество общих метрик программного кода, таких как цикломатическая сложность (см. врезку в главе 6 «Цикломатическая сложность» на с. 112) и ряд важных метрик связанности, рассматриваемых далее в разделе «Связанность» на с. 71.

Метрика недостатка связности в методах (Chidamber and Kemerer Lack of Cohesion in Methods, LCOM) определяет структурную связность модуля, обычно компонента. Исходная версия представлена в уравнении 3.1:

$$LCOM = \begin{cases} |P| - |Q|, \text{ если } |P| > |Q| \\ 0 \quad \text{в противном случае} \end{cases}.$$

Уравнение 3.1. LCOM, версия 1

P увеличивается на единицу для любого метода, не обращающегося к конкретному совместно используемому полю, а *Q* уменьшается на единицу для тех методов, которые совместно используют конкретное общее поле. Авторы, конечно, сочувствуют тем, кто не понимает эту формулу. Хуже того, со временем она усложнилась. Вторая версия, появившаяся в 1996 году (и поэтому получившая название LCOM96B), представлена в уравнении 3.2.

$$LCOM = \frac{1}{2}\sum_{j=1}^{a}\frac{m - \mu(Aj)}{m}.$$

Уравнение 3.2. LCOM 96b

Мы не будем тратить время на расшифровку переменных и операторов уравнения 3.2, поскольку прояснить смысл поможет следующее описание. По сути, метрика LCOM выявляет случайную связанность внутри классов. Более понятное определение LCOM выглядит следующим образом:

LCOM

Количество наборов методов, не использующих совместно общие поля.

Рассмотрим класс с приватными полями `a` и `b`. Многие методы обращаются только к полю `a`, многие другие — только к полю `b`. *Количество* наборов методов, не использующих совместно поля (`a` и `b`), значительно, поэтому у данного класса высокий показатель LCOM, обозначающий высокую степень *недостатка связности в методах*. Рассмотрим три класса, показанные на рис. 3.1.

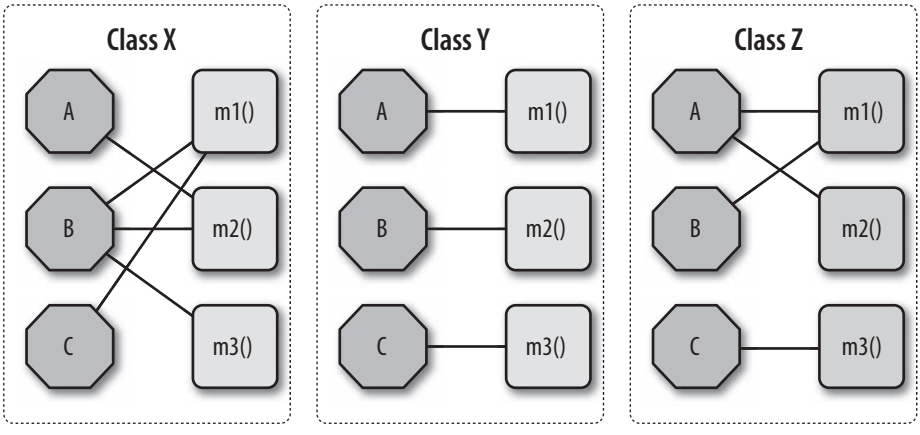


Рис. 3.1. Иллюстрация показателя LCOM, где поля обозначены восьмиугольниками, а методы — квадратами

На рис. 3.1 поля обозначены отдельными буквами, а методы — блоками. У `Class X` метрика LCOM низкая, что является признаком хорошей структурной связности. В свою очередь, у `Class Y` прослеживается недостаток связности; каждая пара поле — метод в `Class Y` может находиться в своем собственном классе, и это никак не повлияет на поведение. В `Class Z` показана смешанная связность, позволяющая разработчикам реструктурировать последнюю комбинацию поле — метод в свой собственный класс.

Метрика LCOM полезна тем архитекторам, которые анализируют кодовые базы для перехода от одного архитектурного стиля к другому. При таких переходах основную проблему представляют общие служебные классы. Использование показателя LCOM помогает архитекторам найти классы, имеющие случайную связанность, код которых изначально не должен был находиться в одном классе.

У многих программных метрик имеются весьма серьезные недостатки, от которых не застрахован и LCOM. Данный показатель может лишь выявить *структурный* недостаток связности; он не способен логически определить, совмещаются отдельно взятые части или нет. Это отсылает нас ко Второму закону архитектуры программного обеспечения: *почему* важнее, чем *как*.

Связанность

К счастью, у нас есть более эффективные инструменты для анализа связанности в кодовых базах, частично основанные на теории графов. Поскольку вызовы метода и возвращения из него формируют граф вызовов, появляется возможность

проведения анализа на основе математических методов. В 1979 году Эдвард Йордон (Edward Yourdon) и Ларри Константин (Larry Constantine) опубликовали книгу «Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design» (издательство Prentice-Hall), дав определения многим ключевым концепциям, включая показатели *афферентной* и *эфферентной* связанности. Афферентная связанность измеряется количеством *входящих* соединений с элементом кода (компонентом, классом, функцией и т. д.). Эфферентная связанность измеряется количеством *исходящих* соединений с другими фрагментами кода. Практически для каждой платформы существуют инструменты, позволяющие архитекторам проводить анализ показателей связанности кода, помогающий в реструктуризации, миграции или лучшем понимании кодовой базы.

ПОЧЕМУ У МЕТРИК СВЯЗАННОСТИ ТАКИЕ ПОХОЖИЕ НАЗВАНИЯ?

Почему две важнейшие метрики в мире архитектуры, представляющие противоположные концепции, имеют практически одинаковые названия, отличающиеся всего лишь гласными с очень похожим звучанием? Эти понятия впервые появились в книге Йордона и Константина «Structured Design...». Позаимствовав понятия из математики, они ввели теперь уже широко распространенные термины афферентной и эфферентной связанности, которые следовало бы назвать входящей и исходящей связанностью. Но, поскольку авторы первоисточника имели склонность к математической симметрии, а не к ясности, разработчики придумали несколько мнемонических правил: буква «а» появляется в английском алфавите перед буквой «е», что соответствует тому, что *входящие* предшествуют *исходящим*; или, другой вариант, первая буква («е») в слове «efferent» та же, что первая букве в слове «exit» (выход), что соответствует исходящим соединениям.

Абстрактность, нестабильность и удаленность от главной последовательности

Исходная величина связанности компонентов очень важна для архитекторов ПО, но ряд других оценочных показателей позволяют провести более глубокий анализ. Эти метрики были выведены Робертом Мартином (Robert Martin) для книги по языку C++, но могут быть применимы и в других объектно-ориентированных языках.

Абстрактность выражается отношением абстрактных элементов (абстрактных классов, интерфейсов и т. д.) к конкретным (реализации). Эта метрика определя-

ет степень абстрактности по сравнению с реализацией. Рассмотрим, к примеру, кодовую базу без абстракции: просто одну огромную функцию кода (как в одном методе `main()`). А в качестве контрпримера возьмем кодовую базу со слишком большим количеством абстракций, что затрудняет разработчикам отслеживание взаимных связей (например, разработчикам требуется некоторое время, чтобы понять, что им делать с `AbstractSingletonProxyFactoryBean`).

Формула абстрактности показана в уравнении 3.3:

$$A = \frac{\Sigma m^a}{\Sigma m^c}.$$

Уравнение 3.3. Абстрактность

В этом уравнении m^a представляет *абстрактные* элементы (интерфейсы или абстрактные классы) в модуле, а m^c — *конкретные элементы* (неабстрактные классы). Метрика абстрактности основана на том же критерии. Проще всего визуализировать ее следующим образом. Представьте себе приложение из 5000 строк кода, целиком содержащихся в методе `main()`. В числителе будет 1, а в знаменателе — 5000: получится абстрактность, близкая к нулю. Таким образом, эта метрика измеряет степень абстракции в вашем коде.

Архитектор находит *абстрактность*, вычисляя отношение суммы абстрактных артефактов к сумме конкретных.

Еще одна производная метрика, *нестабильность*, определяется как отношение эфферентной связанности к сумме эфферентной и афферентной связанности, что показано в уравнении 3.4:

$$I = \frac{C^e}{C^e + C^a}.$$

Уравнение 3.4. Нестабильность

В этом уравнении C^e представляет *эфферентную* (или исходящую) связанность, а C^a — *афферентную* (или входящую).

Метрика *нестабильности* определяет волатильность кодовой базы. Чем выше нестабильность, тем легче кодовая база ломается при внесении изменений, поскольку у нее высокая степень связанности. Например, если класс вызывает слишком много других классов для того, чтобы перераспределить работу, то у него высокая вероятность возникновения поломок при внесении изменений в один или несколько вызываемых методов.

Расстояние от главной последовательности

Одним из немногих комплексных показателей для оценки архитектурной структуры является *расстояние от главной последовательности* — производная метрика, основанная на метриках *нестабильности* и *абстрактности*. Формула метрики показана в уравнении 3.5:

$$D = |A + I - 1|.$$

Уравнение 3.5. Удаленность от главной последовательности

Здесь A = абстрактность, а I = нестабильность.

Обратите внимание, что *абстрактность* и *нестабильность* являются относительными долями, то есть их значение всегда находится между 0 и 1. Поэтому, изображая отношение графически, мы получим схему, показанную на рис. 3.2.

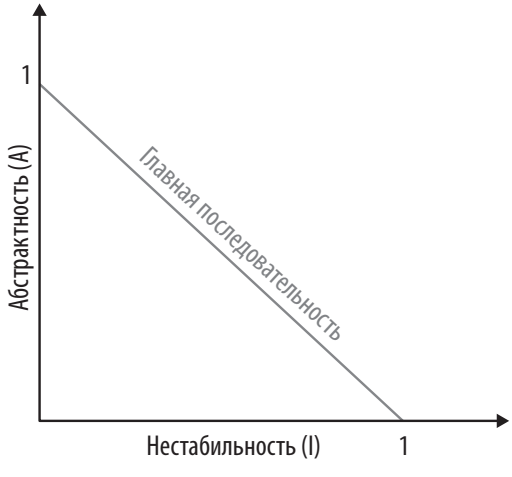


Рис. 3.2. Главная последовательность определяет идеальное соотношение абстрактности и нестабильности

Метрика *расстояния* представляется как идеальное соотношение абстрактности и нестабильности. Компоненты, которые располагаются вблизи этой идеализированной линии, имеют сбалансированное сочетание этих двух конкурирующих показателей. Размещение на графике конкретного компонента (рис. 3.3) позволяет разработчикам вычислить метрику *расстояния от главной последовательности*.

Рис. 3.3. Нормализованное расстояние от главной последовательности для конкретного компонента

На рис. 3.3 разработчики нанесли на график исследуемый класс, затем замерили расстояние от идеализированной прямой. Чем ближе к прямой, тем лучше сбалансирован класс. Слишком удаленные в направлении верхнего правого угла классы входят в так называемую *зону бесполезности (zone of uselessness)*: слишком абстрактный код создает трудности при использовании. С другой стороны, код, далеко отстоящий от прямой в направлении нижнего левого угла, попадает в *зону мучений (zone of pain)*: слишком большая доля реализации и недостаточная абстрактность приводят к хрупкости кода и сложности в сопровождении. Это показано на рис. 3.4.

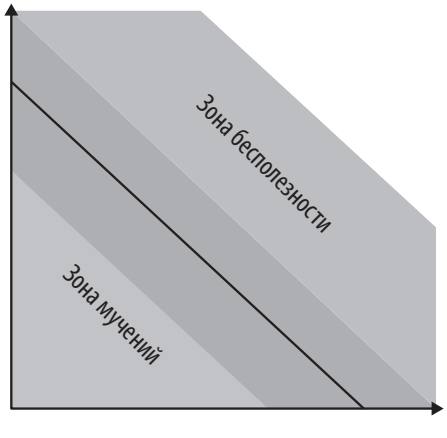


Рис. 3.4. Зона бесполезности и зона мучений

Инструменты, способные выдать эти показатели, имеются на многих платформах. Они помогают архитекторам ПО выполнять анализ кодовых баз при незнании кода, миграции или оценке технического долга.

ОГРАНИЧЕННОСТЬ МЕТРИК

Хотя в сфере разработки ПО имеется ряд метрик на уровне кода, дающих ценную информацию о кодовых базах, наши инструменты все же довольно грубы по сравнению с другими инженерными дисциплинами. Интерпретация требуется даже для тех метрик, которые выведены непосредственно из структуры кода. Например, цикломатическая сложность (см. врезку в главе 6 «Цикломатическая сложность» на с. 112) является показателем сложности в кодовых базах, но не может отделить базовую сложность (обусловленную сложностью поставленной задачи) от случайно допущенной сложности (то есть код получился сложнее, чем мог бы быть). Интерпретации требуют практически все метрики на уровне кода.

Но все же полезно установить базовые значения для наиболее важных показателей, таких как цикломатическая сложность, чтобы архитекторы смогли оценить, с каким именно типом сложности они имеют дело. Создание соответствующих тестов рассматривается в разделе «Управление и функции пригодности» на с. 115.

Следует заметить, что ранее упомянутая книга Эдварда Йордона и Ларри Константина («Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design») появилась до того, как набрали популярность объектно-ориентированные языки, и речь в ней идет в основном об особенностях структурного программирования — функциях (а не методах). В книге также дается определение другим типам связанности, которые здесь не упоминаются, потому что их место заняло другое понятие — *коннасценция*.

Коннасценция

В 1996 году вышла книга Мейлира Пейдж-Джонса (Meilir Page-Jones) «What Every Programmer Should Know About Object-Oriented Design» (издательство Dorset House) с уточнением метрик афферентной и эфферентной связанности и их адаптацией под объектно-ориентированные языки с использованием концепции, названной им *коннасценцией* (*connascence*). Этому понятию автор дал следующее определение:

> Два компонента считаются коннасцентными, если изменения, внесенные в один из них, потребуют модификации другого для поддержания общей работоспособности системы.
>
> *Мейлир Пейдж-Джонс*

Он определил два типа коннасценции: *статическую* и *динамическую*.

Статическая коннасценция

Под *статической коннасценцией* подразумевается связанность на уровне исходного кода (в отличие от связанности на уровне времени выполнения, рассматриваемой ниже в подразделе «Динамическая коннасценция» на с. 78); это уточнение афферентных и эфферентных связанностей, определенных в книге «Structured Design...». Иными словами, архитекторы считают *степенями* афферентной или эфферентной связанности каких-либо элементов следующие формы статической коннасценции:

Коннасценция имени (Connascence of Name, CoN)

Несколько компонентов должны согласовать имя сущности.

Имена методов являются наиболее распространенным и желательным видом связанности кодовых баз, особенно в свете современных инструментов рефакторинга, с легкостью справляющихся с общесистемными изменениями имен.

Коннасценция типа (Connascence of Type, CoT)

Несколько компонентов должны согласовать тип сущности.

Этот вид коннасценции имеется во многих языках со статической типизацией и ограничивает принадлежность переменных и параметров к конкретным типам. Но это не чисто языковая функция: выборочная типизация существует и в ряде языков с динамической типизацией, особенно это характерно для Clojure[1] и Clojure Spec[2].

Коннасценция смысла (Connascence of Meaning, CoM), или коннасценция соглашения (Connascence of Convention, CoC)

Несколько компонентов должны согласовать смысл конкретных значений.

Самым распространенным и очевидным случаем этого типа коннасценции в кодовой базе являются жестко запрограммированные числа, заменяющие

[1] *clojure.org*

[2] *clojure.org/about/spec*

константы. Например, в некоторых языках принято, что где-то имеется определение `int TRUE = 1; int FALSE = 0`. Представьте, какие проблемы возникнут, если смысл этих значений не будет очевиден.

Коннасценция позиции (Connascence of Position, CoP)

Несколько сущностей должны согласовать порядок следования значений.

Этот вопрос касается значений параметров для вызовов методов и функций даже в языках со статической типизацией. Например, если разработчик создает метод `void updateSeat(String name, String seatLocation)` и вызывает его со значениями `updateSeat("14D", "Ford, N")`, то нарушение семантики происходит даже при соблюдении типов.

Коннасценция алгоритма (Connascence of Algorithm, CoA)

Несколько компонентов должны согласовать конкретный алгоритм.

Довольно распространенным примером этой формы коннасценции может послужить определение разработчиком алгоритма защитного хеширования, который должен запускаться как на сервере, так и на клиентской машине и выдавать одинаковые результаты аутентификации пользователя. Несомненно, данный тип представляет весьма высокую форму коннасценции: если в любом из алгоритмов будет что-либо изменено, процедура подтверждения соединения больше не сработает.

Динамическая коннасценция

Еще одним типом коннасценции, согласно определению Пейдж-Джонса, является *динамическая коннасценция*, которая анализирует вызовы во время выполнения программы. Ниже приводится описание различных форм динамической коннасценции.

Коннасценция исполнения (Connascence of Execution, CoE)

Порядок выполнения компонентов имеет значение.

Рассмотрим следующий код:

```
email = new Email();
email.setRecipient("foo@example.com");
email.setSender("me@me.com");
email.send();
email.setSubject("whoops");
```

Этот код не будет правильно работать, потому что конкретные свойства должны быть заданы в определенном порядке.

Коннасценция синхронности (Connascence of Timing, CoT)

Время выполнения нескольких компонентов имеет значение.

Весьма распространенным примером этого типа коннасценции может послужить состояние гонки, вызванное двумя потоками, выполняющимися одновременно и оказывающими влияние на результат совместной операции.

Коннасценция значений (Connascence of Values, CoV)

Возникает, когда несколько значений зависят друг от друга и должны изменяться вместе.

Рассмотрим случай, когда разработчик определил прямоугольник четырьмя точками, представляющими его углы. Чтобы не нарушить целостность структуры данных, специалист не должен вносить произвольные изменения в одну из точек без учета влияния на другие точки.

Еще более распространенный и проблематичный случай связан с транзакциями, особенно в распределенных системах. Когда архитектору ПО, проектирующему систему с отдельными базами данных, требуется обновить конкретное значение во всех базах данных, все значения должны изменяться вместе или же не изменяться вообще.

Коннасценция идентичности (Connascence of Identity, CoI)

Возникает, когда несколько компонентов должны ссылаться на одну и ту же сущность.

Распространенный пример этой формы коннасценции — наличие двух независимых компонентов, которые совместно используют и обновляют общую структуру данных, например распределенную очередь.

Архитекторам ПО гораздо труднее определять динамическую коннасценцию, чем статическую. Инструменты, которые могли бы анализировать вызовы во время выполнения программы с той же эффективностью, что анализ графа вызовов, отсутствуют.

Свойства коннасценции

Коннасценция является инструментом анализа для архитекторов и разработчиков ПО, и некоторые ее свойства позволяют грамотно использовать этот инструмент. Ниже приводится описание каждого из таких свойств:

Сила

Архитекторы определяют *силу* коннасценции по той легкости, с которой разработчик может сделать рефакторинг соответствующего типа связанности; из

изображения на рис. 3.5 следует, что одни типы коннасценции явно предпочтительнее других. С помощью рефакторинга, направленного на поиск более выгодных типов коннасценции, архитекторы и разработчики могут улучшить характеристики связанности своей кодовой базы.

Архитекторам следует отдавать предпочтение не динамической, а статической коннасценции, поскольку разработчики могут выявить ее с помощью обычного анализа исходного кода, а современные инструменты упрощают совершенствование ее характеристик. Рассмотрим, к примеру, случай *коннасценции смысла*, которую разработчики могут улучшить рефакторингом до *коннасценции имени*, создав именованные константы вместо так называемых магических чисел.

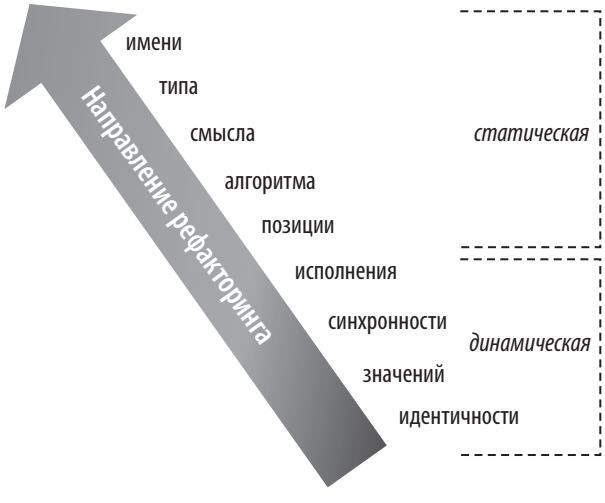


Рис. 3.5. Степень коннасценции может послужить хорошим признаком потребности в рефакторинге

Локальность

Локальность коннасценции измеряет, насколько близко друг к другу в кодовой базе находятся элементы. Близко расположенный код (в одном и том же модуле) имеет более высокие формы коннасценции, чем отдаленные друг от друга фрагменты кода (которые находятся в разных модулях или кодовых базах). Иными словами, когда формы коннасценции, указывающие на слабую связанность, относятся к отдаленным элементам, это лучше, чем когда они относятся к близким элементам. Например, если два класса в одном и том же компоненте обладают коннасценцией смысла, это приносит меньший ущерб кодовой базе, чем наличие этой же формы коннасценции у двух отдельных компонентов.

Разработчики должны рассматривать силу и локальность в едином ключе. Более сильные формы коннасценции, выявленные внутри одного и того же модуля, принесут меньше проблем, чем те же формы коннасценции, разнесенные по разным модулям.

Степень

Степень коннасценции зависит от меры ее воздействия: она влияет на ограниченное количество классов или на многие классы? Чем меньше степень коннасценции, тем меньше ущерб, наносимый кодовой базе. Иными словами, иметь высокую степень динамической коннасценции не страшно, если у вас всего лишь несколько модулей. Но кодовая база склонна к разрастанию, и маленькая проблема легко превращается в большую.

Пейдж-Джонс рекомендует три способа работы с коннасценцией, чтобы улучшить модульности систем:

1. Свести общую коннасценцию к минимуму за счет разбиения системы на инкапсулированные элементы.
2. Минимизировать любую остающуюся коннасценцию, пересекающую границы инкапсуляции.
3. Довести до максимума коннасценцию внутри границ инкапсуляции.

Легендарный рационализатор архитектуры программного обеспечения Джим Вейрих (Jim Weirich) вновь взялся за популяризацию понятия коннасценции и дает два полезных совета:

> **Правило степени:** сильные формы коннасценции следует переделывать в слабые.
>
> **Правило локальности:** по мере увеличения расстояния между элементами программы следует использовать все более слабые формы коннасценции.

Объединение показателей связанности и коннасценции

До сих пор связанность и коннасценция рассматривались исходя из предположения, что их показатели относятся к разным областям и применяются с разными целями. Но с точки зрения архитектора, эти два показателя пересекаются. То, что Пейдж-Джонс назвал статической коннасценцией, относится к степеням как входящей, так и исходящей связанности. В структурном программировании интересуются только тем, что на входе или на выходе, а коннасценция рассматривает, как именно элементы связаны между собой. Чтобы получить

наглядное представление о том, как именно эти концепции пересекаются, рассмотрим схему на рис. 3.6.

Рис. 3.6. Объединение связанности и коннасценции

На рис. 3.6 концепции связанности структурного программирования показаны слева, а характеристики коннасценции — справа. Коннасценция определяет, как должно быть реализовано то, что в структурном программировании называется *связанностью данных* (вызовы методов). Структурное программирование фактически не затрагивает области, охваченные динамической коннасценцией; краткое изложение этой концепции рассматривается в разделе «Архитектурные кванты и гранулярность» на с. 125.

Проблемы с коннасценцией 1990-х годов

При использовании этих ценных показателей для проведения анализа и проектирования систем у архитекторов возникает ряд проблем. Во-первых, эти показатели позволяют проанализировать особенности низкоуровневого кода, сосредоточиваясь на качестве кода и его чистоте, а не на нужной архитектурной структуре. Архитекторам ПО свойственно больше интересоваться тем, *как* именно связаны модули, а не *степенью* связанности. Например, сравнение синхронного обмена данными с асинхронным более интересно, чем то, как все это реализовано.

Во-вторых, проблема с коннасценцией заключается в том, что она, по сути, не имеет никакого отношения к фундаментальному решению, которое приходится принимать многим современным архитекторам: синхронным или асинхронным должен быть обмен данными в распределенных архитектурах типа микросервисов? Возвращаясь к Первому закону архитектуры ПО, вспомним, что все

соткано из компромиссов. После рассмотрения свойств архитектуры в главе 7 мы расскажем о новых способах осмысления современной коннасценции.

От модулей к компонентам

Понятие *модуль* используется повсеместно как общее название для объединения взаимосвязанного кода. Однако большинство платформ поддерживает ту или иную форму *компонента*, одного из основных строительных блоков для архитектуры программного обеспечения. Концепция и соответствующий анализ физического или логического разделения существовали еще на заре компьютерной науки. И все же, что бы там ни думали и ни писали о компонентах и разделении, разработчикам и архитекторам до сих пор не удалось окончательно достичь хороших результатов.

Выделение компонентов из проблемных областей будет рассмотрено в главе 8, но сначала нужно разобраться с другими фундаментальными аспектами архитектуры ПО: свойствами архитектуры и областью их действия.

ГЛАВА 4

Основные свойства архитектуры

Когда компания планирует решить определенную задачу с помощью программного обеспечения, она формирует список требований к создаваемой системе. Существует масса методик, по которым происходит сбор информации, но обычно требования определяются командой разработчиков в процессе разработки ПО. Однако архитектор должен учитывать и множество других факторов при проектировании программно-аппаратного решения (рис. 4.1).

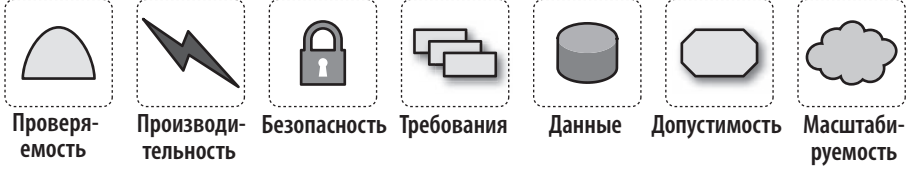


Рис. 4.1. Программное решение состоит как из требований предметной области, так и из свойств архитектуры

Архитектор ПО может участвовать в определении технических требований с точки зрения предметной области или бизнес-требований, но одна из его ключевых обязанностей заключается в определении, выявлении или применении всего того, что должно делать программное обеспечение, но что не имеет прямого отношения к предметной функциональности, а именно *свойств архитектуры*.

Чем отличается выстраивание архитектуры ПО от программирования и проектирования? Многим, и в том числе ролью, которую играют архитекторы в определении свойств архитектуры, то есть важных аспектов системы, не зависящих от области решаемой задачи. Многие организации называют эти характеристики программного средства по-разному, в том числе и *нефункциональными требованиями*, но нам этот термин не нравится из-за оттенка пренебрежения. Архитекторы придумали этот термин, чтобы отделить свойства

архитектуры от *функциональных требований*, но называя что-либо *нефункциональным*, мы получаем негативный смысловой оттенок, а как убедить команду уделять достаточное внимание чему-то «нефункциональному»? Не нравится нам и другой термин — *атрибуты качества*, — поскольку им определяется не проектирование, а оценка качества готового продукта. Мы предпочитаем термин *свойства архитектуры*, потому что им описываются понятия, играющие решающую роль в успехе архитектуры, а следовательно, и системы в целом, без обесценивания их важности.

Свойства архитектуры должны отвечать трем критериям:

- определять взгляд на проектирование, не относящийся к предметной области (непредметный взгляд);
- влиять на ряд структурных аспектов проектирования;
- играть решающую или весьма важную роль в успехе приложения.

Эти взаимосвязанные части нашего определения показаны на рис. 4.2.

Рис. 4.2. Отличительные черты свойств архитектуры

Определение, показанное на рис. 4.2, состоит из трех компонентов, перечисленных ниже.

Непредметный взгляд на проектирование

При проектировании требования определяют, что должно делать приложение, а архитектурные свойства задают эксплуатационные и проектные

критерии успеха, имеющие отношение к способам реализации требований и к причинам выбора конкретных вариантов. Например, важное архитектурное свойство определяет уровень производительности приложения, который часто не отражается при формировании требований. Еще более удачным примером будет отсутствие во всех документах с требованиями упоминаний об «устранении технического долга», хотя это вполне стандартное требование для архитекторов и разработчиков. Более подробно разница между явно выраженными и подразумеваемыми свойствами будет раскрыта в разделе «Выбор архитектурных свойств на основе задач предметной области» на с. 96.

Влияние на ряд структурных аспектов проектирования

Основная причина, по которой архитекторы стремятся дать описание архитектурных свойств, обусловлена вопросами проектирования: требует ли успешная реализация того или иного архитектурного свойства какой-то особой структуры? Например, практически в каждом проекте затрагиваются вопросы *безопасности*, и во всех системах должны быть соблюдены базовые меры предосторожности при проектировании и программировании. Но когда архитектор работает над каким-то особым проектом, безопасность выходит на уровень архитектурного свойства. Рассмотрим два случая проведения оплаты в некой системе:

Сторонний платежный процессор

Если реквизиты платежа обрабатываются средством, интегрированным с приложением, архитектуре не нужна никакая особая структура. В дизайн должны включаться стандартные средства безопасности, например шифрование и хеширование, но специальная структура ей не требуется.

Обработка платежей в самом приложении

Если обработку платежей будет выполнять приложение, архитектор может спроектировать для этого специальный модуль, компонент или сервис, чтобы структурно изолировать код с особыми мерами безопасности. В данном случае архитектурные свойства оказывают влияние как на архитектуру, так и на само проектирование.

Разумеется, зачастую для принятия подобного решения недостаточно даже этого критерия: здесь могут приниматься в расчет прошлые инциденты, связанные с безопасностью, особенности интеграции со сторонними средствами и множество других критериев. В данном случае показаны лишь некоторые факторы, учитываемые архитекторами при реализации в проекте конкретных возможностей.

Решающая или весьма важная роль в успехе приложения

> Приложения *могут* поддерживать огромное количество архитектурных свойств... но вряд ли в этом присутствует какой-либо здравый смысл. Поддержка каждого архитектурного свойства усложняет проектирование. Поэтому архитектору важнее всего выбрать наименьший, а не наибольший из возможных наборов архитектурных свойств.

Затем мы делим архитектурные свойства на явные и неявные. Последние редко фигурируют в требованиях, но они необходимы для успеха проекта. Например, доступность, надежность и безопасность составляют основу практически всех приложений, но редко указываются в проектной документации. Архитектору следует применить свои знания предметной области, чтобы выявить необходимые архитектурные свойства еще на этапе анализа. К примеру, компании, занятой высокочастотным трейдингом, вряд ли стоит указывать, что в каждой системе требуется малое время отклика. Но архитекторы в данной предметной области должны быть в курсе важности этого свойства. Явно выраженные архитектурные свойства указываются в технических заданиях или же в других конкретных инструкциях.

Треугольник на рис. 4.2 выбран неспроста: каждый элемент определения поддерживает другие элементы, а те, в свою очередь, поддерживают общую конструкцию системы. Жесткая конструкция треугольника символизирует частое взаимодействие архитектурных свойств друг с другом, что заставляет архитекторов повсеместно вводить в обиход понятие *компромисса*.

Список (неполный) архитектурных свойств

Архитектурные свойства присущи широкому спектру программных систем, начиная с таких свойств низкоуровневого кода, как модульность, и заканчивая сложными эксплуатационными понятиями, такими как масштабируемость и адаптируемость. Несмотря на предпринимаемые в прошлом попытки систематизации этих понятий, какого-либо универсального стандарта так и не появилось. Каждая организация имеет собственную интерпретацию. Кроме того, поскольку экосистема программного обеспечения изменяется слишком быстро, постоянно появляются новые концепции, термины, показатели и средства контроля, предоставляя новые возможности для определения архитектурных свойств.

Несмотря на объем и масштабность архитектурных свойств, архитекторы обычно разделяют их на более общие категории. Описание некоторых из них с примерами дается в следующих разделах.

Эксплуатационные свойства архитектуры

К эксплуатационным свойствам архитектуры относятся такие характеристики, как производительность, масштабируемость, адаптируемость, доступность и надежность. Некоторые из таких свойств приведены в табл. 4.1.

Таблица 4.1. Наиболее распространенные эксплуатационные свойства архитектуры

Термин	Определение
Доступность	Время доступности системы (если предполагается работа в режиме 24/7, необходимо сделать все возможное для быстрого восстановления работоспособности в случае сбоя)
Бесперебойность	Способность аварийного восстановления
Производительность	Включает стресс-тестирование, анализ пиковых нагрузок, анализ частоты использования функций, потенциальных возможностей и времени отклика. Иногда для подтверждения необходимого уровня производительности требуются длительные прогоны на протяжении нескольких месяцев
Восстанавливаемость	Требования к непрерывности работы по прямому назначению (например, скорость выхода системы из аварийных ситуаций). Влияет на стратегию резервного копирования и на требования к дублированию оборудования
Надежность и безопасность	Потребность в безотказной работе системы или особая важность ее функционирования в структурах жизнеобеспечения. Также отвечает на вопрос: «Насколько разорительным для компании станет сбой в работе системы?»
Робастность	Способность к обработке ошибок в ходе выполнения программы и предельных условий работы при отключении от интернета или же при потере сетевого напряжения или аппаратном сбое
Масштабируемость	Работоспособность системы при росте числа пользователей или ужесточении уровня технических требований

Во многих программных проектах эксплуатационные свойства архитектуры тесно переплетаются с вопросами практического использования программных средств и применения к ним методологии DevOps.

Структурные свойства архитектуры

Архитекторы должны заботиться и о структуре кода. Зачастую архитектор несет персональную или разделенную ответственность за качество кода, например за

его высокую модульность, контролируемую связанность его компонентов, удобочитаемость и за выполнение множества других внутренних оценок качества.

Некоторые структурные свойства архитектуры перечислены в табл. 4.2.

Таблица 4.2. Структурные свойства архитектуры

Термин	Определение
Конфигурируемость	Возможность легкого изменения конфигурации программного средства конечными пользователями (через UI)
Расширяемость	Насколько важно иметь возможность подключения новых функций
Инсталлируемость	Простота установки на все необходимые платформы
Возможность многократного использования	Возможность использования общих компонентов в нескольких продуктах
Локализуемость	Возможность поддержки нескольких языков при вводе текста или запросов на экранах с полями данных, а в отчетах — выполнение требований по использованию многобайтовых символов и единиц измерений или валют
Сопровождаемость	Легкость внесения в систему изменений и улучшений
Переносимость	Потребность запуска системы на более чем одной платформе. (Например, должен ли интерфейс работать как с Oracle, так и с SAP DB?)
Поддерживаемость	Уровень технической поддержки, который требуется приложению. На каком уровне требуется журналирование и применение других средств контроля для устранения имеющихся ошибок?
Обновляемость	Способность просто и быстро обновляться с прежней версии данного приложения или решения к новой версии на серверных и клиентских машинах

Сквозные свойства архитектуры

Многие свойства архитектуры довольно легко разбиваются на категории, но есть множество других, которые выпадают из общего ряда или не поддаются категоризации, но устанавливают дополнительные правила проектирования. Некоторые из них перечислены в табл. 4.3.

Любой перечень архитектурных свойств не будет исчерпывающим: всегда может появиться необходимость в важных архитектурных свойствах, основанных на уникальных факторах (см., к примеру, далее врезку «Итальянность» на с. 91).

Таблица 4.3. Сквозные свойства архитектуры

Термин	Определение
Доступность	Доступность программного средства для всех пользователей, включая людей с ограниченными возможностями, например страдающих дальтонизмом или глухотой
Архивируемость	Нужно ли архивировать или удалять данные по истечении определенного срока? (Например, учетные данные клиентов подлежат удалению по истечении трех месяцев или получают метку устаревших и сбрасываются в архив во вторичной базе данных для последующего обращения к ним)
Аутентифицируемость	Требования мер безопасности, позволяющие убедиться в подлинности личности пользователей
Авторизируемость	Требования мер безопасности, позволяющие открывать пользователям доступ только к строго определенным функциям приложения (по сценарию использования, подсистеме, веб-странице, бизнес-правилу, локальному уровню и т. д.)
Правомерность	Каковы правовые ограничения работы системы (защита данных, закон Сарбейнса — Оксли, GDPR и т. д.). Сохранение каких прав нужно компании? Имеются ли какие-либо правила относительно способа создания или развертывания приложения?
Конфиденциальность	Способность скрывать транзакции от сотрудников самой компании (подразумевает использование зашифрованных транзакций, невидимых даже администраторам баз данных и сетевым архитекторам)
Безопасность	Нужно ли шифровать информацию в базе данных? Нужно ли шифровать информацию при обмене данными по сети между внутренними системами? Какой тип аутентификации следует применять для удаленного доступа пользователей?
Поддерживаемость	Какой уровень технической поддержки требуется приложению? На каком уровне требуется журналирование и применение других средств контроля для устранения имеющихся ошибок?
Удобство и простота использования	Уровень подготовки, необходимый пользователям для достижения намеченных ими целей с помощью приложения. К удобству и простоте использования нужно относиться так же серьезно, как и к любому другому вопросу архитектуры приложения

Кроме всего прочего, многие из предыдущих терминов не отличаются особой точностью и конкретностью из-за каких-то деталей или отсутствия объективных определений. Например, *интероперабельность (interoperability)* и *совместимость (compatibility)* могут показаться эквивалентными, что для некоторых систем будет вполне справедливо. Но они отличаются друг от друга,

поскольку *интероперабельность* подразумевает простоту интеграции с другими системами, что, в свою очередь, обозначает наличие опубликованных, задокументированных API-интерфейсов. А понятие *совместимости* больше относится к отраслевым стандартам и предметной области. Еще одним примером может послужить *изучаемость (learnability)*. Одним из значений этого понятия является легкость обучения пользователей работе с программой, а другим — уровень автоматического изучения системой своей рабочей среды с применением алгоритмов машинного обучения, достаточный для ее самонастраиваемости или самооптимизируемости.

ИТАЛЬЯННОСТЬ

Коллега Нила любит рассказывать историю об уникальной природе архитектурных свойств. Шла работа с клиентом, которому требовалась централизованная архитектура. Однако после каждого предложенного варианта заказчик задавал один и тот же вопрос: «А что будет, если мы потеряем Италию?». Несколько лет назад из-за внезапных сбоев была утрачена связь головного офиса с итальянскими филиалами, что нанесло урон организационного плана. Поэтому ко всем будущим архитектурам предъявлялось жесткое требование, которое команда в конечном итоге назвала *итальянностью*, что на самом деле означало уникальное сочетание доступности, восстанавливаемости и устойчивости к внешним воздействиям.

Многие определения пересекаются. Рассмотрим, к примеру, доступность и надежность, которые, казалось бы, практически во всех случаях перекликаются друг с другом. Однако возьмем протокол UDP, который лежит в основе TCP. UDP доступен по IP, но ненадежен: пакеты могут приходить не по порядку, и получателю, возможно, придется запрашивать пропущенные пакеты еще раз.

Полного перечня стандартов не существует. Международная организация по стандартизации (ISO) публикует перечень, сгруппированный по функциональности. По большому счету, он представляет собой далеко не полный список категорий, в котором частично дублируются многие из перечисленных нами свойств. Ниже приведены некоторые из ISO-определений:

Производительность (Performance efficiency)

Метрика производительности применительно к количеству ресурсов, используемых при известных условиях. К ней относятся *временные характеристики* (скорость отклика, сроки обработки и/или пропускная способность), *потреб-*

ление ресурсов (объемы и типы задействованных ресурсов) и *загруженности* (степень превышения максимально установленных пределов).

Совместимость (Compatibility)

Метрика способности продукта, системы или компонента обмениваться информацией с другими продуктами, системами или компонентами и/или выполнять возложенные функции при совместном использовании аппаратной или программной среды. Включает понятия *сосуществования* (возможности успешного выполнения функций при использовании среды и ресурсов совместно с другими продуктами) и *интероперабельности* (насколько две и более системы могут обмениваться и использовать информацию).

Удобство и простота использования (Usability)

Метрика эффективности, результативности и успешности использования системы в намеченных пользователем целях. Включает понятия *простоты выявления пригодности* (пользователи могут определить, отвечает ли программное средство их нуждам), *изучаемости* (насколько легко пользователи могут научиться пользоваться программным средством), *защиты от ошибок пользователя* (защита от ошибок, совершаемых пользователями) и *доступности* (сделать программное средство доступным для людей с разными особенностями и возможностями).

Надежность (Reliability)

Насколько система сохраняет работоспособность при определенных условиях и за определенный период времени. Это свойство разбивается на подкатегории *зрелости* (соответствует ли ПО требованиям надежности в нормальных условиях), *доступности* (ПО работоспособно и доступно), *отказоустойчивости* (работает ли ПО по назначению, несмотря на аппаратные или программные сбои) и *восстанавливаемости* (может ли ПО восстановиться после сбоя, восстановив поврежденные данные и вернувшись в желаемое состояние системы).

Безопасность (Security)

Насколько ПО защищает информацию и данные, чтобы люди либо другие продукты или системы имели степень доступа к данным, соответствующую их типам и уровням авторизации. К данному семейству свойств относятся *конфиденциальность* (данные доступны только тем, кто к ним допущен), *неприкосновенность* (ПО предотвращает несанкционированный доступ или внесение изменений в программу либо в данные), *неподдельность* (возможность доказать подлинность действий или событий), *подотчетность*

(возможность отслеживания действий пользователя) и *достоверность* (подтверждение личности пользователя).

Сопровождаемость (Maintainability)

Степень эффективности и результативности, с которой разработчики могут модифицировать ПО с целью его совершенствования, доработки или адаптации к изменениям среды и/или требований. Включает понятия *модульности* (насколько ПО составлено из отдельных компонентов), *возможности многократного использования* (степени, в которой разработчики могут использовать тот или иной объект более чем в одной системе либо при создании других объектов), *анализируемости* (насколько просто разработчики могут собрать конкретные показатели работы ПО), *модифицируемости* (степени допустимости внесения изменений, при которой не будет нарушена работа и не будет снижено качество приложения) и *тестируемости* (насколько легко разработчикам или кому-либо еще тестировать ПО).

Переносимость (Portability)

Насколько легко разработчики могут перенести систему, продукт либо компонент с одной аппаратной, программной или иной рабочей среды или среды использования в другую. Это свойство разбивается на *адаптируемость* (могут ли разработчики эффективно и результативно адаптировать ПО к другой или более развитой аппаратной, программной или иной рабочей среде либо среде использования), *инсталлируемость* (можно ли установить и/или удалить ПО из конкретной среды) и *заменяемость* (насколько просто разработчики могут заменить имеющиеся функциональные возможности другим ПО).

Последний пункт в перечне ISO касается тех функциональных аспектов программных средств, которых, по нашему мнению, не должно быть в этом списке:

Функциональная пригодность (Functional suitability)

Степень предоставления продуктом или системой функций, удовлетворяющих заявленным и предполагаемым потребностям при использовании в определенных условиях. Ее составляющими являются:

Функциональная законченность (Functional completeness)

Степень охвата набором функций всех указанных задач и пользовательских целей.

Функциональная корректность (Functional correctness)

Степень предоставления продуктом или системой корректных результатов с необходимым уровнем точности.

Функциональная целесообразность (Functional appropriateness)

Насколько функции облегчают выполнение определенных задач и достижение поставленных целей. Это не свойство архитектуры, а скорее мотивационные установки для создания ПО. Здесь показывается, какое развитие получило представление о взаимосвязанности архитектурных свойств и предметной области. Это развитие будет рассмотрено в главе 7.

МНОГОЧИСЛЕННЫЕ НЕОПРЕДЕЛЕННОСТИ В АРХИТЕКТУРЕ ПО

Отсутствие четких определений многих критически важных аспектов, включая задачи самой архитектуры ПО, вызывает у архитекторов постоянное недовольство! Компаниям приходится вводить свою собственную терминологию, и это приводит к неразберихе во всей отрасли, поскольку архитекторы используют либо малопонятные термины, либо, что еще хуже, применяют одни и те же термины для обозначения совершенно разных понятий. Как бы нам ни хотелось, мы не можем навязывать миру разработки ПО какую-то стандартную терминологию. Но мы сами следуем и всем рекомендуем следовать положениям, выработанным в области предметно-ориентированного проектирования, чтобы все же найти с коллегами общий язык и снизить количество недоразумений, связанных с терминологией.

Компромиссы и наименее худшая архитектура

В силу разных причин приложения могут поддерживать только некоторые из перечисленных нами архитектурных свойств. Во-первых, каждое поддерживаемое свойство требует усилий по проектированию и, возможно, структурной поддержки. Во-вторых, более существенной проблемой является то, что каждое архитектурное свойство зачастую оказывает влияние на другие свойства. Например, если архитектор стремится повысить *безопасность*, это практически всегда негативно отражается на *производительности*: в приложении приходится повышать объемы шифрования, выполняемого в режиме реального времени, косвенных обращений с целью сохранения секретной информации и предпринимать ряд других мер, потенциально снижающих производительность.

Эту взаимосвязь можно проиллюстрировать следующей метафорой. Говорят, что пилотам трудно научиться управлять вертолетом, поскольку при этом требуется отдельно контролировать движения обеих рук и ног, причем любое движение может повлиять на другие. Поэтому при управлении приходится балансировать, что напоминает поиск компромиссов при выборе архитектурных свойств. Каж-

дое архитектурное свойство, для которого архитектор разрабатывает поддержку, усложняет общий дизайн.

По сути, архитекторам не нужно проектировать систему, где каждое архитектурное свойство максимально выражено. Чаще всего решения сводятся к нахождению компромиссов между несколькими конкурирующими интересами.

Всегда стремитесь не к лучшей, а к наименее худшей архитектуре.

Переизбыток архитектурных свойств приводит к появлению универсальных решений, подходящих для любой бизнес-задачи. Но такие архитектуры редко бывают работоспособными из-за громоздкости дизайна.

Из этого следует, что архитекторам нужно стремиться к поэтапному (итеративному) подходу. Если архитектура легче поддается изменениям, можно меньше беспокоиться о поиске абсолютно правильных решений с первой попытки. Одним из важнейших уроков разработки в стиле Agile является ценность последовательных улучшений. Данное утверждение справедливо для всех уровней разработки, включая выстраивание архитектуры.

ГЛАВА 5

Выбор архитектурных свойств

Одним из первых шагов по созданию новой или по оценке пригодности существующей архитектуры является выбор основных архитектурных свойств. Определение ключевых свойств (выражаемых словами с окончанием на *-ость*) для конкретной задачи или приложения требует от архитектора не только компетентности в предметной области, но и сотрудничества с основными стейкхолдерами.

Архитектор выявляет архитектурные свойства как минимум тремя путями: на основе задач предметной области, на основе требований и на основе неявных особенностей предметной области. Ранее мы уже рассматривали неявные свойства, а об остальных поговорим далее.

Выбор архитектурных свойств на основе задач предметной области

Чтобы понять, каким архитектурным свойствам уделить больше внимания, архитектор должен проанализировать задачи предметной области. Например, необходимо выяснить: действительно ли масштабируемость является самым значимым свойством либо основной упор следует сделать на отказоустойчивости, безопасности или производительности? Возможно, системе требуется сочетание всех четырех свойств? Понимание ключевых целей предметной области и ее специфики позволяет архитектору определить необходимые архитектурные свойства.

Совет: обсуждая со стейкхолдерами ключевые свойства архитектуры, стремитесь к тому, чтобы список получился максимально коротким. В архитектуре весьма распространен антипаттерн, заключающийся в попытке разработать *универсальную архитектуру*, поддерживающую *все* архитектурные свойства. С каждым новым архитектурным свойством усложняется дизайн всей системы,

а поддержка чрезмерного количества свойств приводит к постоянному наращиванию сложности еще до практической реализации самих задач. Необходимо стараться сохранять простоту структуры, не зацикливаясь на количестве свойств.

КОНКРЕТНЫЙ ПРИМЕР: VASA

Весьма поучительным примером того, как чрезмерная детализация архитектурных свойств в конечном итоге убивает проект, может послужить история корабля Vasa. Речь идет о шведском боевом судне, построенном в период с 1626 по 1628 год по приказу короля, желавшего получить самый совершенный из всех когда-либо существовавших кораблей. До этого корабли предназначались либо для перевозки войск, либо для ведения морского боя, а судно Vasa должно было стать и тем и другим. Большинство кораблей были однопалубными, а Vasa — двухпалубным. Пушек было вдвое больше, чем на других подобных кораблях. Несмотря на опасения опытных кораблестроителей (не смеющих перечить королю Адольфу), судно все же было построено. И чтобы отпраздновать это, корабль вошел в гавань и отсалютовал залпом из пушек с одного из бортов. К несчастью, из-за неустойчивости судно перевернулось и затонуло, погрузившись на дно залива. В начале XX века корабль был поднят спасателями и теперь является экспонатом одного из музеев Стокгольма.

Многие архитекторы и стейкхолдеры предметной области стремятся расставить приоритеты в окончательном списке архитектурных свойств приложения или системы. Идея, может быть, и заманчивая, но в большинстве случаев неразумная и связанная не только с пустой тратой времени, но и с массой ненужных разочарований и разногласий среди ключевых стейкхолдеров. К общему согласию стороны вряд ли придут. Лучше попросите стейкхолдеров выбрать из окончательного списка три самых главных свойства (в произвольном порядке). Это не только существенно упростит достижение консенсуса, но и позволит сфокусироваться на обсуждении того, что действительно важно, помогая тем самым архитектору провести анализ компромиссов при принятии архитектурных решений.

Основная часть архитектурных свойств выбирается после переговоров, во время которых у стейкхолдеров предметной области уточняются ключевые требования. Казалось бы, здесь все просто, но проблема в том, что архитектор и заказчик разговаривают на разных языках. Архитектор использует такие понятия, как *масштабируемость*, *интероперабельность*, *отказоустойчивость*, *изучаемость* и *доступность*. А заказчик оперирует понятиями *слияние* и *поглощение*, *удовлетворение запросов пользователей*, *сроки выпуска и конкурентные*

преимущества. Получаются «ошибки перевода», вызывающие взаимное недопонимание. Архитектор не может придумать архитектуру, позволяющую добиться удовлетворения всех запросов пользователей, а заказчик не понимает, почему в приложении уделяется так много внимания доступности, интероперабельности, изучаемости и отказоустойчивости. К счастью, все же существует перевод с языка бизнес-задач на язык архитектурных свойств. Некоторые наиболее распространенные задачи предметной области и соответствующие им свойства перечислены в табл. 5.1.

Таблица 5.1. Перевод задач предметной области в архитектурные свойства

Задача предметной области	Архитектурные свойства
Слияние и поглощение	Интероперабельность, масштабируемость, адаптируемость, расширяемость
Сроки выпуска	Гибкость, тестируемость, развертываемость
Удовлетворение запросов пользователей	Производительность, доступность, отказоустойчивость, тестируемость, развертываемость, гибкость, безопасность
Конкурентоспособность	Гибкость, тестируемость, развертываемость, масштабируемость, доступность, отказоустойчивость
Сроки и бюджет разработки	Простота, реализуемость

Важно отметить, что гибкость не является эквивалентом сроков выпуска на рынок. Для выдерживания намеченных сроков нужна сумма свойств: гибкость + + тестируемость + развертываемость. В эту ловушку недопонимания при переводе задач предметной области попадают многие архитекторы. Сосредоточиться только на одном свойстве — это как забыть добавить муку в тесто для торта. Например, заказчик может сказать: «Согласно нормативным требованиям нам совершенно необходимо вовремя завершать расчет стоимости фондов на конец дня». Слабый архитектор может сосредоточиться только на производительности, которая, по его представлениям, находится в центре внимания предметной задачи. Но по многим причинам он потерпит неудачу. Во-первых, если система недоступна в нужный момент, то абсолютно неважно, насколько быстро она работает. Во-вторых, по мере роста объемов задач предметной области и создания дополнительных фондов система также должна расширяться, чтобы успевать вовремя обработать данные. В-третьих, система должна быть не только доступна, но еще и надежна, чтобы в процессе расчета стоимости фондов не случился сбой. В-четвертых, а что произойдет, если на момент готовности расчета примерно на 85% система выйдет из строя? Она должна быть в состоянии восстановиться и запустить расчет стоимости с того самого места, на котором он остановился.

ПРОИСХОЖДЕНИЕ АРХИТЕКТУРНЫХ КАТА

Несколько лет назад известным архитектором Тедом Ньюардом (Ted Neward) были изобретены архитектурные ката — хитрый способ, позволяющий начинающим архитекторам практиковаться в определении требуемых архитектурных свойств на основе описаний предметной области. В японских и других восточных боевых искусствах *ката* — это индивидуальное тренировочное упражнение, в котором акцент делается на правильные формы и технику исполнения.

> Как создаются великие дизайнеры? Великие дизайнеры, разумеется, занимаются дизайном.
>
> *Фредерик Брукс*

Откуда возьмутся великие архитекторы, если они будут заниматься архитектурой едва ли пять-шесть раз за всю их карьеру?

Чтобы предоставить учебную программу для начинающих архитекторов, Тед создал первый сайт с архитектурными ката, адаптацией и обновлением которого занялись ваши авторы, Нил и Марк. Основная идея ката-упражнения в том, что архитекторы получали задачу в терминах предметной области, а также сопроводительный контекст (то, что могло не попасть в требования, но все же оказывало влияние на проектирование). Небольшие команды в течение 45 минут работали над проектом, а затем показывали результаты другим группам, которые голосовали за тех, кто, по их мнению, придумал самую удачную архитектуру. Согласно своему исходному предназначению, архитектурные ката служат ценной лабораторией для начинающих архитекторов.

В каждом ката имеются предопределенные разделы:

Описание

> Общая задача предметной области, для решения которой предназначается система.

Пользователи

> Ожидаемое количество и/или типы пользователей системы.

Требования

> Предметные или относящиеся к уровню предметной области требования, которые архитектор получает от пользователей или специалистов предметной области.

Несколько лет спустя Нил обновил формат своего блога, добавив в каждое ката раздел *дополнительного контекста* с важными вспомогательными факторами, придающими упражнениям более реалистичный вид.

Дополнительный контекст

Многие факторы, учитываемые архитектором, не фигурируют в требованиях в явном виде, но формируются на основе его знания неявных характеристик предметной области.

Мы настоятельно рекомендуем начинающим архитекторам воспользоваться этим сайтом для выполнения собственного ката-упражнения. Любой желающий может принести с собой все исходные данные, устроить тренинг по решению задачи для команды начинающих архитекторов и попросить опытного архитектора оценить проект и провести анализ компромиссных решений либо сразу на месте, либо потом, после краткого изучения результатов. Проект не будет сложным, поскольку упражнение ограничено по времени. В идеале весь состав команды получит комментарии от опытного архитектора, где будут указаны неучтенные компромиссы и альтернативные варианты проекта.

И наконец, система может быть быстродействующей, но будут ли стоимости фондов рассчитаны правильно? Получается, что в дополнение к производительности архитектору следует уделить внимание доступности, масштабируемости, надежности, восстанавливаемости и проверяемости.

Выбор архитектурных свойств на основе требований

Ряд архитектурных свойств берется из явно выраженных положений, содержащихся в требованиях. Например, в документах указывается ожидаемое количество пользователей и возможное масштабирование. Выбор свойств также основан на собственных представлениях архитектора, поэтому ему крайне необходимо понимание предметной области. Предположим, к примеру, что архитектор участвует в проектировании приложения для учета посещения занятий студентами университета. Для простоты предположим, что мы имеем дело с тысячей студентов и 10 учетными часами. Должен ли архитектор спроектировать систему без масштабируемости, допуская, что студенты будут приходить на занятия равномерным потоком? Или же, зная студенческие привычки,

он создаст систему, способную провести учет всей тысячи студентов за 10 минут до начала учебного часа? Ответ на этот вопрос очевиден всем, кто понимает, что многие студенты склонны все делать в последнюю минуту! Подобные особенности редко отмечаются в требованиях, но при этом оказывают существенное влияние на проектные решения.

Конкретный пример: Silicon Sandwiches

Для наглядного объяснения некоторых концепций мы воспользуемся *архитектурным ката* (см. врезку «Происхождение архитектурных ката» на с. 99). Чтобы показать, как архитекторы извлекают архитектурные свойства из требований, возьмем ката под названием Silicon Sandwiches.

Описание

Сеть магазинов сэндвичей хочет ввести систему онлайн-заказов (в дополнение к действующей системе заказов по телефону).

Пользователи

Тысячи; возможно, когда-то их число достигнет миллиона.

Требования

- Пользователи размещают заказ и получают время готовности сэндвича и маршрут, ведущий к магазину (маршрут строится на основе информации от нескольких картографических сервисов, включая данные о дорожном трафике).
- Если при заказе выбран вариант с доставкой, водитель с сэндвичем отправляется к пользователю.
- Доступ с мобильных устройств.
- Общенациональные ежедневные акции и специальные предложения.
- Локальные ежедневные акции и специальные предложения.
- Прием онлайн-платежей при заказе или при доставке.

Дополнительный контекст

- Сэндвич-магазины работают по франшизе, и у каждого свой собственный владелец.
- В ближайшем будущем головная компания планирует расширяться за рубеж.

- Цель корпорации — нанять недорогую рабочую силу для максимизации прибыли.

Какие архитектурные свойства можно извлечь из этого сценария? Каждая часть требований может подтолкнуть к определению одного или нескольких аспектов архитектуры (но не всегда). Здесь архитектор не проектирует всю систему: главные усилия в проекте по-прежнему должны прилагаться к созданию кода для решения задач предметной области. Для начала проводится анализ всего того, что воздействует или оказывает влияние на систему, особенно на ее структуру.

В первую очередь нужно разделить предполагаемые архитектурные свойства на явные и неявные.

Явные свойства

В спецификации требований явные архитектурные свойства определены как необходимая часть проекта. Например, интернет-магазин способен поддерживать определенное количество одновременно обращающихся к нему пользователей, которое аналитики предметной области указывают в требованиях. Архитектор должен рассмотреть влияние каждого требования на архитектурные свойства. Но сначала необходимо оценить ожидаемые показатели, представленные в ката в разделе «Пользователи».

В первую очередь архитектор должен уделить внимание количеству пользователей: пока речь идет о тысячах, но возможно, когда-то их число достигнет миллиона (у нас весьма амбициозный магазин сэндвичей!). Отсюда берется одно из ключевых свойств архитектуры — *масштабируемость* , то есть возможность одновременно обслужить большое количество пользователей без серьезного снижения производительности. Заметьте, что в постановке задачи требование масштабируемости не прописывается явно, но подразумевается ожидаемым числом пользователей. Архитекторам часто приходится «переводить» бизнес-требования на инженерно-технический язык.

Но нам, возможно, понадобится и *адаптируемость* — возможность обработки всплесков запросов. Зачастую эти свойства объединяют, но у них разные цели. Масштабируемость наглядно представлена на графике (рис. 5.1).

А адаптируемость, судя по графику на рис. 5.2, измеряется всплесками сетевого трафика.

Некоторые системы обладают масштабируемостью, но не отличаются особой адаптируемостью. Рассмотрим, к примеру, систему резервирования мест

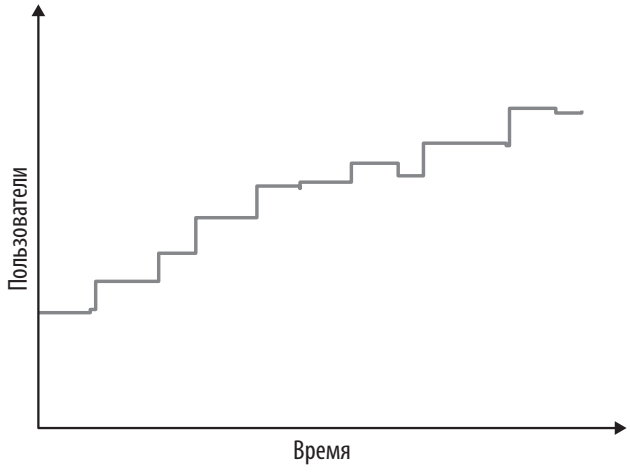


Рис. 5.1. Масштабируемость измеряется количеством одновременно обслуживаемых пользователей

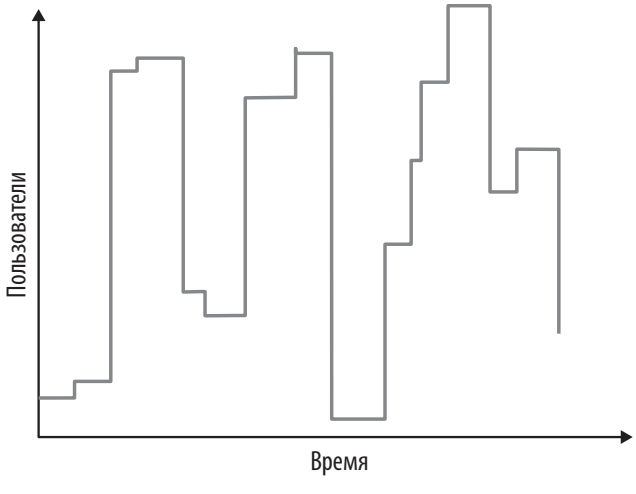


Рис. 5.2. Адаптируемые системы должны выдерживать всплески пользовательской активности

в отелях. Если не будет специальных распродаж или каких-либо мероприятий, количество пользователей, скорее всего, будет примерно постоянным. А теперь возьмем систему бронирования концертных билетов. Когда в продажу поступят билеты на новый концерт, сайт заполнится пылкими фанатами, вот тогда и будет востребована высокая степень адаптируемости. Зачастую адаптируемым системам также требуется масштабируемость: возможность обработки всплесков активности и большого числа одновременно обращающихся к ним пользователей.

Адаптируемость отсутствует в требованиях к Silicon Sandwiches, но архитектор должен учесть ее как важный архитектурный аспект. Иногда архитектурные свойства указываются в требованиях напрямую, но некоторые из них подразумеваются в формулировке задачи предметной области. Рассмотрим магазин сэндвичей. Является ли постоянным его сетевой трафик в течение дня? Или ему приходится справляться со всплесками трафика в обеденный перерыв? На последний вопрос почти всегда следует утвердительный ответ. Таким образом, грамотный архитектор должен выявить это потенциальное архитектурное свойство.

Для выявления необходимых архитектурных свойств архитектор должен тщательно проанализировать каждое из следующих бизнес-требований:

1. Пользователи размещают заказ, а затем получают время готовности сэндвича и маршрут к магазину (который должен составить единое целое с информацией от ряда внешних картографических служб, включая данные по загруженности трасс).

   У внешних картографических служб предполагается наличие точек интеграции, которые могут повлиять на надежность. Например, если разработчик создает систему, полагающуюся на сторонний сервис, а ее обращение к этому сервису терпит неудачу, то это оказывает влияние на надежность вызывающей системы. Но при этом архитекторы должны избегать чрезмерного уточнения архитектурных свойств. Что произойдет, если даст сбой внешний сервис интенсивности дорожного движения? Выйдет ли сайт Silicon Sandwiches из строя или же он просто будет выдавать информацию без данных об интенсивности трафика? Архитекторы всегда должны остерегаться излишней уязвимости или хрупкости дизайна.

2. Если при заказе выбран вариант с доставкой, водитель с сэндвичем отправляется к пользователю.

   Похоже, для поддержки этого требования никакого особого архитектурного свойства не требуется.

3. Доступ с мобильных устройств.

   Это требование в первую очередь повлияет на *дизайн* приложения, указывая на то, что оно должно стать либо кросс-платформенным веб-приложением, либо несколькими нативными веб-приложениями. Исходя из бюджетных ограничений и желаемой простоты приложения, архитектор, вероятно, сочтет излишним создание нескольких приложений и решит, что нужно спроектировать веб-приложение, оптимизированное для мобильных устройств. В таком случае ему может потребоваться определение ряда особых архитектурных свойств, влияющих на производительность и скорость загрузки страницы, а также некоторых других свойств, важных для мобильных устройств. Учтите, что архитектор в подобных ситуациях должен действовать не в одиночку,

а совместно с дизайнерами пользовательского интерфейса, стейкхолдерами предметной области и другими смежными специалистами.

4. Общенациональные ежедневные акции и специальные предложения.
5. Локальные ежедневные акции и специальные предложения.

   Из этих требований следует наличие настраиваемых рекламных акций и специальных предложений. Надо учесть, что первое требование также подразумевает обновление информации об интенсивности трафика в зависимости от адреса доставки. Основываясь на всех трех изложенных требованиях, архитектор может рассмотреть такое архитектурное свойство, как настраиваемость. Например, настраиваемое поведение весьма успешно поддерживается архитектурным стилем микроядер, где нужный результат достигается за счет определения архитектуры подключаемого модуля (плагина). В таком случае исходное поведение закладывается в ядре, а разработчики создают с помощью плагинов дополнительные настраиваемые части, основанные на географической привязке. Но данное требование может быть удовлетворено и в традиционном дизайне с помощью паттернов проектирования, например Шаблонного метода (Template Method). Такие дилеммы встречаются довольно часто и требуют от архитекторов постоянно сравнивать альтернативные варианты в поисках компромисса. Более подробно такой компромисс будет рассмотрен ниже во врезке «Компромиссы: дизайн или архитектура» на с. 107.
6. Прием онлайн-платежей при заказе или при доставке.

   Онлайн-платежи подразумевают применение мер безопасности, но в этом требовании нет конкретного указания уровня защиты и тем самым допускается использование стандартного уровня.
7. Сэндвич-магазины работают по франшизе, и у каждого свой собственный владелец.

   Это требование может накладывать ограничения на затраты по созданию архитектуры. Архитектор должен проверить реализуемость (учитывая ограничения по стоимости, срокам и навыкам персонала), чтобы понять, какая из архитектур окажется наиболее подходящей: упрощенная или посложнее.
8. В ближайшем будущем головная компания планирует расширение за рубеж.

   Это требование подразумевает применение *интернационализации (i18n)*. Для его удовлетворения существует масса методов проектирования, не требующих особой структуры. Но это, безусловно, означает, что нужны определенные проектные решения.
9. Цель корпорации — нанять недорогую рабочую силу для максимизации прибыли.

Это требование предполагает отведение важной роли удобству и простоте использования, но опять же, это больше относится к дизайну, нежели к архитектурным свойствам.

Третьим свойством архитектуры, выведенным из предыдущих требований, является *производительность*: вряд ли клиентам захочется заказывать сэндвичи в магазине, имеющем низкую производительность, особенно в часы пик. Но *производительность* — понятие растяжимое: какую именно производительность должен задать архитектор? Различные нюансы производительности будут рассмотрены в главе 6.

Желательно также определить показатели производительности в совокупности с пределами масштабируемости. Иными словами, нужно установить базовый уровень производительности без конкретного масштаба, а также определить приемлемый уровень производительности для конкретного числа пользователей. Взаимодействие свойств архитектуры встречается довольно часто, заставляя архитекторов определять показатели с учетом их взаимосвязанности.

Неявные свойства

Многие архитектурные свойства не отражаются в технических требованиях, но при этом составляют важный аспект дизайна. Одним из неявных архитектурных свойств, которое может пригодиться системе, является *доступность*: она дает гарантию беспрепятственного доступа пользователей к сайту. С доступностью довольно тесно соприкасается *надежность*: гарантия того, что сайт останется активным в процессе работы с ним, ведь вряд ли кому-то захочется делать заказ на сайте, постоянно разрывающем соединение, заставляя клиентов снова и снова входить в систему.

Безопасность является неявной характеристикой каждой системы: никому не нужны небезопасные программные продукты. Тем не менее она может стать приоритетной в зависимости от степени важности системы, а также от взаимосвязанности других свойств. Архитектор считает безопасность архитектурным свойством, если она оказывает влияние на ряд структурных аспектов дизайна или играет решающую либо особо важную роль для приложения.

Для нашего приложения Silicon Sandwiches архитектор может допустить, что платежи обрабатываются третьей стороной. То есть до тех пор пока разработчики следуют общим правилам соблюдения мер безопасности (не передают номера кредитных карт простым текстом, не хранят слишком большой объем информации и т. д.), архитектору для соблюдения мер безопасности не понадобится никакой специальной структуры: достаточно будет качественных проектных решений

КОМПРОМИССЫ: ДИЗАЙН ИЛИ АРХИТЕКТУРА

В ката Silicon Sandwiches архитектор, наверное, определил бы настраиваемость как часть системы, но возникает вопрос, к чему она относится: к архитектуре или к дизайну? Архитектура подразумевает некий структурный компонент, а дизайн выстраивается внутри архитектуры. В случае с настраиваемостью в Silicon Sandwiches архитектор может выбрать архитектурный стиль, например микроядро, и выстроить структурную поддержку настраиваемости. Но если архитектор из-за конкурирующих задач выберет другой стиль, разработчики смогут реализовать настраиваемость, воспользовавшись паттерном Шаблонный метод, позволяющим родительским классам определять последовательность выполняемых действий, который можно переопределить в дочерних классах. Какой дизайн лучше?

Как и все в архитектуре, это зависит от ряда факторов. Во-первых, имеются ли веские причины, такие как производительность или связанность, препятствующие реализации архитектуры микроядра? Во-вторых, будет ли реализация других необходимых архитектурных свойств в одном дизайне сложнее, чем в другом? В-третьих, во что обойдется поддержка всех архитектурных свойств в каждом дизайне по сравнению с паттерном? Этот тип анализа архитектурных компромиссов составляет важную часть работы архитектора.

Для архитектора очень важно сотрудничество с разработчиками, руководителями проекта, командой сопровождения и другими участниками создания программной системы. Никакие архитектурные решения не должны приниматься в отрыве от команды специалистов по реализации проекта, поскольку это приведет к зловещему антипаттерну *Башня из слоновой кости* (*Ivory Tower Architect*). Что же касается проекта Silicon Sandwiches, для принятия решения об оптимальной реализации настраиваемости необходимо тесное сотрудничество архитектора, технического руководителя, разработчиков и аналитиков предметной области.

в самом приложении. Каждое архитектурное свойство взаимодействует с другими, расставляя перед архитекторами западню чрезмерного количества свойств архитектуры, действующую не менее разрушительно, чем упущения в их выявлении, поскольку это приводит к необоснованному усложнению дизайна системы.

Последним основным архитектурным свойством проекта Silicon Sandwiches является *настраиваемость*, которая вытекает сразу из нескольких пунктов требований. Заметьте, что настраиваемое поведение подразумевается многими

составляющими предметной области: это, например, рецепты, местные продажи и маршруты. Поэтому архитектура должна поддерживать настройку пользовательского поведения. Обычно такая задача входит в дизайн приложения. Но, согласно нашему определению, часть задач предметной области, требующих для своего решения настраиваемой структуры, переходит в область архитектурных свойств. При этом данный элемент дизайна не играет решающей роли в успехе приложения. Важно отметить, что правильных ответов на вопросы о выборе свойств архитектуры не существует, есть только неправильные, или же, как уже было отмечено Марком в одной из его известных цитат:

> В архитектуре нет неправильных ответов, есть только стоящие больших затрат.

Архитектор может разработать архитектуру, не учитывая настраиваемость в структуре, что потребует поддержки этого поведения от дизайна приложения (см. выше врезку «Компромиссы: дизайн или архитектура»). Архитекторы не должны стремиться к созданию абсолютно безупречного набора архитектурных свойств — разработчики в состоянии реализовать нужную функциональность различными способами. Но грамотное определение наиболее важных структурных элементов может облегчить разработку более простого и элегантного дизайна. Следует помнить: в архитектуре нет наилучшего дизайна, есть только приемлемый (наименее худший) набор компромиссов.

Архитекторы также должны уметь расставлять приоритеты, чтобы найти минимально требуемый набор архитектурных свойств. Полезным упражнением для команды после первого шага на пути определения архитектурных свойств может стать выявление наименее важного свойства: если что-то исключать, то что именно? Как правило, архитекторы исключают какое-то из явно выраженных архитектурных свойств, поскольку многие из неявных свойств обеспечивают общий успех. Определение того, что является решающим или весьма важным, помогает архитекторам понять, действительно ли приложение нуждается в каждом архитектурном свойстве. Выявляя наименее значимое свойство, архитекторы могут облегчить себе задачу определения критических потребностей. Возвращаясь к Silicon Sandwiches: какое же из архитектурных свойств мы назовем наименее важным? Опять же, абсолютно правильного ответа нет. Решение может быть в потере либо настраиваемости, либо производительности. Из архитектурных свойств можно исключить настраиваемость и реализовать ее при разработке самого приложения. В составе эксплуатационных архитектурных свойств производительность, похоже, является наименее влиятельным. Разумеется, разработчики не собираются создавать приложение с крайне низкой производительностью, но все же уделяют ей меньше внимания, чем масштабируемости или доступности.

ГЛАВА 6

Измерение параметров архитектурных свойств и управление их соблюдением

Архитекторам приходится работать с чрезвычайно широким диапазоном архитектурных свойств. Эксплуатационные аспекты, в числе которых производительность, адаптируемость и масштабируемость, сочетаются со структурными вопросами, такими как модульность и развертываемость. В этой главе основное внимание будет уделено более точному определению ряда распространенных архитектурных свойств и созданию механизма контроля за их соблюдением.

Измерение параметров архитектурных свойств

При определении архитектурных свойств в организациях возникает ряд общих проблем:

Они не физического плана

Многие общеупотребительные архитектурные свойства не имеют конкретных значений. Как, например, архитектор проектирует *гибкость* или *развертываемость*? В сфере разработки ПО существует самое разнообразное понимание общепринятых терминов. Иногда это зависит от контекста, а иногда различия просто произвольны.

У них совершенно разные определения

Даже внутри одной организации разные отделы могут не соглашаться с определениями таких решающих свойств, как *производительность*. Пока

разработчики, архитекторы и эксплуатационники не выработают совместно однозначное определение, понять друг друга будет нелегко.

Они слишком сложные

Многие желаемые архитектурные свойства состоят из других, менее масштабных. Например, разработчики могут разложить гибкость на такие свойства, как модульность, развертываемость и тестируемость.

Все три проблемы решаются объективными определениями архитектурных свойств. Путем обсуждения определений и концепций всеми участниками процесса в организации должен создаваться общий архитектурный язык. Кроме того, благодаря точным определениям специалисты могут разложить сложные свойства на измеримые составляющие и проводить с ними дальнейшую работу.

Эксплуатационные показатели

Для многих архитектурных свойств, например производительности или масштабируемости, существуют вполне определенные методы измерения. Но даже они имеют множество интерпретаций в зависимости от целей команды. Например, команда может измерять среднее время ответа на конкретные запросы, что является хорошим примером метрики эксплуатационных свойств архитектуры. Но если команды выполняют замер только средних показателей, что будет, когда при некоторых граничных условиях 1% запросов займет в 10 раз больше времени, чем обычно? Если у сайта достаточно большой трафик, то такие экстремальные отклонения пройдут незамеченными. Чтобы отловить такое резкое увеличение потока, может также потребоваться замер максимального времени отклика.

Профессиональные команды не просто устанавливают жестко заданные количественные показатели производительности; они учитывают результаты статистического анализа. Предположим, к примеру, что сервису потокового видео необходимо управление масштабируемостью. Вместо установки некоего произвольного числа в качестве цели специалисты оценивают масштаб с течением времени и выстраивают статистические модели, а затем бьют тревогу, если показатели в реальном времени выходят за рамки спрогнозированных моделей. Отказ может означать две вещи: модель некорректна (команды хотели бы знать, в чем именно) или же что-то пошло не так (команды также хотели бы знать, что именно).

Количество свойств, доступных на данный момент для измерений, быстро растет, а с ними заодно развиваются инструменты и понимание нюансов. Например, многие команды в последнее время сосредоточились на таких метриках производительности, как *первое существенное отображение (first contentful paint)* и *первое бездействие процессора (first CPU idle)*, которые отражают проблемы

производительности веб-страниц у пользователей на мобильных устройствах. По мере изменения устройств, целей, возможностей и многих других вещей команды находят все новые и новые параметры и способы их измерения.

МНОГООБРАЗИЕ ФОРМ ПРОИЗВОДИТЕЛЬНОСТИ

У ряда описываемых нами архитектурных свойств есть множество определений, имеющих едва уловимые отличия. Один из лучших примеров — производительность. Во многих проектах рассматривается общая производительность: например, сколько времени в веб-приложении занимают циклы запрос — ответ. Но архитекторы и DevOps-специалисты выполнили большой объем работы для определения более точных показателей производительности, выделив конкретные метрики для тех или иных частей приложения. Например, многие организации провели исследования пользовательского поведения и выяснили, что оптимальное время загрузки первой страницы (первый очевидный признак активности веб-страницы в браузере или на мобильном устройстве) — 500 мс, то есть полсекунды. Большинство приложений попадают в двузначный диапазон данного показателя. Но для современных сайтов, стремящихся привлечь как можно больше посетителей, это важный показатель, и организации, владеющие такими сайтами, выработали весьма детализированные метрики.

Некоторые из этих метрик играют дополнительную роль при проектировании приложений. Многие организации применяют к скачиваниям страниц *бюджеты, измеряемые в килобайтах (K-weight budgets*), представляющие собой максимальный выраженный в байтах объем библиотек и сред, допустимый на той или иной странице. Это связано с физическим ограничением: только такое количество байтов может одновременно пройти по сети, что особенно важно для мобильных устройств в регионах с большим временем ожидания.

Структурные показатели

Некоторые объективные показатели не столь очевидны, как производительность. Что можно сказать о внутренних структурных свойствах, например о модульности? К сожалению, комплексных показателей внутреннего качества кода пока не существует. Но некоторые показатели и хорошо известные инструменты позволяют архитекторам выявлять критические моменты в структуре кода, хотя и в довольно узких рамках.

Вполне очевидной измеряемой характеристикой кода является сложность, определяемая показателем *цикломатической сложности*.

ЦИКЛОМАТИЧЕСКАЯ СЛОЖНОСТЬ

Цикломатическая сложность (Cyclomatic Complexity, CC) — это показатель на уровне кода, разработанный Томасом Маккейбом (Thomas McCabe) в 1976 году для измерения сложности кода на уровне функции/метода, класса или приложения. Она вычисляется путем применения к коду теории графов, в частности, к точкам принятия решений, определяющим различные пути исполнения. Например, если функция не имеет команд принятия решений (таких, как оператор `if`), то $CC = 1$. Если у функции имеется одно условие, то $CC = 2$, поскольку существуют два возможных пути исполнения.

Формула для вычисления CC для отдельно взятой функции или метода имеет вид $CC = E - N + 2$, где N представляет *узлы* (строки кода), а E — *ребра* (возможные решения). Рассмотрим C-подобный код, показанный в примере 6.1.

Пример 6.1. Пример кода для вычисления цикломатической сложности

```
public void decision(int c1, int c2) {
    if (c1 < 100)
        return 0;
    else if (c1 + C2 > 500)
        return 1;
    else
        return -1;
}
```

Цикломатическая сложность для примера 6.1 равна 3 (=5 – 4 + 2); соответствующий граф показан на рис. 6.1.

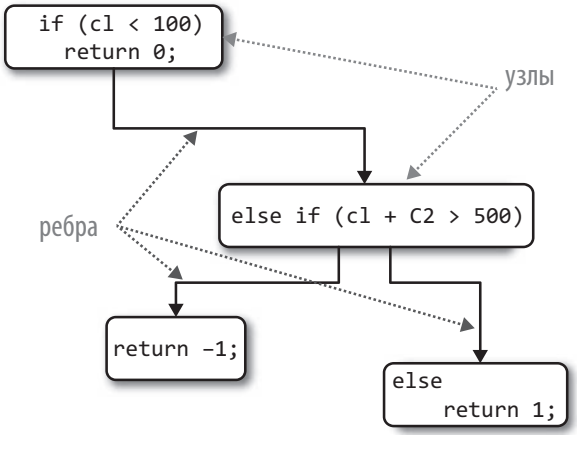


Рис. 6.1. Цикломатическая сложность для функции decision

Число 2 в формуле является упрощением для одной функции или метода. Для разветвляющихся вызовов других методов (известных в теории графов как *компоненты связности*) используется более универсальная формула $CC = E - N + 2P$, где P представляет количество компонентов связности.

Архитекторы и разработчики единодушны в том, что чрезмерно сложный код является «кодом с запашком»; он вредит практически всем желательным свойствам кодовой базы: модульности, тестируемости, развертываемости и т. д. Если команды не будут отслеживать нарастание сложности, она неизбежно появится в кодовой базе.

Показатели процесса

Некоторые архитектурные свойства связаны с процессами разработки ПО. Например, часто в качестве желаемого свойства фигурирует гибкость. Но она является составным архитектурным свойством, которое архитектор может разбить на тестируемость и развертываемость.

Тестируемость можно измерить с помощью инструментов покрытия кода практически для всех платформ, оценивающих полноту тестирования. Как и в случае с любой другой проверкой ПО, она не заменит анализ и разработку плана. Например, у кодовой базы может быть 100%-ное покрытие кода, но плохая логика, не вселяющая ни малейшей уверенности в правильности кода. Но тестируемость, несомненно, является объективно измеряемым свойством. Также с помощью различных показателей можно оценить развертываемость: определить процентное соотношение удачных и неудачных развертываний, засечь продолжительность развертывания, подсчитать количество вопросов и недочетов, всплывающих в процессе развертывания, и т. д. Каждая команда несет ответственность за проведение достаточного количества измерений, чтобы обеспечить организацию всей необходимой информацией, как в качественном, так и в количественном отношении. Многие из этих показателей зависят от приоритетов и целей команды.

Гибкость и ее составляющие имеют явное отношение к процессу разработки ПО. Но этот процесс может повлиять на структуру архитектуры. Например, если простота развертывания и тестируемость входят в круг первоочередных задач, то архитектор будет уделять больше внимания высокой модульности и изолированности на уровне архитектуры, и таким образом архитектурное свойство повлияет на структурное решение. Если удастся соответствовать трем нашим параметрам, то до уровня архитектурного свойства может подняться практически все, что находится в рамках программного проекта.

КАКОЕ ЗНАЧЕНИЕ ЦИКЛОМАТИЧЕСКОЙ СЛОЖНОСТИ МОЖНО СЧИТАТЬ ПРИЕМЛЕМЫМ?

В разговорах на данную тему авторам часто задают один и тот же вопрос: какое значение CC можно считать приемлемым? Разумеется, здесь неизменно дается традиционный для архитектуры ПО ответ: все зависит от обстоятельств! Если конкретнее, то от сложности решаемой задачи предметной области. Например, если дело касается алгоритмически сложной задачи, в решении будут сложные функции.

Архитекторам нужно отслеживать ряд ключевых аспектов CC: вызвана ли сложность функций задачами предметной области или же она является следствием слабого кода? Не имеем ли мы дело со слабой фрагментацией кода? Иными словами, можно ли разбить крупный метод на более мелкие логические фрагменты, распределив работу (и сложность) между более продуманными методами?

В целом же в сфере создания программных продуктов пороговыми приемлемыми показателями для CC считаются любые значения меньше 10, без учета других соображений, например сложных предметных областей. Мы считаем этот порог слишком высоким и предпочли бы код с показателем меньше 5, что указывало бы на связность и продуманность кода. Измерительный инструмент из мира языка Java, Crap4J[1], оценивает, насколько плохим (crappy) является ваш код, вычисляя комбинацию из CC и из покрытия кода тестами: если CC возрастает до 50, то никакой объем покрытия кода не спасет программу. Самым страшным профессиональным артефактом, с которым когда-либо сталкивался Нил, была отдельно взятая функция на языке C, которая служила основой коммерческого программного пакета, чей показатель CC перевалил за 800! У этой функции было 4000 строк кода со свободным использованием `GOTO` (чтобы избежать невероятно глубоко вложенных циклов).

Практики проектирования, подобные разработке через тестирование (test-driven development, TDD), обладают непреднамеренным (но полезным) побочным эффектом: они создают для решения конкретной задачи предметной области небольшие по объему и несложные методы. Применяя TDD, разработчики стараются создать простой тест для наименьшего объема кода. Пошаговый характер разработки при удачно выбранных границах тестирования способствует написанию добротных, обладающих высокой связностью методов, показывающих низкие значения CC.

[1] http://www.crap4j.org/

Управление и функции пригодности

Каким образом после определения свойств архитектуры и расстановки приоритетов архитекторы могут быть уверены, что разработчики будут соблюдать их рекомендации? Например: модульность является весьма важным свойством, но не таким уж срочным, а поскольку преобладающий фактор для многих программных проектов — срочность, архитекторам нужен некий механизм контроля.

Управление архитектурными свойствами

Управление является важной обязанностью архитектора. Это касается всей сферы разработки программного обеспечения, поскольку архитекторы (включая и архитекторов всего предприятия) должны оказывать влияние на разнообразные аспекты создания ПО. Например, обеспечение качества ПО в организации должно быть под управлением архитектора, потому что это относится к области архитектуры, а пренебрежение этим может привести к катастрофическим проблемам качества.

К счастью, есть комплексные решения, помогающие архитекторам, что является хорошим примером постепенного роста возможностей в экосистеме разработки ПО. Стремление к автоматизации программных проектов, порожденное методологией экстремального программирования (Extreme Programming[1]), привело к непрерывной интеграции, что, в свою очередь, привело к дальнейшей автоматизации в области эксплуатации программных продуктов (методология DevOps), которая переходит уже к архитектурному управлению. В книге «Building Evolutionary Architectures» (издательство O'Reilly)[2] описывается целое семейство технологий, называемых функциями пригодности, которые применяются для автоматизации архитектурного управления.

Функции пригодности

Слово «evolutionary» («эволюционный») в названии книги «*Building Evolutionary Architectures*» происходит от эволюционных вычислений, а не от биологии. Один из авторов, доктор Ребекка Парсонс, какое-то время провела в эволюционном вычислительном пространстве, где, в частности, оперировала

[1] http://www.extremeprogramming.org/

[2] Нил Форд, Ребекка Парсонс, Патрик Куа. «Эволюционная архитектура. Поддержка непрерывных изменений». СПб., издательство «Питер».

такими инструментами, как генетические алгоритмы. Генетический алгоритм выполняется, выдает результат, а затем мутирует с помощью известных методов эволюционных вычислений. Если разработчик пытается спроектировать генетический алгоритм для получения какого-либо полезного результата, то ему нужно управлять алгоритмом, предоставляя ему объективные показатели качества конечного результата. Этот управляющий механизм называется *функцией пригодности* и представляет собой объектную функцию, используемую для оценки близости результата к достижению цели. Предположим, к примеру, что разработчику надо решить задачу коммивояжера, знаменитую задачу, используемую в качестве основы для машинного обучения. Есть продавец и список городов для посещения с указанием расстояний между ними; каким будет оптимальный маршрут? Если разработчик для решения данной задачи спроектирует генетический алгоритм, одна из функций пригодности может вычислять протяженность маршрута, при этом самый короткий возможный маршрут будет представлять собой достигнутую цель. Еще одна функция пригодности может заниматься оценкой общих затрат, связанных с маршрутом, и попыткой свести их к минимуму. Еще одна такая функция может вычислять время, когда коммивояжер находится в пути, и оптимизировать его, чтобы сократить общее время путешествия.

Эта концепция была позаимствована для создания *архитектурной функции пригодности*:

> *Архитектурная функция пригодности*
>
> Любой механизм, предоставляющий объективную оценку целостности какого-либо архитектурного свойства или сочетания архитектурных свойств.

Функции пригодности не являются каким-то новым фреймворком, который надо загрузить, скорее это новый взгляд на многие существующие инструменты. Обратите внимание на использованное в определении словосочетание *«любой механизм»*: методы проверки соблюдения свойств архитектуры не менее разнообразны, чем сами свойства. Функции пригодности в зависимости от способа их использования (в качестве показателей, обработчиков событий, библиотек юнит-тестирования, хаос-инжиниринга и т. д.) перекрывают многие существующие механизмы проверки (рис. 6.2).

Для реализации функций пригодности можно использовать весьма широкий спектр инструментов в зависимости от требуемых свойств архитектуры. Например, в разделе «Связанность» на с. 71 были представлены метрики, позволяющие архитектору оценить модульность. Рассмотрим несколько примеров функций пригодности, которые проверяют различные аспекты модульности.

Рис. 6.2. Механизмы функций пригодности

Циклические зависимости

Модульность является неявным архитектурным свойством, которому архитекторы должны уделять пристальное внимание, поскольку слабо поддерживаемая модульность вредит структуре кодовой базы. Однако есть силы, противодействующие благим намерениям архитектора. Например, вот что происходит при программировании в любой популярной среде разработки, относящейся к Java- или .NET-технологии. Как только разработчик ссылается на еще не импортированный класс, IDE-среда любезно открывает диалоговое окно, в котором разработчику задается вопрос, не хочет ли он автоматически импортировать ссылку. Это происходит настолько часто, что у большинства программистов вырабатывается привычка рефлекторно закрывать диалог автоматического импорта. Но произвольный импорт классов или компонентов между блоками кода означает катастрофическую ситуацию с модульностью. Например, на рис. 6.3 показан крайне разрушительный антипаттерн, применения которого архитекторы стремятся избегать.

Каждый компонент, показанный на рис. 6.3, ссылается на что-нибудь, находящееся в других компонентах. Наличие подобной сети компонентов разрушает модульность, поскольку разработчик не может повторно использовать отдельно взятый компонент без привлечения других компонентов. И конечно же, если эти другие компоненты связаны еще с какими-то компонентами, архитектура все больше будет скатываться к антипаттерну Большой ком грязи (Big Ball of Mud). Как архитекторы могут управлять этим процессом, не заглядывая через плечо программисту? Проверка кода, конечно, помогает, но она проводится в цикле разработки слишком поздно, теряя всякую эффективность. Если архитектор

позволяет команде разработчиков в течение недели до проверки бесконтрольно практиковать импорт внутри кодовой базы, то этой базе уже будет нанесен серьезный ущерб.

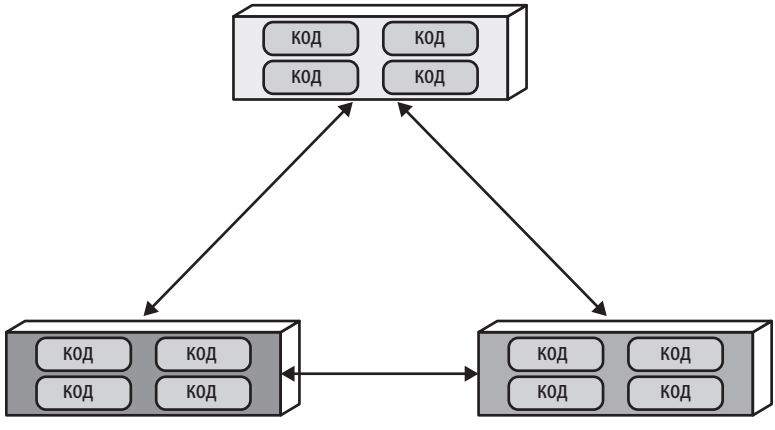


Рис. 6.3. Циклические зависимости между компонентами

Решением проблемы может стать создание функции пригодности, показанной в примере 6.2, которая будет следить за циклами.

Пример 6.2. Функция пригодности для обнаружения циклических зависимостей компонентов

```
public class CycleTest {
    private JDepend jdepend;

    @BeforeEach
    void init() {
        jdepend = new JDepend();
        jdepend.addDirectory("/path/to/project/persistence/classes");
        jdepend.addDirectory("/path/to/project/web/classes");
        jdepend.addDirectory("/path/to/project/thirdpartyjars");
    }

    @Test
    void testAllPackages() {
        Collection packages = jdepend.analyze();
        assertEquals("Cycles exist", false, jdepend.containsCycles());
    }
}
```

В представленном коде для проверки зависимостей между пакетами архитектор использует метрики JDepend[1]. Этот инструмент анализирует структуру Java-

[1] https://oreil.ly/ozzzk

пакетов и проваливает тест при наличии какого-либо цикла. Архитектор может подключить такой тест к непрерывной сборке проекта и перестать волноваться о случайном создании циклов нерадивыми разработчиками. Это отличный пример функции пригодности, контролирующей важные, но несрочные методы разработки ПО: на повседневное кодирование она оказывает весьма незначительное влияние, однако ее роль для архитекторов трудно переоценить.

Расстояние от главной последовательности

В разделе «Связанность» на с. 71 мы обсуждали более редкий показатель: расстояние от главной последовательности. Его оценка также может быть произведена с помощью функции пригодности, показанной в примере 6.3.

Пример 6.3. Расстояние от главной последовательности, определяемое функцией пригодности

```
@Test
void AllPackages() {
    double ideal = 0.0;
    double tolerance = 0.5; // project-dependent
    Collection packages = jdepend.analyze();
    Iterator iter = packages.iterator();
    while (iter.hasNext()) {
        JavaPackage p = (JavaPackage)iter.next();
        assertEquals("Distance exceeded: " + p.getName(),
            ideal, p.distance(), tolerance);
    }
}
```

В этом коде архитектор использует инструмент JDepend для задания приемлемых пороговых значений. При выходе какого-либо класса за их пределы функция пригодности проваливает тест.

Это пример не только объективного измерения архитектурных свойств, но и важности сотрудничества разработчиков и архитекторов при проектировании и реализации функций пригодности. Архитекторы не должны уединяться в башне из слоновой кости и создавать загадочные функции пригодности, которые разработчики не смогут понять.

Прежде чем навязывать разработчикам функцию пригодности, архитекторы должны убедиться, что разработчики понимают, в чем заключается ее цель.

За последние несколько лет функции пригодности постоянно совершенствовались, и в них включались специфические инструменты. Одним из них является

ArchUnit[1] — фреймворк тестирования Java, созданный под впечатлением экосистемы JUnit[2] и использующий несколько ее составляющих. В ArchUnit имеется множество предопределенных правил управления, запрограммированных в виде юнит-тестов и позволяющих архитекторам создавать специализированные тесты для проверки модульности. Рассмотрим многоуровневую архитектуру, показанную на рис. 6.4.

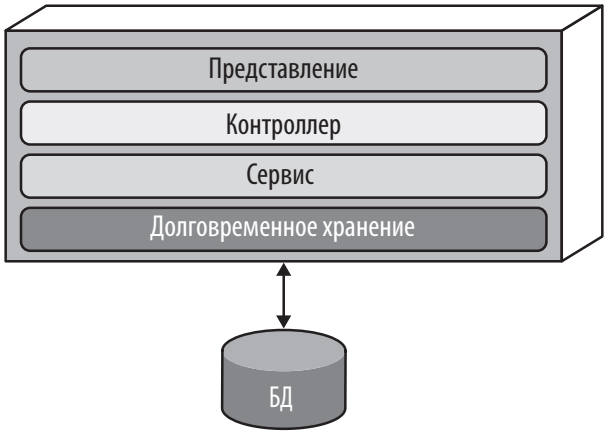


Рис. 6.4. Многоуровневая архитектура

При проектировании многоуровневого монолита вроде того, что показан на рис. 6.4, архитектор вполне обоснованно определяет уровни (мотивация, компромиссы и другие особенности многоуровневой архитектуры рассматриваются в главе 10). Но как архитектор сможет гарантировать, что разработчики будут соблюдать эти уровни? Одни не поймут важность паттернов, а другие могут придерживаться принципа «Лучше попросить прощения, чем вымаливать разрешение», что связано, например, с приоритетом производительности. Но если позволить разработчикам разрушать замыслы, заложенные в архитектуру, это повредит ее здоровью в долгосрочной перспективе.

ArchUnit позволяет архитекторам решать эту задачу с помощью функции пригодности, показанной в примере 6.4.

В примере 6.4 архитектор задает желаемое взаимодействие уровней и создает проверочную функцию пригодности для контроля за соблюдением своего замысла.

---

[1] https://www.archunit.org/

[2] https://junit.org/

Пример 6.4. Функция пригодности, использующая ArchUnit для управления уровнями

```
layeredArchitecture()
    .layer("Controller").definedBy("..controller..")
    .layer("Service").definedBy("..service..")
    .layer("Persistence").definedBy("..persistence..")
    .whereLayer("Controller").mayNotBeAccessedByAnyLayer()
    .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")
    .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service")
```

Аналогичный инструмент в .NET-пространстве, NetArchTest, позволяет проводить такие же тесты; проверка уровней на языке C# показана в примере 6.5.

Пример 6.5. NetArchTest для проверки зависимости уровней

```
// Классы, находящиеся в представлении, не должны напрямую обращаться
к хранилищам данных
var result = Types.InCurrentDomain()
    .That()
    .ResideInNamespace("NetArchTest.SampleLibrary.Presentation")
    .ShouldNot()
    .HaveDependencyOn("NetArchTest.SampleLibrary.Data")
    .GetResult()
    .IsSuccessful;
```

Еще один пример функции пригодности — созданное компанией Netflix инструментальное средство Chaos Monkey (обезьяна хаоса), входящее в пакет Simian Army[1] (армии обезьян). В частности, там имеются Conformity Monkey (обезьяна соответствия), Security Monkey (обезьяна безопасности) и Janitor Monkey (обезьяна-уборщик). Conformity Monkey позволяет архитекторам Netflix определять правила управления на производстве. Например, если архитекторы решили, что каждый сервис должен эффективно реагировать на все RESTful-команды, они встраивают соответствующую проверку в Conformity Monkey. Аналогично этому, Security Monkey ведет проверку каждого сервиса на наличие широко распространенных просчетов в системе безопасности вроде портов, которые не должны быть активными, и ошибок конфигурации. И наконец, Janitor Monkey ищет экземпляры, к которым больше уже не обращаются другие сервисы. В Netflix используется эволюционная архитектура, поэтому разработчики регулярно переходят на новые сервисы, оставляя старые сервисы запущенными без компаньонов. Поскольку на сервисы, запущенные в облаке, тратятся денежные средства, Janitor Monkey ищет ненужные сервисы и выводит их из эксплуатации.

[1] https://oreil.ly/GipHq

ПРОИСХОЖДЕНИЕ ОБЕЗЬЯНЬЕЙ АРМИИ

Когда компания Netflix решила перенести свои операции в облако Amazon, архитекторы забеспокоились, что они больше не смогут контролировать работу системы. А что, если дефект проявится в процессе эксплуатации? Для решения этой проблемы они придумали так называемый хаос-инжиниринг (Chaos Engineering). Вначале там была одна обезьяна, привносящая хаос (Chaos Monkey), а затем появилась целая обезьянья армия (Simian Army). Chaos Monkey занимается имитацией общего хаоса в производственной среде, чтобы увидеть, насколько успешно его выдержит система. Проблемой некоторых AWS-экземпляров была задержка ответной реакции, поэтому Chaos Monkey имитировала высокую задержку (проблема была настолько актуальной, что впоследствии для ее решения была создана специальная обезьяна задержки — Latency Monkey). А такие инструменты, как Chaos Kong, имитировавшие глобальный сбой датацентра Amazon, помогли Netflix избежать печальных последствий, когда подобные сбои происходили в реальности.

Хаос-инжиниринг открывает для архитектуры новые интересные перспективы: вопрос не в том, сломается ли что-нибудь со временем, а в том, что будет, когда это произойдет. Предвидение этих поломок и создание тестов для их предотвращения существенно повышает надежность системы.

Несколько лет назад в весьма важной книге «The Checklist Manifesto»[1], написанной Атулом Гаванде (Atul Gawande)[2] и вышедшей в издательстве Picador, было дано описание того, как чек-листы (иногда требуемые по закону) используются пилотами авиакомпаний и хирургами. Причина не в том, что профессионалы не знают свою работу или страдают забывчивостью. Дело в том, что когда профессионалы из раза в раз выполняют одну и ту же кропотливую работу, мелочи могут легко ускользнуть от их внимания, и тогда краткий чек-лист послужит полезным напоминанием. Такой же подход будет правильным и в отношении функций пригодности: вместо громоздкого механизма управления функции пригодности предоставляют архитекторам способ передачи важных архитектурных принципов и автоматической проверки их соблюдения. Разработчики знают, что они не должны выпускать небезопасный код, но данный приоритет для сильно занятых разработчиков конфликтует с десятками или сотнями других приоритетов. Инструменты, подобные Security Monkey, в частности и функции пригодности в целом позволяют архитекторам встроить важные проверки в саму архитектуру приложения.

[1] https://oreil.ly/XNcV9

[2] Атул Гаванде. «Чек-лист. Как избежать глупых ошибок, ведущих к фатальным последствиям».

ГЛАВА 7

Область действия архитектурных свойств

В мире архитектуры ПО область действия архитектурных свойств традиционно и незыблемо помещалась на системный уровень. Например, если архитекторы говорили о масштабируемости, как правило, подразумевалась масштабируемость всей системы. Подобное допущение не вызывало возражений лет десять назад, когда практически все системы были монолитными. Но с появлением современных приемов проектирования и разрешенных ими архитектурных стилей, например микросервисов, область применения архитектурных свойств значительно сузилась. Этот пример отлично показывает, как аксиомы теряют свою актуальность в процессе непрерывного развития экосистемы разработки ПО.

Когда авторы работали над книгой «Building Evolutionary Architectures»[1], им понадобилась технология измерения степени структурной эволюционности конкретных архитектурных стилей. Ни один из существующих методов измерения не обеспечивал приемлемого уровня детализации. В главе 6 на с. 111 в разделе «Структурные показатели» рассматривались разные метрики на уровне кода, позволяющие анализировать структурные аспекты архитектуры. Однако все эти метрики относятся только к низкоуровневым подробностям кода и не позволяют оценивать зависимые компоненты, которые не входят в кодовую базу (например, базы данных), но все же оказывают влияние на многие свойства архитектуры, особенно на эксплуатационные. Например, сколько бы усилий ни прилагал архитектор к разработке производительной и адаптируемой кодовой базы, но если база данных, используемая системой, не соответствует этим свойствам, приложение не будет успешным.

---

[1] Нил Форд, Ребекка Парсонс, Патрик Куа. «Эволюционная архитектура. Поддержка непрерывных изменений». СПб., издательство «Питер».

При оценке многих эксплуатационных свойств архитектор ПО должен учитывать зависимые компоненты за пределами кодовой базы. Следовательно, архитекторам нужен метод измерения для такого рода зависимостей. Это привело авторов книги «Building Evolutionary Architectures» к введению понятия *квант архитектуры* (*architecture quantum*). Чтобы разобраться в определении кванта архитектуры, нужно сначала рассмотреть одну из ключевых метрик, которая называется коннасценцией.

Связанность и коннасценция

Многие метрики связанности на уровне кода, например *афферентные* и *эфферентные* связанности (рассмотренные в разделе «Связанность» в главе 3 на с. 71), не слишком полезны для архитектурного анализа из-за узкой специализации показателей. В 1996 году Мейлир Пейдж-Джонс опубликовал книгу под названием «What Every Programmer Should Know About Object Oriented Design», в которую были включены несколько новых метрик связанности. Автор назвал их *коннасценцией* (*connascence*) и дал такое определение:

> *Коннасценция*
>
> Два компонента считаются коннасцентными, если изменения, внесенные в один из них, потребуют модификации другого для поддержания общей работоспособности системы.

Он выделял два типа коннасценции: *статическую*, выявляемую с помощью статического анализа кода, и *динамическую*, относящуюся ко времени выполнения приложения. Для обозначения кванта архитектуры нам нужно оценить степень «скрепленности» компонентов, что соответствует понятию коннасценции. Например, если два сервиса в архитектуре микросервисов совместно используют одно и то же определение какого-либо класса, допустим, *адреса*, говорится, что они *статически* коннасцентны относительно друг друга: изменения, вносимые в общий класс, потребуют изменений в обоих сервисах.

Для динамической коннасценции мы вводим два типа: *синхронный* и *асинхронный*. Синхронные вызовы между двумя распределенными сервисами заставляют вызывающий сервис ждать ответа от вызываемого. А *асинхронные* вызовы позволяют в архитектурах, управляемых событиями, воспользоваться семантикой «выстрелил и забыл» (fire-and-forget), то есть два сервиса могут отличаться в эксплуатационной архитектуре.

Архитектурные кванты и гранулярность

Связанность (coupling) на уровне компонентов не единственное, что соединяет элементы программного обеспечения. Многие бизнес-концепции объединяют части системы по смыслу, формируя *функциональную связность (functional cohesion)*. Чтобы успешно проектировать, анализировать и развивать ПО, разработчики должны учитывать все связи, которые могут нарушиться.

Многим разработчикам знакомо понятие кванта из физики: это минимальное количество любой физической сущности, участвующей во взаимодействии. Само слово «квант» произошло от латинского quantum, означающего «как много» или «сколько». Мы воспользовались этим понятием для определения *архитектурного кванта*:

> *Архитектурный квант*
>
> Независимо развертываемый объект с высокой функциональной связностью и синхронной коннасценцией.

Это определение состоит из нескольких частей.

Независимое развертывание

Архитектурный квант содержит все необходимые компоненты, чтобы функционировать независимо от других составных частей архитектуры. Например, если приложение использует базу данных, то она является частью кванта, поскольку без нее система работать не сможет. Это означает, что практически все унаследованные системы (legacy systems), развернутые с использованием одной и той же базы данных, являются по определению одним квантом. А в архитектурном стиле микросервисов каждый сервис включает в себя свою собственную базу данных (это часть философии управления *ограниченным контекстом (bounded context)*, подробно рассматриваемой в главе 17), создавая внутри этой архитектуры несколько квантов.

Высокая функциональная связность

Связность в дизайне компонента означает то, насколько хорошо содержащийся в нем код унифицирован по назначению. Например, компонент `Customer` (Клиент) со свойствами и методами, относящимися только к сущности *Клиент*, проявляет высокую связность, а компонент `Utility` с произвольной коллекцией разнообразных методов — нет.

Под *высокой функциональной связностью* подразумевается, что архитектурный квант служит одной конкретной цели. В традиционной монолитной

архитектуре с единой базой данных эта особенность не имеет практически никакого значения. Но в архитектуре микросервисов разработчики, как правило, проектируют каждый сервис так, чтобы он отвечал за отдельно взятый рабочий процесс (*ограниченный контекст*, см. описание, приведенное ниже во врезке «Ограниченный контекст предметно-ориентированного проектирования» на с. 127), обеспечивая тем самым высокую функциональную связность.

Синхронная коннасценция

Синхронная коннасценция подразумевает выполнение синхронных вызовов в контексте приложения или между распределенными сервисами, образующими этот архитектурный квант. Например, если один сервис в архитектуре микросервисов вызывает другой сервис в синхронном режиме, то каждый из них не может иметь каких-либо существенных отличий в эксплуатационных свойствах архитектуры. Если масштабируемость вызывающего сервиса гораздо выше, чем у вызываемого, то возникнут простои и другие проблемы с надежностью. Следовательно, синхронные вызовы создают динамическую коннасценцию на протяжении всего вызова: если одному компоненту приходится ожидать другой, их эксплуатационные архитектурные свойства в этот период времени должны быть одинаковыми.

Ранее, в главе 3, уже была определена взаимосвязь между традиционными показателями связанности и коннасценцией, но туда не был включен наш новый показатель *коннасценции при обмене данными (communication connascence)*. Обновленная схема представлена на рис. 7.1.

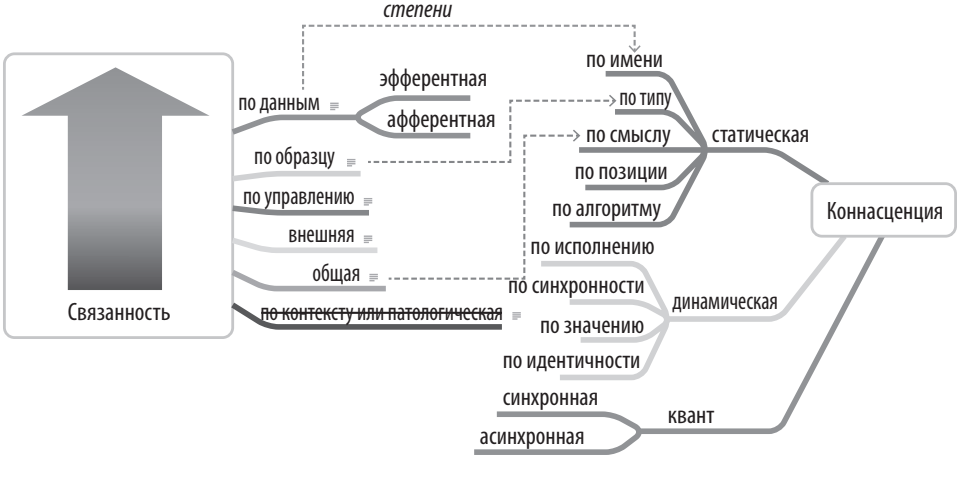


Рис. 7.1. Добавление к общей схеме квантовой коннасценции

Рассмотрим в качестве еще одного примера архитектуру микросервисов с сервисом `Payment` (Платежи) и сервисом `Auction` (Аукционы). Когда аукцион закончен, сервис `Auction` отправляет информацию о платеже сервису `Payment`. Но предположим, что сервис платежей может обрабатывать платеж только каждые 500 мс. Что произойдет, когда одновременно закончатся сразу несколько аукционов? Небрежно спроектированная архитектура позволит пройти первому вызову, а вот с другими допустит задержку. В качестве альтернативы архитектор может спроектировать асинхронный обмен данными между `Payment` и `Auction`, позволив очереди сообщений сгладить различия за счет временной буферизации. В данном случае асинхронная коннасценция позволяет создать более гибкую архитектуру. Детальное рассмотрение данной темы будет предложено в главе 14.

ОГРАНИЧЕННЫЙ КОНТЕКСТ ПРЕДМЕТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ

Книга Эрика Эванса (Eric Evans) «Domain-Driven Design» (издательство Addison-Wesley Professional)[1] оказала существенное влияние на современное архитектурное мышление. Предметно-ориентированное проектирование (Domain-driven design, DDD)[2] — метод моделирования, позволяющий разделять сложные предметные задачи на множество простых. В DDD определяется ограниченный контекст (bounded context), где все относящееся к данной предметной области доступно внутри контекста, но непрозрачно для других ограниченных контекстов. До появления DDD разработчики стремились к повторному использованию всех общих сущностей внутри организации. Но создание общих, совместно используемых артефактов повлекло за собой целый ворох проблем, таких как связанность, более сложная координация и возрастающая сложность приложения. Концепция *ограниченного контекста* учитывает, что каждая сущность лучше работает в рамках локализованного контекста. Вместо создания унифицированного класса `Customer` для всей организации для каждой задачи предметной области может быть создан свой собственный класс, а различия будут согласовываться в точках интеграции.

Концепция архитектурных квантов создает новые области для архитектурных свойств. В современных системах архитекторы определяют архитектурные свойства на уровне квантов, а не на уровне системы. Рассматривая более узкий

[1] Эванс Э. «Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем».

[2] https://dddcommunity.org/

круг важных эксплуатационных проблем, архитекторы могут заранее выявить потенциальные проблемы, что приведет к созданию гибридных архитектур.

Чтобы проиллюстрировать области применения архитектурных квантов, рассмотрим очередное архитектурное ката *Going, Going, Gone* (Кто больше? Продано!).

Конкретный пример: Going, Going, Gone

Понятие архитектурных ката было представлено в главе 5. Рассмотрим еще один пример: интернет-аукцион. Исходные данные архитектурного ката:

Описание

Компания хочет проводить интернет-аукционы на общенациональном уровне. Клиенты выбирают аукцион для участия, дожидаются начала и делают ставки так, как будто они находятся в одном помещении с организатором.

Пользователи

До сотен участников в каждом аукционе, потенциально — до тысячи участников; одновременно может проводиться неограниченное число аукционов.

Требования

- Аукционы, насколько это возможно, должны проходить в режиме реального времени.
- Участники торгов регистрируются с помощью кредитной карты: система автоматически списывает средства с карты, если участник выигрывает торг.
- Участников нужно отслеживать по их индексу репутации.
- Участники торгов могут смотреть прямую видеотрансляцию аукциона и всех ставок по мере их поступления.
- Онлайн-ставки должны обрабатываться в режиме реального времени в порядке их поступления.

Дополнительный контекст

- Аукционная компания агрессивно расширяется за счет слияния с более мелкими конкурентами.
- Бюджет не ограничен. Это стратегическое направление.
- Компания только что вышла из судебного процесса, урегулировав иск по обвинению в мошенничестве.

Здесь, как и в разделе «Конкретный пример: Silicon Sandwiches» из главы 5 на с. 101, архитектору для определения необходимых архитектурных свойств следует учесть каждое из требований:

1. «На общенациональном уровне», «До сотен участников в каждом аукционе, потенциально — до тысячи участников; одновременно может проводиться неограниченное число аукционов», «Аукционы, насколько это возможно, должны проходить в режиме реального времени».

   Каждое из этих требований подразумевает как масштабируемость для поддержки огромного числа пользователей, так и адаптируемость для поддержки разного числа одновременно проводимых аукционов. Масштабируемость требуется явным образом, а вот адаптируемость относится к неявным свойствам, характерным для задач предметной области. Что же касается самих аукционов, возникает вопрос: все ли пользователи ведут себя одинаково сдержанно в ходе торгов или же ближе к окончанию проявляется их бурная активность? Знание архитектором предметной области играет решающую роль в подборе неявных архитектурных свойств. С учетом проведения аукционов в режиме реального времени архитектор, конечно же, в качестве основного свойства станет рассматривать производительность.

2. «Участники торгов регистрируются с помощью кредитной карты: система автоматически списывает средства с карты, если участник выигрывает торг», «Компания только что вышла из судебного процесса, урегулировав иск по обвинению в мошенничестве».

   Оба этих требования указывают на то, что в перечне архитектурных свойств обязательно должна фигурировать безопасность. Как было показано в главе 5, безопасность является неявным архитектурным свойством практически каждого приложения. Следовательно, архитекторы на основании второй части определения архитектурных свойств приходят к выводу, что безопасность оказывает влияние на ряд структурных аспектов проекта. Нужно ли предусматривать введение особых мер безопасности или же для ее обеспечения будет вполне достаточно придерживаться обычных норм при проектировании и программировании? Архитекторами уже разработаны приемы безопасного обращения с кредитными картами на уровне проектирования без выстраивания какой-либо особой структуры. Например, до тех пор пока разработчики гарантированно не станут сохранять номера кредитных карт в обычном тексте, будут шифровать их при передаче и т. д., архитектору не придется вводить особые положения безопасности.

   Но вот вторая фраза должна вынудить архитектора взять паузу и потребовать дополнительных разъяснений. Видимо, какой-то из аспектов безопасности (при котором был допущен факт мошенничества) в прошлом был проблем-

ным, поэтому архитектор должен запросить дополнительные сведения независимо от того, какой уровень безопасности разрабатывается командой.

3. «Участников нужно отслеживать по их индексу репутации».

   Это требование может привести к мысли о неких экзотических свойствах вроде «антитроллинговости» (anti-trollability). Но если сфокусироваться на слове «*отслеживание*», то скорее нужно думать о таких свойствах, как проверяемость и регистрируемость. Чтобы принять решение, нужно снова вернуться к определяющему фактору: выходит ли рассматриваемый вопрос за рамки решения задач предметной области? Такой анализ является лишь малой частью общих усилий по проектированию и реализации приложения, а основная проектная работа выполняется уже после этой фазы! На данном этапе архитекторы определяют, какие факторы могут повлиять на структуру помимо решения задач предметной области.

   «Лакмусовой бумажкой» для определения, относится ли некоторое свойство к предметной области или же к архитектуре, будет ответ на вопрос: требует ли оно знания предметной области для внедрения? Или же это абстрактное архитектурное свойство? В ката «Going, Going, Gone» архитектор, столкнувшись со словосочетанием «индекс репутации», станет искать бизнес-аналитика или же другого эксперта, чтобы тот объяснил, что конкретно имеется в виду. Иными словами, словосочетание «индекс репутации» не является стандартным определением, похожим на более общие свойства архитектуры. В качестве контрпримера можно привести случай обсуждения архитекторами такого свойства, как *адаптируемость*, то есть возможность справляться с резкими наплывами пользователей. Оно может быть отнесено к общим свойствам, и совершенно неважно, говорим ли мы о приложении, связанном с банковской деятельностью, с ведением сайта каталогов, с потоковым видео или чем-то еще. Архитекторы должны определить, подразумевается ли какое-то требование самой предметной областью *и* не диктует ли оно в связи с этим необходимость применения какой-либо конкретной структуры, которая переведет данный вопрос в разряд архитектурных свойств.

4. «Аукционная компания агрессивно расширяется за счет слияния с более мелкими конкурентами».

   Это требование хоть и не оказывает существенного влияния на архитектуру, но может оказаться решающим при компромиссе между несколькими равнозначными вариантами. Например, архитекторам для обеспечения интеграции часто приходится выбирать такие детали, как протоколы связи: если слияние с другими компаниями не вызывает особого беспокойства, это дает архитектору возможность применить что-то очень конкретное для решения задачи. Или, наоборот, архитектор может выбрать нечто далеко не идеальное, чтобы

учесть ряд дополнительных компромиссов, связанных, к примеру, с интероперабельностью. Слабо выраженные неявные архитектурные свойства, подобные этому, буквально пронизывают архитектуру. Это объясняет, почему качественное выполнение работы сопряжено с трудностями.

5. «Бюджет не ограничен. Это стратегическое направление».

   На решения некоторых архитектурных ката накладываются бюджетные ограничения, чтобы приблизить их к реальным условиям. Но к нашему ката «Going, Going, Gone» это не относится. Поэтому архитектор может остановить свой выбор на более сложных и/или специализированных архитектурах, что особенно пригодится для выполнения следующих требований.

6. «Участники торгов могут смотреть прямую видеотрансляцию аукциона и всех ставок по мере их поступления». «Онлайн-ставки должны обрабатываться в режиме реального времени в порядке их поступления».

   Это требование ставит весьма интересную архитектурную задачу, несомненно влияющую на структуру приложения и показывающую бесполезность рассмотрения архитектурных свойств в виде общесистемных определений. Взять, к примеру, доступность: является ли она одинаковой для всей архитектуры? Иными словами, является ли доступность для какого-то одного конкретного участника торгов более важной, чем доступность для любого из сотен других участников? Разумеется, архитектор желает добиться надлежащих показателей для всех участников, но все же доступность для одного из них намного важнее: если организатор аукциона не получит выхода на сайт, онлайн-торгов не будет ни для кого. Надежность зачастую идет рядом с доступностью: она связана с такими эксплуатационными аспектами, как время безотказной работы, а также целостность данных и другие показатели стабильности работы приложения. Например, на сайте аукционных торгов архитектор должен обеспечить абсолютную корректность порядка следования сообщений, исключив состояние гонки и возникновение иных проблем.

Последнее требование в ката «Going, Going, Gone» подчеркивает необходимость детализированного подхода к архитектуре. Используя понятие архитектурного кванта, можно перенести архитектурные свойства с уровня системы на уровень квантов. Например, в «Going, Going, Gone» архитектор заметил бы, что разные части этой архитектуры нуждаются в разных свойствах: тремя вполне очевидными претендентами на особые свойства являются потоковые ставки, онлайн-участники и организатор торгов. Архитекторы используют архитектурные кванты в обсуждении вопросов развертывания, связанности, мест размещения данных и стилей обмена данными внутри архитектуры. В данном ката архитектор может провести анализ потребностей в различных архитектурных свойствах

для каждого кванта и прийти к мысли о разработке гибридной архитектуры на более ранних стадиях процесса.

В итоге для «Going, Going, Gone» мы выделили следующие кванты и соответствующие им архитектурные свойства:

Bidder feedback (Взаимодействие с участниками торгов)

Включает в себя поток и видеопоток заявок

- Доступность
- Масштабируемость
- Производительность

Auctioneer (Организатор торгов)

Реальный организатор торгов

- Доступность
- Надежность
- Масштабируемость
- Адаптируемость
- Производительность
- Безопасность

Bidder (Участник торгов)

Онлайн-участники торгов и торги

- Надежность
- Доступность
- Масштабируемость
- Адаптируемость

ГЛАВА 8

Компонентно-ориентированное мышление

В главе 3 рассматривались *модули* как группы взаимосвязанного кода. Но обычно архитекторы мыслят в понятиях *компонентов*, являющихся физическим воплощением модуля.

Разработчики упаковывают модули различными способами, часто зависящими от платформы разработки. Физическая упаковка модулей называется *компонентом*. Такая упаковка поддерживается большинством языков: `jar`-файлами в Java, `dll` в .NET, `gem` в Ruby и т. д. В этой главе мы рассмотрим компоненты с точки зрения архитектуры, начиная с области применения и заканчивая их выявлением.

Область применения компонентов

Разделение ПО на компоненты можно осуществлять с помощью разных подходов; некоторые из них показаны на рис. 8.1.

Компоненты предлагают специфичный для того или иного языка механизм группировки различных объектов, используя многослойную структуру. На рис. 8.1 показано, что даже самый простой компонент упаковывает код на более высоком уровне модульности, чем класс (или функция в необъектно-ориентированных языках). Эта простая упаковка часто называется *библиотекой*, то есть выполняется по тому же адресу памяти, что и вызывающий код, и взаимодействует через языковые механизмы вызова функций. Библиотеки обычно являются зависимостями этапа компиляции (за исключением динамически подключаемых библиотек [DLL], которые многие годы раздражали пользователей Windows).

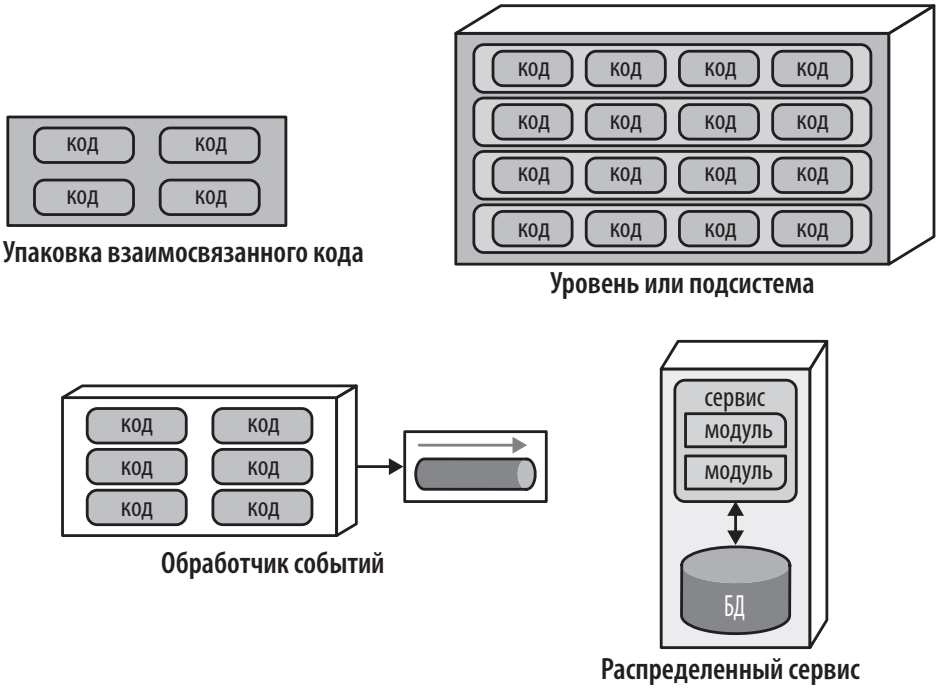


Рис. 8.1. Многообразие компонентов

Компоненты также выступают в роли подсистем или уровней архитектуры, как развертываемая единица для многих обработчиков событий. Еще один тип компонентов, *сервис*, обычно запускается в своем собственном адресном пространстве и обменивается данными с помощью низкоуровневых сетевых протоколов типа TCP/IP либо таких высокоуровневых форматов, как REST или очереди сообщений, формируя автономные развертываемые единицы — микросервисную архитектуру.

От архитекторов не требуется обязательно работать с компонентами, просто зачастую целесообразно использовать уровень модульности выше минимально предлагаемого самим языком. Например, в архитектуре микросервисов одним из архитектурных принципов является простота. Сервис может состоять из достаточного объемного кода, чтобы имело смысл создавать компоненты, а может быть совсем простым, содержащим небольшой объем кода, как показано на рис. 8.2.

Компоненты крайне важны для архитекторов, потому что являются основным строительным блоком для создания модулей. По сути, одним из первостепенных обязательных решений, принимаемых архитектором, является высокоуровневая разбивка на компоненты.

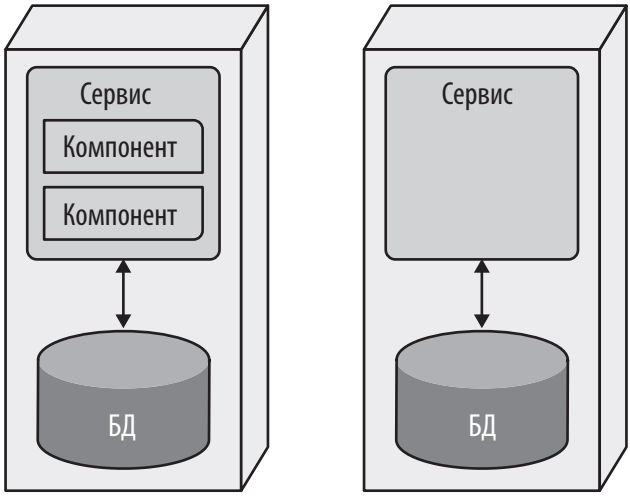


Рис. 8.2. У микросервиса может быть так мало кода, что компоненты ему не понадобятся

Задача архитектора

Обычно архитектор определяет компоненты, составляющие архитектуру, уточняет их состав и управляет их созданием и сопровождением. Архитекторы ПО в сотрудничестве с бизнес-аналитиками, экспертами в предметной области, разработчиками, тестировщиками, специалистами по сопровождению и корпоративными архитекторами создают исходный дизайн ПО, учитывая архитектурные свойства, рассмотренные в главе 4, и требования к программной системе.

Практически все вопросы, которые мы рассматриваем в этой книге, не зависят от выбранного метода проектирования ПО: архитектура не обусловлена процессом разработки. Исключением из этого правила является практика проектирования, впервые примененная в различных вариантах Agile-разработки, особенно в областях развертывания и автоматизации управления. Но в целом архитектура ПО существует отдельно от процесса. Следовательно, архитекторам в конечном счете все равно, откуда берутся требования: из формального процесса совместного проектирования приложений (Joint Application Design, JAD), пространного анализа и проектирования в водопадном стиле, Agile-подходов... или же из любого их сочетания.

Компонент, как правило, является самым низким уровнем программной системы, с которым архитектор работает напрямую, за исключением показателей качества кода, рассмотренных в главе 6 и влияющих на кодовые базы в целом.

Компоненты состоят из классов или функций (в зависимости от платформы реализации), за разработку которых отвечают техлиды или разработчики. Это не означает, что архитекторы не должны принимать участие в проектировании классов (особенно в выявлении или применении подходящих паттернов проектирования). Однако необходимо избегать контроля за принятием каждого решения, даже самого незначительного. Если архитекторы не позволят никому другому самостоятельно принимать решения, организации в дальнейшем придется бороться со слишком широкими правами и обязанностями следующего поколения архитекторов.

Архитектор должен определиться с компонентами в самом начале разработки нового проекта. Но перед этим необходимо научиться разбивать архитектуру на части.

Разбиение архитектуры

Первый закон архитектуры ПО гласит, что все в программах соткано из компромиссов, включая и определение используемых компонентов. Поскольку компоненты представляют собой общий механизм упаковки кода в контейнеры, архитектор может применять разбиение любого подходящего ему типа. Существует несколько общепринятых стилей с различными наборами компромиссов. Более подробно архитектурные стили будут рассматриваться в части II книги. А здесь мы коснемся одного из ключевых вопросов стиля: *высокоуровневого разбиения* архитектуры на части.

Рассмотрим два типа архитектурных стилей, показанных на рис. 8.3.

Одним из популярных типов архитектуры, показанным на рис. 8.3, является *многоуровневый монолит* (мы подробно рассмотрим его в главе 10). Второй тип, популяризированный в свое время Симоном Брауном (Simon Brown), называется *модульным монолитом* и является единицей развертывания, связанной с базой данных и разделенной по задачам предметной области, а не по техническим возможностям. Эти два стиля представляют собой различные способы *высокоуровневого разбиения* архитектуры на части. Следует заметить, что в каждом из вариантов любой из высокоуровневых компонентов (уровней или собственно компонентов) обладает, вполне вероятно, своими собственными встроенными компонентами. Высокоуровневое разбиение представляет особый интерес для архитекторов, поскольку при этом определяется фундаментальный архитектурный стиль и способ разбиения кода на части.

Организация архитектуры на основе технических возможностей с получением многоуровневой архитектуры представляет собой *техническое многоуровневое разбиение*. Его типичная версия представлена на рис. 8.4.

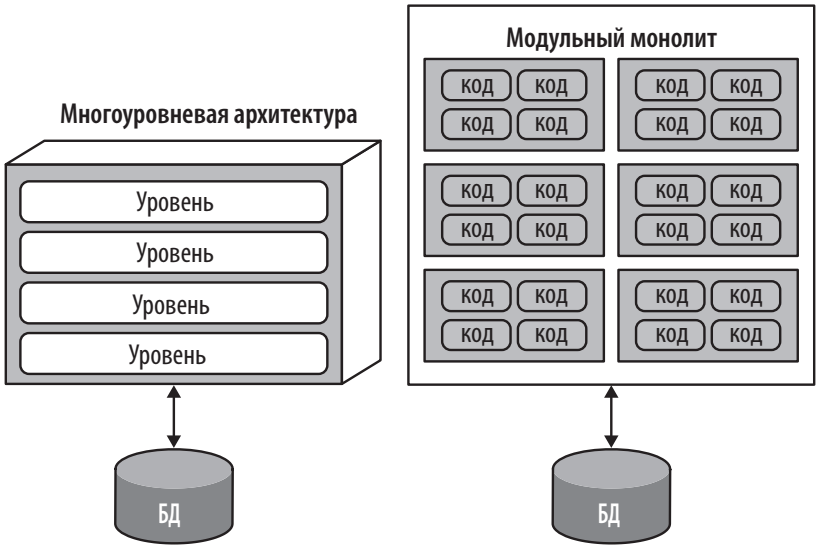


Рис. 8.3. Два типа высокоуровневого разбиения архитектуры на части: на уровни и на модули

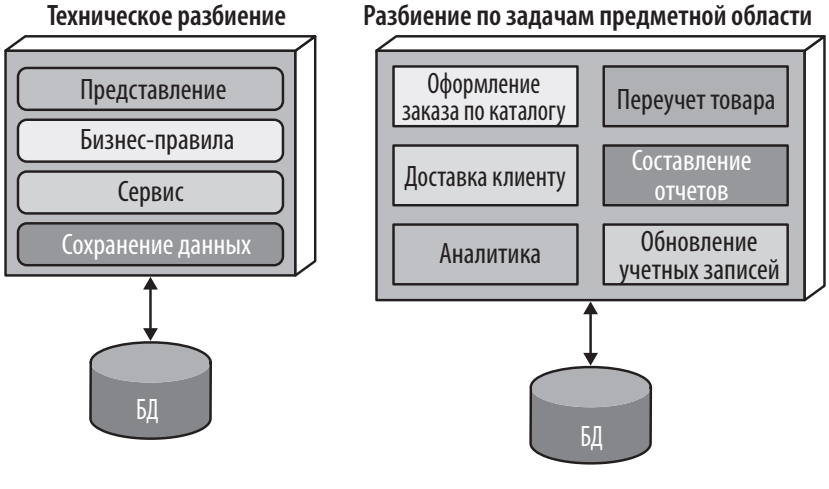


Рис. 8.4. Два типа высокоуровневого разбиения архитектуры

На рис. 8.4 показано, что архитектор разбил функциональность системы по *техническим* возможностям: представление, бизнес-правила, сервисы, сохранение данных и т. д. Такой способ организации кодовой базы, конечно же, имеет смысл. Весь код, связанный с сохранением данных, находится на одном уровне архитектуры, упрощая разработчикам поиск кода, имеющего отношение к этому сохранению. Несмотря на то что основная концепция разбиения архитектуры на уровни появилась на целые десятилетия раньше паттерна Модель — Представление — Контроллер (Model — View — Controller, MVC), он соответствует этой архитектурной схеме, упрощая разработчикам понимание кодовой базы. Поэтому данная архитектура часто используется по умолчанию.

Интересный побочный эффект преобладания многоуровневой архитектуры связан с тем, как организации рассаживают специалистов по рабочим местам.

Можно заметить интересную особенность: многоуровневая архитектура чаще всего применяется в разработке у тех организаций, где специалистов объединяют в группы по выполняемым задачам. При работе с такой архитектурой удобнее, если все разработчики серверной части будут в одном отделе, администраторы баз данных — в другом, фронтэнд-специалисты — в третьем и т. д. В таких организациях действует *закон Конвея*.

ЗАКОН КОНВЕЯ

В конце 60-х годов прошлого века Мелвин Конвей (Melvin Conway) подметил то, что впоследствии стали называть *законом Конвея*:

Организации, проектирующие системы ... создают проекты со структурой, которая копирует коммуникацию в этих организациях

Закон можно перефразировать и предположить, что при разработке группой специалистов какого-нибудь технического решения сложившаяся в группе организационная структура в конечном итоге воспроизводится в проекте. Люди на всех уровнях организаций видят этот закон в действии и иногда принимают решения на его основе. Например, работников часто разделяют по специализациям, что целесообразно с чисто организационной точки зрения, но затрудняет их сотрудничество из-за искусственно созданного несовпадения целей.

Наблюдение, сделанное Джонни Лероем (Jonny Leroy) из ThoughtWorks, превратилось в обратный закон Конвея (Inverse Conway Maneuver), который заключается в том, что для получения необходимой архитектуры требуется преобразование организационной структуры команды.

Другая разновидность архитектуры, показанная на рис. 8.4, представляет собой *предметное разбиение*, основанное на методе предметно-ориентированного проектирования (Domain-driven design, DDD), изложенном в книге Эрика Эванса «Domain-Driven Design»[1]. Это метод моделирования для декомпозиции сложных программных систем. Применяя DDD, архитектор определяет независимые друг от друга и несвязанные предметные области или рабочие потоки. На этой философии построен архитектурный стиль микросервисов (рассматриваемый в главе 17). В модульном монолите архитектор разбивает архитектуру не по техническим возможностям, а по предметным областям или рабочим потокам. Поскольку компоненты зачастую вложены друг в друга, каждый из компонентов, показанных на рис. 8.4 в той части, где отражено предметное разбиение (например, компонент *CatalogCheckout* — Оформление заказа по каталогу), может использовать библиотеку сохранения данных и иметь отдельный уровень для бизнес-правил, но при этом высокоуровневое разбиение будет основано на предметных областях.

Одно из фундаментальных различий между разными архитектурными паттернами — тип поддерживаемого высокоуровневого разбиения. Это также оказывает огромное влияние на решение архитектора по исходной идентификации компонентов: необходимо разбиение в техническом или же в предметном плане?

При использовании технического разбиения архитекторы выстраивают компоненты системы по их техническим возможностям: представлению, бизнес-правилам, сохранению данных и т. д. Следовательно, одним из принципов выстраивания такой архитектуры является ее *разделение по техническим задачам*. А это, в свою очередь, приводит к созданию удобных уровней развязывания (decoupling): если уровень сервиса обменивается данными только с расположенным ниже уровнем сохранения данных и расположенным выше уровнем бизнес-правил, то вносимые в него изменения потенциально смогут повлиять только на эти два уровня. Этот стиль разбиения предоставляет метод уменьшения связанности, который снижает негативные побочные эффекты, влияющие на зависимые компоненты. Более подробно этот архитектурный стиль будет рассмотрен при разборе многоуровневого архитектурного паттерна в главе 10. Выстраивание систем с помощью технического разбиения — довольно логичное решение, но, как и все в архитектуре ПО, не обходится без целого ряда компромиссов.

Разделение на основе технического разбиения позволяет разработчикам быстро находить конкретные категории кодовой базы. Но большинству реальных про-

[1] Эванс Э. «Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем».

граммных систем требуются рабочие потоки, которые не могут быть ограничены каким-то одним техническим уровнем. Рассмотрим обычный рабочий процесс *CatalogCheckout* (*Оформление заказа по каталогу*).

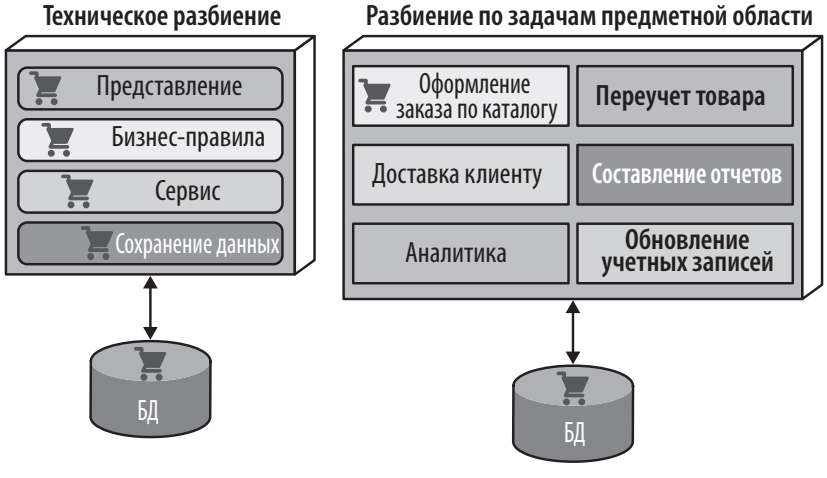


Рис. 8.5. Места, где предметные области или рабочие потоки появляются в архитектурах, разбитых по техническому и предметному принципам

На рис. 8.5 показано, что в архитектуре с техническим разбиением код решения задачи *CatalogCheckout* появляется на всех уровнях, то есть задача предметной области «размазана» по техническим уровням. Сравним это с разбиением по задачам предметной области, где используется высокоуровневое разбиение, выстраивающее компоненты по предметным задачам, а не по техническим возможностям. На этом же рисунке видно, что архитекторы, задумавшие архитектуру по принципу предметного разбиения, выстраивают высокоуровневые компоненты вокруг рабочих потоков и/или задач предметной области. Каждый компонент при разбиении по предметному принципу может иметь подкомпоненты, включая уровни, но разбиение верхнего уровня фокусируется на предметной области, и это лучше отражает виды изменений, чаще всего происходящие в проектах.

Ни один из этих стилей не лучше другого: вспомните Первый закон архитектуры программного обеспечения. И тем не менее последние несколько лет в сфере производства ПО наблюдается тенденция предметного разбиения для монолитных и распределенных (например, микросервисных) архитектур. Но выбор стиля является одним из первоочередных обязательных решений архитектора.

Конкретный пример: Silicon Sandwiches. Разбиение

Вернемся к одному из наших учебных ката «Конкретный пример: Silicon Sandwiches» из главы 5 на с. 101. При определении компонентов одной из основных задач, стоящих перед архитектором, будет высокоуровневое разбиение. Рассмотрим первую из двух различных возможностей для Silicon Sandwiches — предметное разбиение, показанное на рис. 8.6.

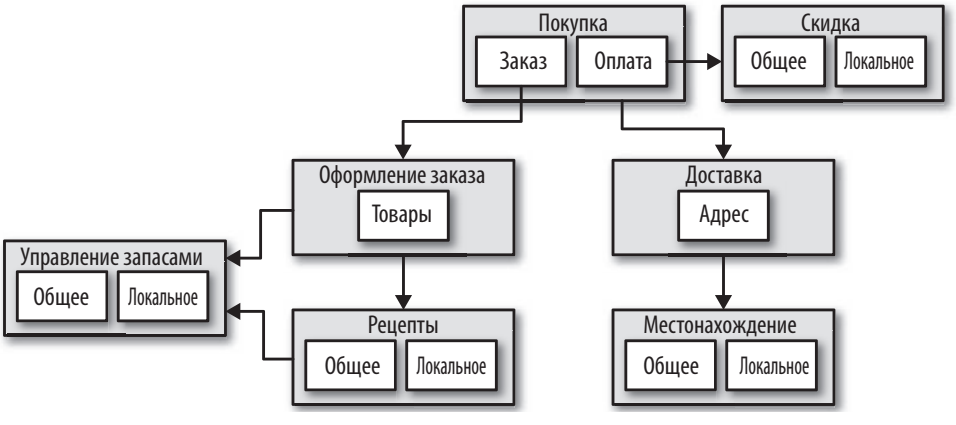


Рис. 8.6. Схема проекта Silicon Sandwiches при предметном разбиении

На рис. 8.6 показано, что архитектор выстроил проект вокруг задач предметной области (рабочих потоков), создавая отдельные компоненты `Purchase` (Покупка), `Promotion` (Скидка), `MakeOrder` (Оформление заказа), `ManageInventory` (Управление запасами), `Recipes` (Рецепты), `Deliver` (Доставка) и `Location` (Местонахождение). Многие из этих компонентов содержат подкомпоненты для работы с различными настройками, как общими, так и локальными.

Альтернативный вариант, в котором общие и локальные части изолированы в своих разделах, показан на рис. 8.7. В состав высокоуровневых компонентов входят `Common` (общее) и `Local` (локальное), а для действий в рамках рабочего процесса остаются `Purchase` и `Delivery`.

Что лучше? Все зависит от конкретных обстоятельств! У каждого разбиения свои достоинства и недостатки.

Предметное разбиение

В архитектурах с предметным разбиением высокоуровневые компоненты разделяются по рабочим потокам и/или предметным областям.

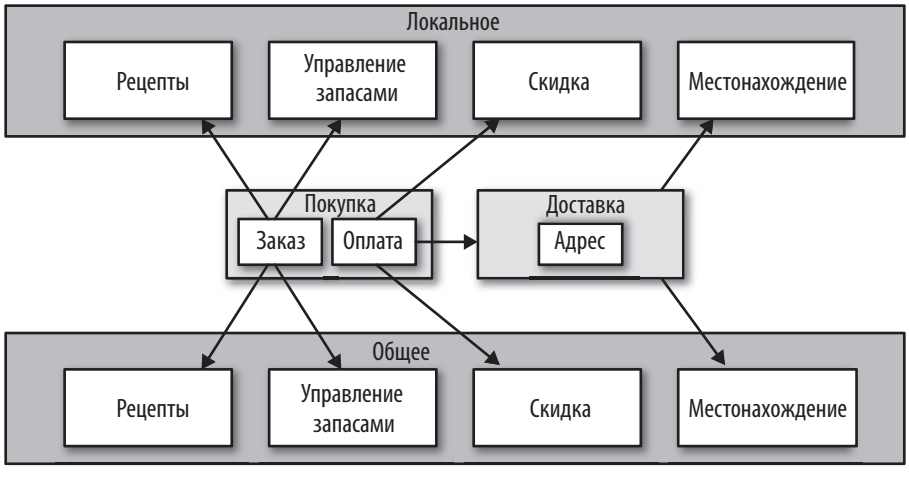


Рис. 8.7. Схема проекта Silicon Sandwiches при техническом разбиении

Достоинства

- Моделирование больше ориентировано на бизнес-процессы, а не на особенности реализации.
- Упрощает использование обратного закона Конвея для формирования кросс-функциональных команд вокруг задач предметной области.
- Больше подходит для архитектурных стилей модульного монолита и микросервисов.
- Поток сообщений соответствует кругу задач предметной области.
- Упрощает миграцию данных и компонентов в распределенные архитектуры.

Недостатки

- Настроечный код имеется сразу в нескольких местах.

Техническое разбиение

В архитектуре, разбитой по техническому принципу, высокоуровневые компоненты отделяются друг от друга не по признакам отдельных рабочих потоков, а на основе технических возможностей. Это может проявляться в виде уровней, соответствующих разделению по схеме Модель — Представление — Контроллер, или же в виде какого-либо другого специализированного технического разбиения. На рис. 8.7 показано разбиение на компоненты на основе настроек под конкретные технические требования.

Достоинства

- Настроечный код четко разделен.
- Больше подходит для паттерна многоуровневой архитектуры.

Недостатки

- Более высокий уровень глобальной связанности. Изменения, вносимые в компонент `Common` или в компонент `Local`, скорее всего, потребуют внесения изменений и во все остальные компоненты.
- Разработчики могут столкнуться с необходимостью повторять код решения предметных задач как на уровне `common`, так и на уровне `local`.
- Как правило, на уровне данных возникает более высокий уровень связанности. В такой системе архитекторам приложений и данных придется создавать единую базу данных, где будут настройки, а также решения задач предметной области. В результате возникнут трудности с разрывом связанности данных, если потребуется миграция этой архитектуры в распределенную систему.

Другие факторы, которые влияют на решение архитектора о выборе архитектурного стиля для своего дизайна, будут рассмотрены в части II книги.

Задача разработчика

Разработчики обычно берут компоненты, определенные совместно с архитекторами, и приступают к их дальнейшему разбиению на классы, функции или подкомпоненты. По сути, дизайн классов и функций является общей обязанностью архитекторов, техлидов и разработчиков, но основная тяжесть возлагается на разработчиков.

Разработчики не должны воспринимать компоненты, созданные архитекторами, как истину в последней инстанции: как правило, разработка ПО от итерационного проектирования только выигрывает. Скорее исходный дизайн нужно рассматривать как первый набросок, который будет конкретизироваться и уточняться.

Процесс выявления компонентов

Выявление компонентов лучше всего делать итерационным путем, когда кандидаты предлагаются и уточняются через обратную связь. Цикл показан на рис. 8.8.

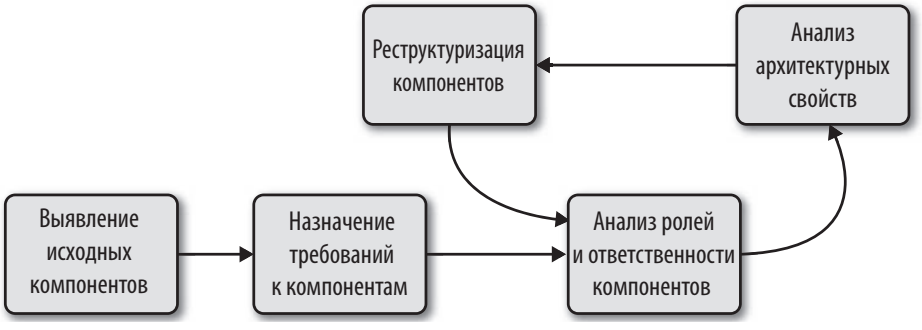


Рис. 8.8. Цикл выявления компонентов

Этот цикл описывает общий процесс утверждения архитектуры. Для специфических предметных задач, возможно, придется добавлять в процесс дополнительные этапы или даже полностью изменить его. Например, в некоторых предметных областях часть кода должна пройти этапы проверки безопасности или пригодности. Каждый этап, показанный на рис. 8.8, будет рассмотрен в следующих разделах.

Выявление исходных компонентов

Прежде чем для программного проекта будет написан какой-либо код, архитектор должен определить исходные высокоуровневые компоненты на основе выбранного принципа высокоуровневого разбиения. Помимо этого, архитектор может задать любые нужные ему компоненты, а затем сопоставить с ними функционал предметной области, чтобы понять, где именно будет закладываться нужное поведение программы. Возможно, это будет весьма поверхностный анализ, но начинать с чего-то более конкретного, если архитектор разрабатывает системы с нуля, довольно трудно. Вероятность получения удачного дизайна из исходного набора компонентов крайне мала, поэтому архитекторы должны итерационно уточнять дизайн компонентов, добиваясь его улучшения.

Назначение требований к компонентам

После того как архитектор выявит набор исходных компонентов, следующим этапом будет сопоставление требований (или пожеланий заказчиков) и компонентов, чтобы определить, насколько они соответствуют друг другу. Это может привести к созданию новых компонентов, объединению существующих или

к разбиению компонентов на части с меньшей функциональностью. Данная картина не должна быть абсолютно точной; пока речь идет о примерной структуре, допускающей дальнейшую проработку и уточнение дизайна с помощью техлидов и/или разработчиков.

Роли и ответственности

При распределении задач по компонентам архитектор также обращает внимание на роли и ответственности, вытекающие из требований, чтобы убедиться, что гранулярность (степень детализации) соответствует им. Анализируя роли и особенности поведения, которые должны поддерживаться приложением, архитектор может выровнять гранулярность компонентов и предметной области. Одна из самых сложных проблем для архитекторов как раз и заключается в выборе правильной степени детализации, и для ее решения очень эффективен итерационный подход.

Анализ архитектурных свойств

При назначении требований к компонентам архитектор должен также иметь в виду заданные ранее архитектурные свойства и то, как они влияют на разбиение архитектуры на компоненты и гранулярность. Например, если две части системы работают с пользовательским вводом, та часть, которая одновременно обслуживает сотни пользователей, будет нуждаться в иных архитектурных свойствах, нежели другая часть, которая обслуживает лишь нескольких. Следовательно, в то время как при чисто функциональном взгляде может быть выявлен один компонент для взаимодействия с пользователями, анализ архитектурных свойств приведет к его разделению на два отдельных компонента.

Реструктуризация компонентов

Обратная связь в проектировании ПО играет решающую роль. Поэтому архитекторы должны постоянно итеративно взаимодействовать с разработчиками при дизайне компонентов. В проектировании ПО возникают разнообразные непредвиденные трудности: никто не в состоянии заранее предугадать весь круг потенциальных проблем. Ключом к успеху является подход с последовательным приближением. Во-первых, практически невозможно учесть все неожиданности и краевые случаи, которые могут привести к пересмотру дизайна компонентов. Во-вторых, по мере углубления в процесс создания приложения у архитекторов

и разработчиков возникает более детальное понимание того, где должны находиться нужное поведение и роли.

Гранулярность компонентов

Выявление правильной гранулярности компонентов — одна из сложнейших задач, решаемых архитектором. Слишком мелкое дробление влечет за собой увеличение количества взаимодействий между компонентами. Наоборот, грубая степень детализации приводит к высоким показателям внутренней связанности, создающей сложности в развертываемости и тестируемости, а также негативные побочные эффекты, связанные с недостаточной модульностью.

Проектирование компонентов

Не существует никакого общепринятого «правильного» способа проектирования компонентов, скорее есть разнообразие методов, у каждого из которых свои преимущества и недостатки. В любом случае архитектор исходит из требований и старается в первом приближении определить строительные блоки, из которых будет состоять приложение. Все существующие разнообразные методы сотканы из компромиссов и привязаны к принятому в команде и организации процессу разработки ПО. Здесь мы расскажем лишь о нескольких основных способах выявления компонентов и о том, как не попасть в ту или иную западню.

Выявление компонентов

Архитекторы совместно с другими специалистами — разработчиками, бизнес-аналитиками и экспертами в предметной области — создают исходный дизайн компонентов на основе общего понимания системы, а также выбранного способа ее декомпозиции (техническое или предметное разбиение). Цель команды — определиться с первоначальным дизайном, грубо разбивая на части пространство задачи и учитывая необходимые архитектурные свойства.

Западня сущности

Единственно верного способа выявления компонентов не существует, зато есть распространенный антипаттерн: *Западня сущности (Entity trap)*. Предположим, что архитектор работает над проектированием компонентов для нашего

ката Going, Going, Gone и остановился на дизайне, примерно изображенном на рис. 8.9.

Рис. 8.9. Выстраивание архитектуры в виде объектно-реляционного отображения

На рис. 8.9 показано, что архитектор просто взял каждую выявленную в требованиях сущность и на ее основе создал компонент `Manager`. Это не архитектура, а простое компонентно-реляционное отображение фреймворка на базу данных. Иными словами, если система нуждается всего лишь в простых CRUD-операциях в базе данных (CRUD — create [создание], read [чтение], update [обновление], delete [удаление]), то архитектор может загрузить фреймворк для создания пользовательского интерфейса непосредственно из базы данных.

Антипаттерн Западня сущности складывается тогда, когда архитектор неверно отождествляет связи в базе данных с рабочими процессами в приложении (что довольно редко встречается в реальном мире). Применение этого антипаттерна означает, скорее всего, отсутствие представления о реальной работе приложения. Компоненты, созданным под влиянием Западни сущности, слишком грубо детализированы, что не дает разработчикам понимания того, как упаковывать и структурировать исходный код.

Подход типа актор — действия

Подход *актор — действия (actor/actions)* является весьма популярным способом, с помощью которого архитекторы выявляют соответствие требований и компонентов. При этом подходе, изначально созданном в рамках методологии Rational Unified Process, архитекторы определяют акторов, выполняющих действия в приложении, и действия, которые могут выполняться этими акторами. Тем самым выявляются типичные пользователи системы и то, что они могут делать с системой.

NAKED OBJECTS И АНАЛОГИЧНЫЕ ФРЕЙМВОРКИ

Более 10 лет назад появилось целое семейство фреймворков, превративших создание простых CRUD-*приложений в тривиальную задачу.* Примером может послужить Naked Objects (со временем разделившийся на два проекта: .NET-версию, сохранившую название NakedObjects[1], и Java-версию под названием Isis[2], переместившуюся на Apache-сервер с открытым исходным кодом). В основу этих фреймворков заложено создание пользовательского интерфейса на основе объектов базы данных. Например, в Naked Objects разработчик указывает фреймворку на таблицы базы данных, и тот, в свою очередь, создает пользовательский интерфейс на основе этих таблиц и отношений между ними.

Также существует несколько других популярных фреймворков, которые в основном предоставляют пользовательский интерфейс по умолчанию на основе структуры таблиц базы данных. Например, Ruby on Rails[3] с помощью механизма скаффолдинга по умолчанию отображает базу данных на веб-сайт (с возможностью расширения и усложнения финальной версии приложения).

Если потребности архитектора ограничиваются простым отображением базы данных на пользовательский интерфейс, полноценная архитектура не нужна, вполне достаточно будет одного из этих фреймворков.

Подход актор — действия стал популярным в связи с конкретными процессами разработки ПО, особенно с наиболее формальными, предпочтение в которых отдается весьма существенному объему предварительного проектирования. Он по-прежнему популярен и вполне приемлем в том случае, если в требованиях ясно прописаны роли и варианты их действий. Такой стиль декомпозиции компонентов работает для всех типов систем, как монолитных, так и распределенных.

Штурм событий

Штурм событий (event storming) как метод выявления компонентов пришел из предметно-ориентированного проектирования (DDD) и разделяет свою популярность с микросервисами, на которые также оказала существенное влияние

[1] https://github.com/NakedObjectsGroup/NakedObjectsFramework

[2] http://isis.apache.org/

[3] https://rubyonrails.org/

методология DDD. При штурме событий архитектор предполагает, что в проекте для обмена данными между различными компонентами будут использоваться сообщения и/или события. Команда старается на основе требований и ролей определить, какие события происходят в системе, после чего выстроить компоненты вокруг обработчиков этих событий и сообщений. Данный метод неплохо работает в распределенных архитектурах вроде микросервисов, где используются события и сообщения, поскольку он помогает архитекторам определить сообщения, используемые в конечной системе.

Подход на основе рабочих потоков

Это альтернатива штурму событий, но с более общим подходом, не использующим DDD или обмен сообщениями. *Подход на основе рабочих потоков (workflow approach)* заключается в моделировании компонентов вокруг рабочих потоков, что во многом напоминает штурм событий, но без явных ограничений, связанных с построением системы на основе сообщений. Подход на основе рабочих потоков позволяет выявлять ключевые роли, определять, в какие рабочие потоки вовлечены эти роли, и создавать компоненты вокруг выявленных действий.

Ни один из описанных методов не имеет явных преимуществ перед остальными: все они содержат те или иные компромиссы. Если команда использует водопадный подход или другие более ранние процессы разработки ПО, она может отдать предпочтение методу *актор — действия*, потому что он наиболее универсальный. При использовании DDD и соответствующих этой методологии архитектур вроде микросервисов лучшим выбором будет *штурм событий*.

Конкретный пример: Going, Going, Gone. Выявление компонентов

Если у команды нет особых ограничений и она ищет подходящий универсальный способ декомпозиции компонентов, то в качестве общего решения вполне подойдет подход актор — действия. Именно им мы и воспользуемся в учебном примере для ката Going, Going, Gone.

В главе 7 было представлено архитектурное ката Going, Going, Gone (GGG) и найдены подходящие для него архитектурные свойства. Здесь есть три очевидные роли: *участник торгов (bidder)*, *организатор торгов (auctioneer)* и частый фигурант данного приема моделирования — *система (system)* для внутренних действий. Первые две роли взаимодействуют с приложением, представленным

системой; именно она, а не участники или организатор торгов определяет, когда приложение инициирует выдачу события. Например, в GGG, как только аукцион завершится, система запускает сервис обработки платежей.

Мы можем также определить стартовый набор действий для каждой из этих ролей:

`Bidder` (Участник торгов)

Просматривает видеопоток реального (живого) аукциона, просматривает поток живых ставок, делает ставку.

`Auctioneer` (Организатор торгов)

Вводит в систему живые ставки, получает онлайн-ставки, помечает позицию проданной.

`System` (Система)

Запускает аукцион, выполняет платеж, отслеживает активность участников торгов.

Исходя из этих действий, мы можем итеративно составить набор стартовых компонентов для GGG. Одно из таких решений показано на рис. 8.10.

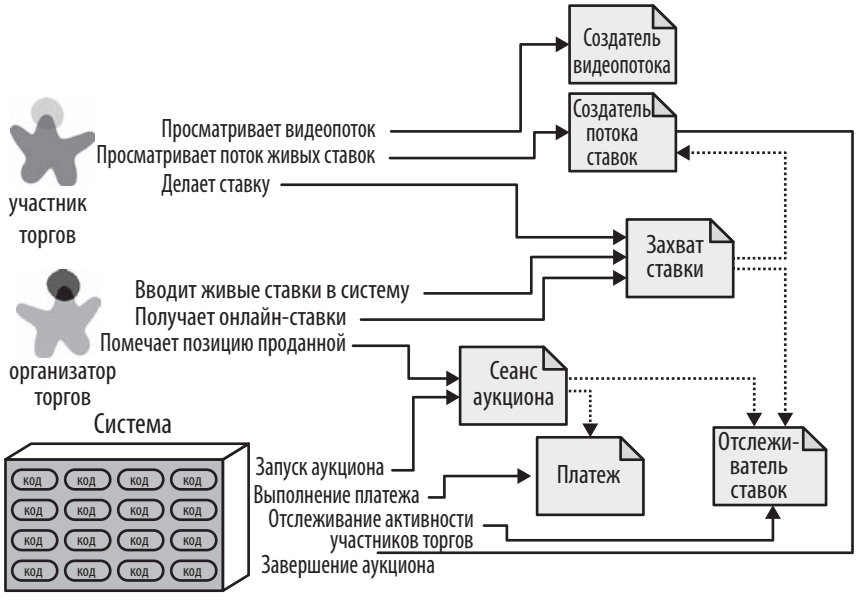


Рис. 8.10. Исходный набор компонентов для Going, Going, Gone

Каждая роль и каждое действие, показанные на рис. 8.10, сопоставляются с компонентом, которому, в свою очередь, может потребоваться совместная работа с информацией. Вот компоненты, выявленные для данного решения:

`VideoStreamer` (Создатель видеопотока)

Отправляет пользователям потоковое видео аукциона.

`BidStreamer` (Создатель потока ставок)

Отправляет пользователям потоковое видео сделанных ставок. Оба компонента, VideoStreamer и BidStreamer, предлагают участнику торгов просмотр аукциона в режиме только для чтения.

`BidCapture` (Захватчик ставок)

Этот компонент захватывает ставки как от организатора, так и от участников торгов.

`BidTracker` (Отслеживатель ставок)

Отслеживает ставки и действует как система записи.

`AuctionSession` (Сеанс аукциона)

Запускает и останавливает аукцион. Когда участник торгов закрывает аукцион, предпринимает шаги по выполнению и разрешению платежа, включая уведомление участников торгов о завершении аукциона.

`Payment` (Платеж)

Сторонний платежный сервис для расчетов по кредитным картам.

Согласно блок-схеме на рис. 8.8, после исходного выявления компонентов архитектор на следующем шаге должен проанализировать свойства архитектуры, чтобы решить, надо ли менять дизайн. Для данной системы архитектор определенно может выявить различные наборы архитектурных свойств. Например, в текущем дизайне есть компонент `BidCapture`, предназначенный для захвата ставок как от участников торгов, так и от организатора, что имеет смысл с точки зрения функциональности: захват ставок от кого бы то ни было может обрабатываться одинаково. А как при этом обстоят дела с архитектурными свойствами, относящимися к захвату ставок? Организатору не нужен такой же уровень масштабируемости или адаптируемости, как тысячам участников торгов. К тому же архитектор обязан гарантировать, что такие архитектурные свойства, как надежность (подключения не должны обрываться) и доступность (система должна сохранять работоспособность), для организатора должны быть обеспечены на

более высоком уровне, чем для других частей системы. Для бизнеса, конечно, плохо, если участник торгов не сможет авторизоваться на сайте или пострадает от разорванного подключения, но если любая из этих неприятностей случится с организатором, для аукциона это станет катастрофой.

Поскольку здесь явно прослеживаются разные уровни архитектурных свойств, архитектор решает разбить компонент `BidCapture` на `BidCapture` и `AuctioneerCapture`, чтобы каждый из этих двух компонентов мог поддерживать разные архитектурные свойства. Обновленный дизайн показан на рис. 8.11.

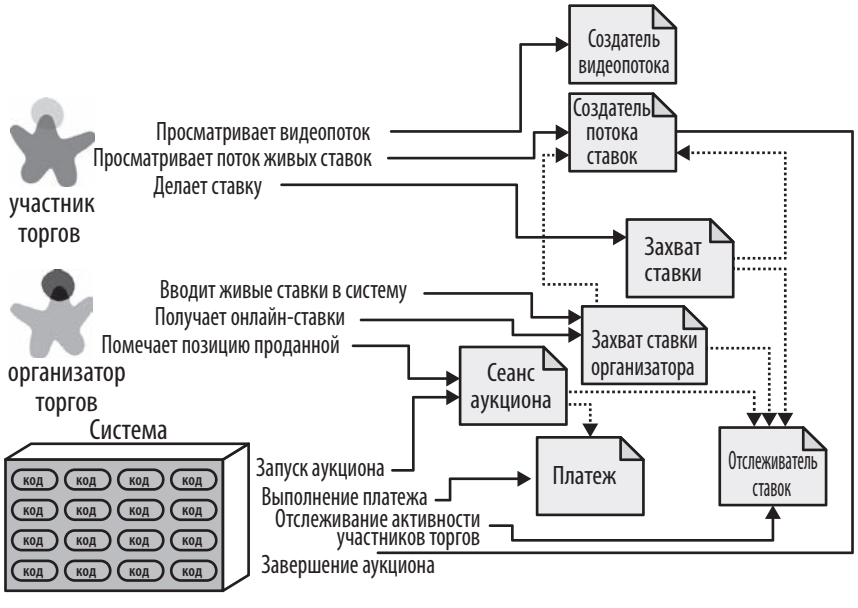


Рис. 8.11. Включение архитектурных свойств в дизайн компонентов GGG

Как видно из рис. 8.11, архитектор создает новый компонент — захват ставки организатора (`AuctioneerCapture`) и обновляет информационные ссылки как на `BidStreamer` (чтобы онлайн-участники аукциона видели живые ставки), так и на `BidTracker`, управляющего потоками ставок. Следует заметить, что теперь `BidTracker` является компонентом, объединяющим два совершенно разных потока информации: один информационный поток от организатора и несколько потоков от участников аукциона.

Дизайн, показанный на рис. 8.11, вряд ли можно признать окончательным вариантом. Еще должны быть раскрыты и дополнительные требования (в которых указано, как люди проходят регистрацию, как администрируются платежи

и т. д.). И все же этот пример — отправная точка для дальнейшего последовательного уточнения дизайна.

Это всего лишь один из возможных наборов компонентов для решения задач GGG, но он не единственный и не факт, что правильный. Редко у каких программных систем есть единственный способ их реализации, и у каждого дизайна имеется свой собственный набор компромиссов. Архитектор не должен зацикливаться на поиске единственно верного дизайна, поскольку приемлемыми могут быть и другие (и возможно, они будут проще). Лучше постараться объективно оценить компромиссы, присущие разным решениям, и выбрать наименее худший вариант.

Архитектурный квант. Выбор между монолитной и распределенной архитектурой

Возвращаясь к определению архитектурного кванта (см. главу 7, раздел «Архитектурные кванты и гранулярность», с. 125), вспомним, что архитектурный квант ограничивает область действия архитектурных свойств. Поэтому архитектор после завершения первоначального проектирования должен ответить на важный вопрос: архитектура должна быть монолитной или распределенной?

Монолитная архитектура представляет собой единый развертываемый модуль, подключенный, как правило, к единой базе данных и включающий все функциональные возможности системы. Типы монолитных архитектур, включая многоуровневый и модульный монолит, подробно рассматриваются в главе 10. *Распределенная* архитектура является полной противоположностью монолитной: приложение состоит из нескольких сервисов, запущенных в собственных экосистемах и обменивающихся данными через сетевые протоколы. У распределенных архитектур более детализированные модели развертывания, в которых каждый сервис может иметь свою собственную периодичность релизов и приемы создания согласно предпочтениям команды разработчиков.

Каждый архитектурный стиль предлагает множество компромиссов, рассматриваемых в части II книги. Но основное решение зависит от того, сколько архитектурных квантов будет выявлено в процессе проектирования. Если системе достаточно одного кванта (иными словами, одного набора архитектурных свойств), то лучшим выбором будет монолитная архитектура. Наоборот, различающиеся архитектурные свойства компонентов, как было показано в ходе анализа компонентов для GGG, требуют применения распределенной архитектуры. Например, как `VideoStreamer`, так и `BidStreamer` предлагают

участникам аукциона его просмотр в режиме только для чтения. С точки зрения дизайна архитектору вряд ли захочется связываться с предназначенной только для чтения потоковой передачей данных в сочетании с широкомасштабными обновлениями. Наряду с вышеупомянутыми различиями между участником торгов и организатором, эта разница в свойствах подталкивает архитектора к выбору распределенной архитектуры.

Одним из преимуществ использования архитектурных квантов для анализа архитектурных свойств и связанности является возможность определения стиля архитектуры (монолитная или распределенная) на ранней стадии процесса проектирования.

ЧАСТЬ II

Архитектурные стили

Разница между архитектурным стилем и архитектурным паттерном может сбивать с толку. Мы определяем *архитектурный стиль* как всеобъемлющую структуру организации исходного кода пользовательского интерфейса и бэкенда (например, в виде уровней монолитного развертывания или же в виде автономно развертываемых сервисов), а также взаимодействие этого кода с хранилищем данных. *Архитектурные паттерны*, в свою очередь, представляют собой низкоуровневые структуры проектирования, помогающие сформировать конкретные решения в рамках архитектурного стиля (они, к примеру, показывают, как достичь высокого уровня масштабируемости или производительности в пределах набора операций или между наборами сервисов).

Начинающие архитекторы должны потратить немало времени и усилий для изучения архитектурных стилей. Необходимо разбираться в стилях и свойственных каждому из них компромиссах, чтобы делать правильный выбор для каждой конкретной бизнес-задачи.

ГЛАВА 9

Архитектурные стили. Основы

Архитектурные стили (которые иногда называют архитектурными паттернами) описывают взаимоотношение компонентов, охватывающее множество архитектурных свойств. Названия архитектурных стилей, по аналогии с названиями паттернов проектирования, стандартны и используются в среде опытных архитекторов в качестве условных обозначений. Например, когда упоминается многоуровневый монолит, собеседнику сразу понятны аспекты структуры, хорошо (или плохо) работающие архитектурные свойства, типичные модели развертывания, стратегии работы с данными, а также целый ряд другой информации. Следовательно, архитекторы должны быть знакомы со стандартными названиями основных общепринятых архитектурных стилей.

В каждом названии заложено множество конкретных особенностей, что и является одной из причин применения паттернов проектирования. Архитектурный стиль описывает топологию, предполагаемые и присущие ему по умолчанию архитектурные свойства, как полезные, так и вредные. В части II книги будут рассмотрены многие распространенные паттерны современной архитектуры. Но архитекторы должны быть знакомы с рядом базовых моделей, из которых состоят более крупные паттерны.

Базовые паттерны

Несколько базовых паттернов используются на протяжении всей истории разработки архитектуры ПО, потому что они обеспечивают удобство организации кода, методов развертывания или других аспектов архитектуры. Например, концепция уровней, разделения различных задач по функциональности, так же стара, как и само программное обеспечение. Тем не менее многоуровневая модель продолжает появляться в различных формах, в том числе и современных, которые будут рассмотрены в главе 10.

Большой ком грязи

Отсутствие какой-либо четко распознаваемой архитектурной структуры архитекторы называют *Big Ball of Mud*, то есть большим комом грязи, по названию одноименного антипаттерна, описание которого было приведено в 1997 году в статье Брайана Фута (Brian Foote) и Джозефа Йодера (Joseph Yoder):

> *Big Ball of Mud* (*Большой ком грязи*) — хаотично разбросанные, неряшливые, перепутанные, наспех скрепленные изолентой и проволокой джунгли из спагетти-кода. Такая система склонна к нерегулируемому росту и постоянно требует правок. Информация распределяется между удаленными элементами системы беспорядочно до такой степени, что все важные сведения становятся общедоступными или же продублированными.
>
> Возможно, общая структура такой системы никогда не была четко сформулирована.
>
> Если же когда-то и было четкое определение, то оно уже давно изменилось до неузнаваемости. Программисты, у которых есть хотя бы малая толика архитектурного мышления, сторонятся этого болота. Работа над такими системами устраивает только безразличных к архитектуре и, возможно, тех, кто чувствует себя вполне комфортно, инертно следуя повседневной рутине латания дыр в этих постоянно размываемых дамбах.

Сейчас *Большой ком грязи* может подразумевать простое приложение, не имеющее реальной внутренней структуры и выполненное в виде сценария с обработчиками событий, напрямую подключенными к вызовам базы данных. С этого начинаются многие примитивные приложения, которые, разрастаясь, становятся трудноуправляемыми.

Как правило, архитекторы стремятся любой ценой избежать создания подобного типа архитектуры из-за проблем с развертыванием, тестированием, масштабированием и производительностью. К тому же отсутствие структуры существенно усложняет внесение изменений.

К сожалению, в реальной жизни данный антипаттерн встречается с завидной регулярностью. Вряд ли кто-либо из архитекторов намеренно стремится к его созданию, но из-за отсутствия контроля качества кода и его структуры во многих проектах неизбежно возникает беспорядок. Например, Нилу как-то приходилось работать с клиентским проектом, структура которого показана на рис. 9.1.

Клиент (чье имя по понятным причинам не разглашается) в течение нескольких лет создавал веб-приложение на Java. Архитектурная связанность демонстриру-

ется следующим образом[1]: каждая точка на окружности представляет собой класс, а каждая линия — связь между классами, где более жирные линии показывают наиболее тесные связи. В этой кодовой базе при любом изменении в классе почти невозможно предвидеть бесчисленные побочные эффекты, оказывающие влияние на другие классы; внесение изменений становится пугающей перспективой.

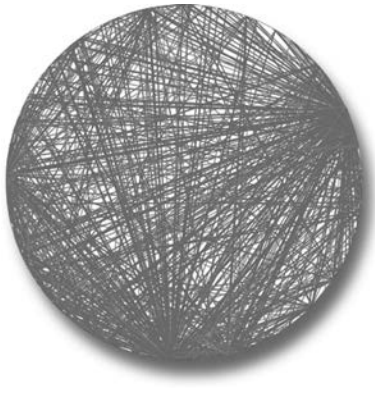


Рис. 9.1. Визуализация архитектуры в стиле Большой ком грязи на примере реальной кодовой базы

Цельная архитектура

На заре развития программного обеспечения существовал только компьютер, и поэтому код запускался только на нем. Аппаратное и программное обеспечение представляли собой единое целое, но затем, по мере усложнения выполняемых задач, они разделились. Например, мейнфреймы сперва выступали в виде сингулярных систем, а затем выделили данные в отдельные системы. Точно так же при появлении первых персональных компьютеров основная часть коммерческих разработок была сконцентрирована на применении отдельно взятых машин. Когда объединение персональных компьютеров в сеть стало обычным явлением, появились распределенные системы (такие, как клиент-сервер).

Цельные архитектуры сохранились разве что во встроенных системах и в других средах с жесткими ограничениями. Как правило, функциональные возможности программных систем со временем возрастают, что требует разделения ответственности для поддержания таких эксплуатационных архитектурных свойств, как производительность и масштабируемость.

---

[1] Выполнено с помощью ныне вышедшего из употребления инструмента под названием XRay, дополнительного модуля Eclipse.

Клиент-сервер

Со временем в силу различных причин потребовалось уйти от цельной системы. Способы реализации разделения системы стали основой многих рассматриваемых нами стилей. В большинстве архитектурных стилей определены методы эффективного разбиения системы.

Базовый стиль в архитектуре разделяет техническую функциональность на интерфейсную (frontend) и серверную, или фронтенд и бэкенд. При этом создается так называемая *двухуровневая*, или *клиент-серверная*, архитектура. В зависимости от требований и вычислительных возможностей существует множество различных вариантов подобной архитектуры.

Стационарный компьютер + сервер базы данных

Ранняя архитектура персональных компьютеров вдохновляла разработчиков на создание многофункциональных десктопных приложений в пользовательских интерфейсах, таких как Windows, с сохранением данных на отдельном сервере базы данных. Появление этой архитектуры совпало по времени с возникновением автономных серверов баз данных, к которым можно было подключаться по стандартным сетевым протоколам. Это позволяло размещать логику представления на стационарном компьютере, а более интенсивные вычислительные действия (как по объему, так и по сложности) выполнять на более надежных серверах баз данных.

Браузер + веб-сервер

С появлением современной веб-разработки в практику вошло разделение на подключенные друг к другу веб-браузер и веб-сервер (который, в свою очередь, был подключен к серверу базы данных). Такое разделение ответственности было похоже на предыдущий вариант, но с более тонкими клиентами — браузерами, что допускало их более широкое распределение как внутри, так и за пределами межсетевых экранов. Несмотря на разделение базы данных и веб-сервера, архитекторы зачастую продолжают считать эту архитектуру двухуровневой, поскольку веб-сервер и сервер базы данных работают в операционном центре на одном классе машин, а пользовательский интерфейс — в браузере пользователя.

Трехуровневые системы

В конце 1990-х годов весьма популярной стала *трехуровневая архитектура*, предоставившая еще больше разделительных уровней. Как только в Java и .NET приобрели популярность такие инструментальные средства, как серверы при-

ложений, компании начали создавать в своих топологиях еще больше уровней: уровень базы данных, использующий серверы баз данных промышленного класса, уровень приложения, управляемый сервером приложений, уровень пользовательского интерфейса, который выполняется в сгенерированном HTML-коде и все чаще — в коде JavaScript.

Трехуровневая архитектура соответствовала таким протоколам сетевого уровня, как Common Object Request Broker Architecture, CORBA (общая архитектура брокера объектных запросов)[1], и Distributed Component Object Model, DCOM[2] (распределенная модель объектных компонентов), которые позволили упростить выстраивание распределенных архитектур.

ТРЕХУРОВНЕВАЯ АРХИТЕКТУРА, РАЗРАБОТКА ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И ДОЛГОСРОЧНЫЕ ПОСЛЕДСТВИЯ

Во времена разработки языка Java трехуровневые вычисления были в моде. Считалось, что в дальнейшем все системы станут трехуровневыми. Типичным неудобством существующих на тот момент языков, таких как C++, был громоздкий процесс последовательного перемещения объектов по сети между системами. Поэтому разработчики языка Java решили встроить эту возможность в ядро самого языка, воспользовавшись приемом, названным *сериализацией*. Каждый объект `Object` в Java реализует интерфейс, который требует поддержки сериализации. Разработчики считали, что поскольку в будущем останется только трехуровневая архитектура, ее встраивание в язык будет отличной идеей. В итоге этот архитектурный стиль появился и исчез, но его детали остаются в Java и по сей день, сильно раздражая разработчиков языка, поскольку чтобы добавить новые современные функции, нужна поддержка уже устаревшей сериализации для сохранения обратной совместимости.

Крайне сложно предсказать долгосрочные последствия проектных решений, как при разработке программных систем, так и в других инженерных дисциплинах. Неизменный совет в большинстве случаев — отдавать предпочтение простым конструкциям — будет хорошей профилактикой защиты от возможных рисков.

[1] https://www.corba.org/

[2] https://ru.wikipedia.org/wiki/DCOM

Вряд ли архитекторам в наши дни стоит беспокоиться об этом уровне подключения в распределенных архитектурах, и здесь их можно сравнить с разработчиками, которых абсолютно не волнует, как именно работают такие сетевые протоколы, как TCP/IP (достаточно знать, что они работают). Возможности, предлагавшиеся соответствующими инструментами в то время, имеются и сегодня — либо в виде инструментов (подобных очередям сообщений), либо в виде архитектурных паттернов (в качестве примера можно привести архитектуру, управляемую событиями, рассматриваемую в главе 14).

Сравнение монолитной и распределенной архитектуры

Архитектурные стили можно подразделить на два основных типа: *монолитные* (с единым развертываемым блоком для всего кода) и *распределенные* (с несколькими развертываемыми блоками, подключаемыми друг к другу по протоколам удаленного доступа). Хотя ни одна классификация не является идеальной, всем распределенным архитектурам свойствен один и тот же набор сложностей и проблем, не встречающийся в монолитных архитектурных стилях, что делает указанную классификационную схему вполне приемлемой.

В этой книге нам предстоит подробный разбор следующих архитектурных стилей:

Монолитные

- многоуровневая архитектура (глава 10);
- конвейерная архитектура (глава 11);
- микроядерная архитектура (глава 12).

Распределенные

- архитектура на основе сервисов (глава 13);
- архитектура, управляемая событиями (глава 14);
- архитектура на основе пространства (глава 15);
- сервис-ориентированная архитектура (глава 16);
- архитектура микросервисов (глава 17).

Распределенные архитектурные стили хоть и мощнее монолитных в плане производительности, масштабируемости и доступности, но расплачиваются за это ощутимыми компромиссами. Первая группа проблем, присущих всем распределенным архитектурам, была рассмотрена в виде *заблуждений в сфере распределенных вычислений*[1], впервые предложенных Л. Питером Дойчем (L. Peter Deutsch) и его коллегами из компании Sun Microsystems в 1994 году. Под *заблуждением* понимается то, во что ошибочно верят или полагают истинным. Все восемь заблуждений о распределенных вычислениях применимы сегодня к распределенным архитектурам. Каждое из них рассматривается в следующих разделах.

Заблуждение № 1: надежность сети

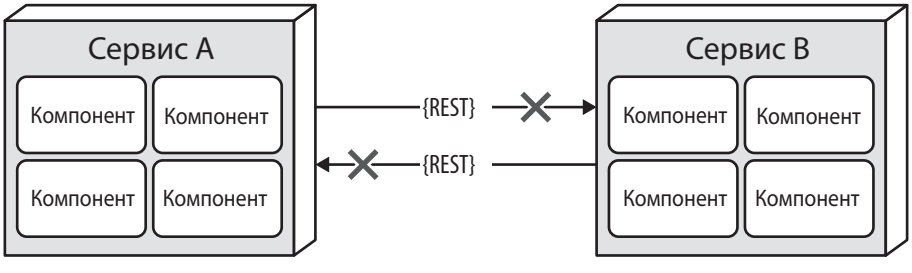


Рис. 9.2. Сеть ненадежна

Разработчики и архитекторы уверены, что сеть надежна, но это не так. Хотя со временем, конечно, ситуация улучшается, дело в том, что абсолютно надежных сетей не бывает. Это важно для всех распределенных архитектур, поскольку все подобные стили зависят от сети в передаче данных от сервиса, к сервису или между сервисами.

На рис. 9.2 показано, что с `Сервисом B` может быть все в порядке, но `Сервис A` не может с ним связаться из-за сетевого сбоя; или, хуже того, `Сервис A` отправил запрос `Сервису B` на обработку неких данных и не получил ответа из-за неполадок в сети. Именно поэтому между сервисами существуют тайм-ауты и автоматическое выключение (circuit breakers). Чем больше система зависит от сети (например, при использовании архитектуры микросервисов), тем менее надежной она становится.

[1] https://oreil.ly/fVAEY

Заблуждение № 2: нулевая задержка

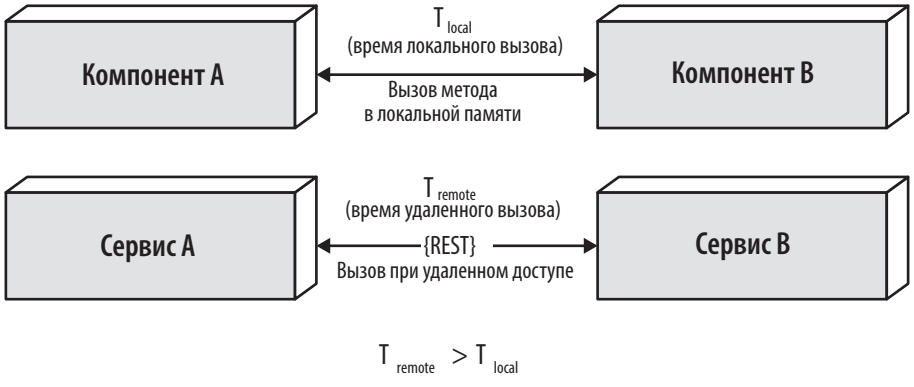


Рис. 9.3. Задержка не нулевая

На рис. 9.3 показано, что при локальном вызове другого компонента посредством вызова метода или функции время этого вызова (`t_local`) измеряется нано- или микросекундами. Но когда точно такой же вызов делается через протокол удаленного доступа (например, через REST, обмен сообщениями или RPC), время доступа к сервису (`t_remote`) измеряется миллисекундами. Поэтому `t_remote` всегда будет больше, чем `t_local`. Задержка в любой распределенной архитектуре отлична от нуля, но большинством архитекторов этот факт игнорируется, потому что в их распоряжении якобы имеются очень быстрые сети. Задайтесь вопросом: известно ли вам значение среднего времени задержки полного цикла RESTful-вызова в вашей производственной среде? 60 мс? Или же 500 мс?

При использовании любой распределенной архитектуры необходимо знать среднее время задержки. Это единственный способ определения пригодности распределенной архитектуры, особенно при выборе микросервисов из-за высокой степени детализации сервисов и объема обмена данными между ними. Если предположить, что среднее время задержки при каждом запросе составит 100 мс, то объединение десяти сервисов для выполнения конкретной бизнес-функции добавит к запросу 1000 мс! Величину среднего времени задержки знать, конечно, важно, но не менее важно знать и значение с 95-го по 99-й процентиль. Среднее время задержки может составить всего лишь 60 мс (что само по себе и неплохо), а вот значение 95-го процентиля может быть 400 мс! Обычно именно такой «длинный хвост» задержки и убивает производительность в распределенной архитектуре. В большинстве случаев архитекторы могут узнать значение задержки у администратора сети (см. далее раздел «Заблуждение № 6: сетью всегда занимается только один администратор» на с. 167).

Заблуждение № 3: пропускная способность ничем не ограничена

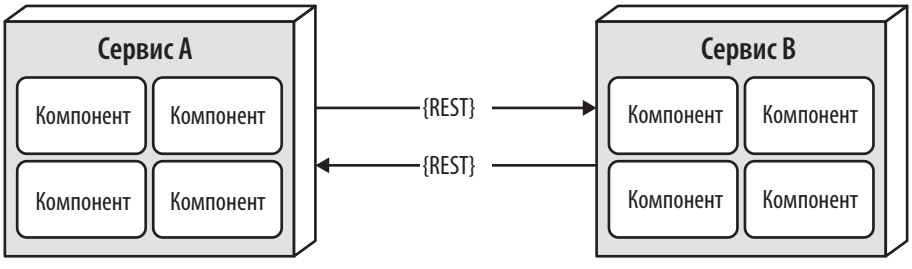


Рис. 9.4. Пропускная способность небесконечна

Пропускная способность обычно не является проблемой для монолитных архитектур, поскольку когда вычисления переходят в монолит, для обработки бизнес-запроса практически не требуется никакая пропускная способность. Но как показано на рис. 9.4, как только системы в распределенной архитектуре вроде микросервисов разбиваются на более мелкие единицы развертывания (сервисы), обмен данными между этими сервисами значительно увеличивает требуемую пропускную способность, что замедляет работу сетей и оказывает за счет этого влияние на время задержки (см. заблуждение № 2) и на надежность (см. заблуждение № 1).

Чтобы показать, насколько важно избавиться от рассматриваемого заблуждения, обратим внимание на два сервиса, показанные на рис. 9.4. Предположим, левый сервис управляет вишлистом веб-сайта, а правый — профилями клиентов. Как только левый сервис получит запрос на выдачу вишлиста, ему понадобится сделать межсервисный запрос к правому сервису, занимающемуся профилями, чтобы получить имя клиента для ответного контракта на пункт из вишлиста, а у левого сервиса вишлиста нет связки с именем. Сервис профилей клиентов возвратит сервису вишлиста 45 атрибутов общим объемом 500 Кбайт, хотя требуется получить только имя (200 байт). Это так называемая *связанность модулей по образцу (stamp coupling)*.

Казалось бы, ничего страшного, но запросы пунктов из вишлиста поступают примерно 2000 раз в секунду. Следовательно, этот межсервисный вызов от сервиса вишлиста к сервису профилей пользователей происходит 2000 раз в секунду. При 500 Кбайт для каждого запроса объем задействованной на один только межсервисный вызов пропускной способности (из сотен сделанных в эту же секунду) составляет 1 Гбайт!

Связанность по образцу в распределенной архитектуре подразумевает весьма существенную ширину пропускной способности. Если бы сервису профилей клиентов потребовалось передавать обратно только те данные, которые нужны сервису вишлиста (в данном случае 200 байт), общее потребление пропускной способности на передачу данных составило бы всего 400 Кбайт. Проблема связанности по образцу может быть решена следующими способами:

- созданием закрытых конечных точек API-интерфейса RESTful;
- использованием в контракте селекторов полей;
- использованием языка GraphQL для разделения контрактов;
- использованием контрактов, ориентированных на ценности (value-driven contracts), вместе с контрактами, ориентированными на потребителя (consumer-driven contracts, CDC);
- использованием внутренних конечных точек обмена сообщениями.

Независимо от применяемой технологии, лучшим способом избавления от этого заблуждения станет обеспечение в распределенной архитектуре передачи минимального объема данных между сервисами и системами.

Заблуждение № 4: сеть безопасна

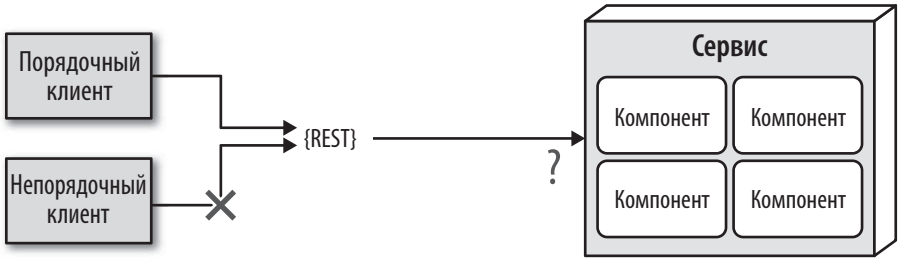


Рис. 9.5. Сеть небезопасна

Большинство архитекторов и разработчиков слишком привыкли использовать виртуальные частные сети (virtual private network, VPN), надежные сети (trusted networks) и брандмауэры и стали забывать о том, что *сеть небезопасна*. Еще сложнее добиться нужной степени безопасности в распределенной архитектуре. Как показано на рис. 9.5, все без исключения конечные точки каждой единицы развертывания должны иметь такую степень безопасности, чтобы до конкретного сервиса не смогли добраться никакие неизвестные или вредоносные запросы.

Территория угроз и атак при переходе от монолитной к распределенной архитектуре увеличивается в разы. Еще одной причиной снижения производительности в синхронных, сильно распределенных архитектурах типа микросервисов или в архитектурах на основе сервисов является необходимость применения особых мер безопасности к каждой конечной точке, даже если речь идет об обмене данными между сервисами.

Заблуждение № 5: топология никогда не меняется

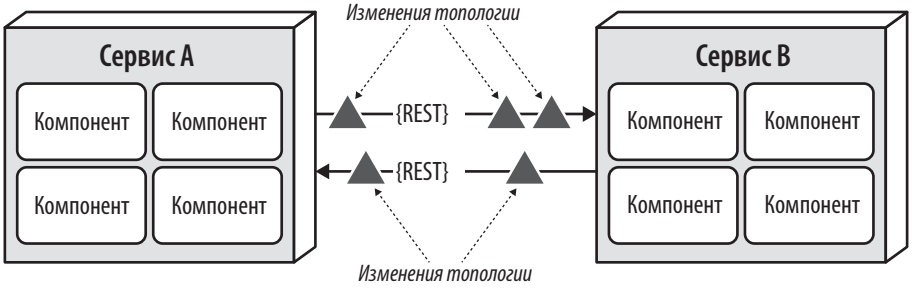


Рис. 9.6. Сетевая топология постоянно меняется

Это заблуждение относится к общей топологии сети, включая все маршрутизаторы, концентраторы, коммутаторы, брандмауэры, сети и устройства, используемые в общей сети. Архитекторы полагают, что топология налажена и никогда не меняется. Разумеется, это не так: *она постоянно меняется*. В чем же особая опасность этого заблуждения?

Представьте, что архитектор приходит в понедельник утром на работу, а там все носятся как угорелые, поскольку сервисы простаивают в нерабочем состоянии. Архитектор вместе с командами лихорадочно пытается выяснить причину происходящего. За неделю не было развернуто ни одного нового сервиса. Так что же это может быть? Через несколько часов выясняется, что в два часа ночи произошло небольшое обновление сети. И это якобы «небольшое» обновление обрушило все предположения о задержках, вызывая тайм-ауты и срабатывание автоматических выключателей.

Архитекторы должны постоянно контактировать с операторами и администраторами сети, чтобы знать, что и когда изменилось, и иметь возможность своевременно внести корректировки и предотвратить неожиданности, подобные истории выше. Казалось бы, здесь все просто и понятно, но это не так. По сути, данное заблуждение напрямую ведет нас к следующему заблуждению.

Заблуждение № 6: сетью всегда занимается только один администратор

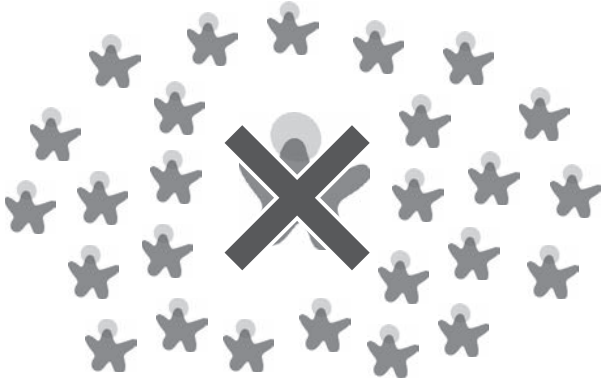


Рис. 9.7. Сетевой администратор не один, их всегда много

Архитекторы постоянно впадают в это заблуждение, предполагая, что им необходимо сотрудничать и поддерживать контакт только с одним администратором. На рис. 9.7 показано, что в обычной крупной компании задействованы десятки сетевых администраторов. С кем именно архитектору нужно общаться по поводу задержки (см. выше раздел «Заблуждение № 2: нулевая задержка» на с. 163) или изменений топологии (см. выше раздел «Заблуждение № 5: топология никогда не меняется» на с. 166)? Это заблуждение указывает на сложность распределенной архитектуры и на объем необходимых координационных усилий, чтобы всё и всегда работало правильно. Монолитным приложениям ввиду однократного развертывания не требуется такой уровень взаимодействия и сотрудничества.

Заблуждение № 7: передача данных ничего не стоит

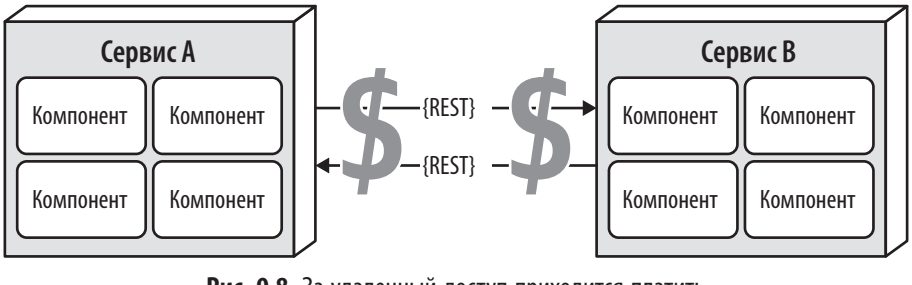


Рис. 9.8. За удаленный доступ приходится платить

Многие архитекторы ПО путают это заблуждение с задержкой (см. выше раздел «Заблуждение № 2: нулевая задержка» на с. 163). Здесь расходы на передачу данных связаны не с задержкой по времени, а с фактическими затратами в денежном выражении, связанными с совершением «простого RESTful-вызова». Архитекторы ошибочно считают, что вся необходимая инфраструктура уже имеется в наличии и ее вполне достаточно для совершения простого RESTful-вызова или для разбиения монолитного приложения на части. *Чаще всего это далеко от истины*. Распределенные архитектуры обходятся намного дороже монолитных, в первую очередь за счет возросших потребностей в дополнительном оборудовании, серверах, шлюзах, брандмауэрах, новых подсетях, прокси-серверах и т. д.

При переходе к распределенной архитектуре мы рекомендуем архитекторам провести анализ текущей серверной и сетевой топологии с точки зрения емкости, пропускной способности, задержек и зон безопасности, чтобы не попасть в западню этого заблуждения.

Заблуждение № 8: сеть однородна

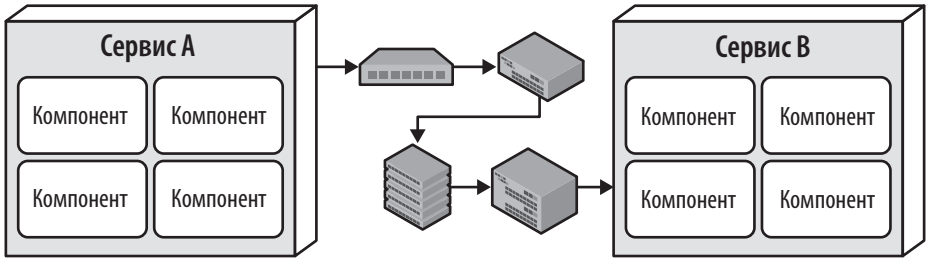


Рис. 9.9. Сеть неоднородна

Большинство архитекторов и разработчиков предполагают, что будут иметь дело с однородной сетью, составленной из оборудования только одного производителя. Ничто не может быть так далеко от истины. В инфраструктуре большинства компаний используется сетевое оборудование от нескольких поставщиков.

И что с того? Суть рассматриваемого заблуждения в том, что не все оборудование от разных производителей способно на успешную совместную работу. В основном особых проблем не возникает, но легко ли оборудование Juniper интегрируется с оборудованием Cisco? С годами сетевые стандарты совершенствовались, снижая остроту проблемы, но фактически полного тестирования всех ситуаций, рабочих нагрузок и различных обстоятельств не проводилось, и поэтому сетевые пакеты иногда теряются. Это, в свою очередь, оказывает

влияние на надежность сети (см. выше раздел «Заблуждение № 1: надежность сети» на с. 162), на предположения и утверждения о времени задержек (см. выше раздел «Заблуждение № 2: нулевая задержка» на с. 163) и на предположения насчет пропускной способности (см. выше раздел «Заблуждение № 3: пропускная способность ничем не ограничена» на с. 164). Иными словами, это заблуждение связано со всеми другими заблуждениями, создавая бесконечный цикл путаниц и разочарований при работе с сетями (без чего не обойтись при использовании распределенных архитектур).

Другие соображения насчет распределенных архитектур

Вдобавок к рассмотренным ранее восьми заблуждениям, у распределенных архитектур существуют другие проблемы и сложности, не встречающиеся у монолитных. Хотя подробное изучение этих вопросов выходит за рамки данной книги, они будут перечислены и обобщены в следующих разделах.

Распределенное журналирование

Определять первопричины потери заказа в распределенных архитектурах крайне сложно и затратно по времени из-за распределенного ведения журналов приложения и журналов системы. В монолитных приложениях ведется, как правило, один журнал, что упрощает отслеживание запроса и выявление проблемы. А распределенные архитектуры содержат десятки, если не сотни различных журналов, и все они находятся в разных местах и ведутся в разных форматах, затрудняя отслеживание ошибок.

Объединять информацию из различных источников и систем в одном общем журнале для того, чтобы просматривать ее на одной консоли, помогают такие инструменты консолидации журналов, как Splunk[1], но это затрагивает лишь поверхностный слой проблем. Подробное описание решений и паттернов для распределенного журналирования выходит за рамки этой книги.

Распределенные транзакции

В мире монолитной архитектуры транзакции воспринимаются архитекторами и разработчиками как само собой разумеющееся явление, поскольку они просты и легки в управлении. Стандартные коммиты (`commits`) и откаты (`rollbacks`), выполняемые из фреймворков сохранения, используют ACID-транзакции (аббревиатура ACID означает атомарность, согласованность, изолированность

[1] https://www.splunk.com/

и стойкость), гарантирующие обновление данных корректным образом, обеспечивая их высокую согласованность и целостность. В распределенных архитектурах дела обстоят совершенно иначе.

В распределенных архитектурах полагаются на так называемую *конечную согласованность (eventual consistency)*, чтобы гарантировать, что данные, обработанные отдельно взятыми единицами развертывания, к некому моменту времени синхронизируются до согласованного состояния. Это один из компромиссов распределенной архитектуры: высокая масштабируемость, производительность и доступность предоставляются в ущерб согласованности и целостности данных.

Одним из способов управления распределенными транзакциями является применение *транзакционных саг (transactional sagas)*. Саги используют либо выдачу событий для компенсации, либо конечные автоматы для управления состоянием транзакции. В дополнение к сагам используются BASE-транзакции. BASE означает (B)asic availability (базовая доступность), (S)oft state (гибкое состояние) и (E)ventual consistency (конечная согласованность). BASE-транзакции являются скорее технологией, а не частью программного обеспечения. *Гибкое состояние* (*soft state*) в BASE относится к передаче данных от источника к цели, а также к несогласованности между источниками данных. Основываясь на базовой доступности задействованных систем или сервисов, системы в конечном итоге приобретают согласованность за счет использования архитектурных паттернов и обмена сообщениями.

Сопровождение контрактов и управление их версиями

Еще одной особенно сложной задачей в распределенной архитектуре является создание контрактов, их сопровождение и управление их версиями. Контракт — это поведение и данные, согласованные как клиентом, так и сервисом. Сопровождение контрактов в распределенных архитектурах составляет особую сложность, в первую очередь из-за несвязанных сервисов и систем, принадлежащих разным командам и отделам. Еще более сложными являются коммуникационные модели, необходимые для прекращения поддержки версий.

ГЛАВА 10

Многоуровневая архитектура

Многоуровневая архитектура, известная также под названием *n-уровневого* архитектурного стиля, — один из самых распространенных архитектурных подходов. Фактически этот стиль является стандартом для большинства приложений, в первую очередь благодаря своей простоте, известности и невысоким затратам. К тому же это очень удобно для разработчиков, потому что, согласно закону Конвея, в приложении повторяется структура взаимодействия внутри компании. В большинстве организаций команды разработчиков состоят из разработчиков пользовательского интерфейса, разработчиков серверной части, разработчиков правил и специалистов по работе с базами данных. Эти организационные уровни хорошо вписываются в структуру традиционной многоуровневой архитектуры и используются уже в самой разработке многих бизнес-приложений. При многоуровневом проектировании также есть риск использования ряда архитектурных антипаттернов, включая антипаттерн *подразумеваемой архитектуры* (*architecture by implication*) и антипаттерн *бессистемной архитектуры* (*accidental architecture*). Если разработчик или архитектор не определился с используемым архитектурным стилем или же Agile-команда «только-только начинает кодирование», велика вероятность, что они выберут стиль многоуровневой архитектуры.

Топология

Компоненты в многоуровневой архитектуре собраны в логические горизонтальные уровни, каждый из которых выполняет в приложении вполне определенную роль (например, реализует интерфейс или бизнес-логику). Конкретных ограничений в отношении количества и типов применяемых уровней не существует, но как показано на рис. 10.1, большинство таких архитектур состоят из четырех стандартных уровней: представления, бизнес-логики, сохранения информации и базы данных. В некоторых случаях уровень бизнес-логики и уровень сохране-

ния информации объединяются в один бизнес-уровень, особенно когда логика сохранения (например, SQL или HSQL) встроена в компоненты уровня бизнес-логики. Поэтому у небольших приложений может быть только три уровня, а большие и более сложные бизнес-приложения могут состоять из пяти и более.

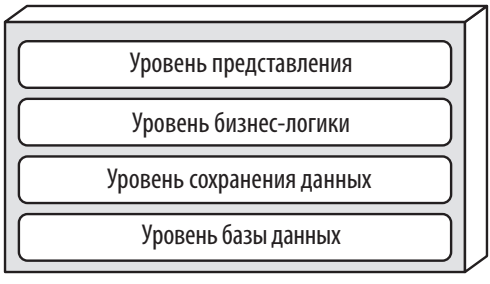


Рис. 10.1. Стандартные логические уровни в архитектуре многоуровневого стиля

На рис. 10.2 показаны различные варианты топологии с точки зрения физического разбиения на уровни (развертывания). В первом варианте в один развертываемый модуль собраны уровни представления, бизнес-логики и сохранения данных, а уровень базы данных, как правило, представлен в виде отдельной внешней физической базы данных (или файловой системы). Во втором варианте уровень представления выделен в самостоятельный модуль развертывания, а уровни бизнес-логики и сохранения данных объединены в другом. В этом варианте уровень базы данных так же, как и обычно, физически выделен во внешнюю базу данных или файловую систему. В третьем варианте все четыре стандартных уровня, включая и уровень базы данных, объединены в единый

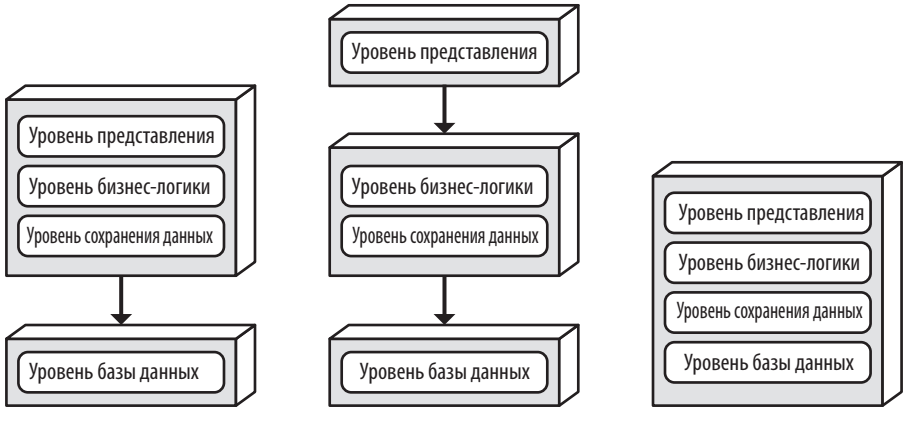


Рис. 10.2. Физические варианты топологии (развертывания)

модуль. Этот вариант может подойти для небольших приложений, использующих либо встроенную базу данных, либо базу данных, размещаемую в оперативной памяти. Многие локально устанавливаемые («on-prem») программные продукты создаются и поставляются клиентам в виде как раз последнего варианта.

У каждого уровня в архитектуре многоуровневого стиля есть своя особая роль и ответственность в рамках этой архитектуры. Уровень представления отвечает за обслуживание пользовательского интерфейса и за логику взаимодействий в браузере, а уровень бизнес-логики занимается выполнением определенных бизнес-правил, связанных с запросами. Каждый архитектурный уровень формирует некую абстракцию действий, которые необходимы для выполнения конкретного бизнес-запроса. К примеру, уровню представления не нужно знать или заботиться о том, как получать данные клиента: его задача ограничивается отображением этой информации на экране в нужном формате. Аналогичным образом уровню бизнес-логики не требуется заниматься форматированием клиентских данных для их отображения на экране или знать, откуда эти данные поступают; его задача — получить данные от уровня сохранения информации, применить к ним бизнес-логику (например, вычислить значения или агрегировать) и передать эту информацию выше на уровень представления.

Эта концепция *разделения ответственности (separation of concerns)* упрощает построение оптимальных ролей и моделей ответственности внутри архитектуры. Компоненты в рамках конкретного уровня ограничены в области действия и работают только с логикой, относящейся к данному уровню. Например, компоненты уровня представления имеют дело только с логикой представления, а компоненты бизнес-уровня — только с бизнес-логикой. Это позволяет разработчикам сфокусироваться на технических аспектах только предметной области (например, на логике представления или логике сохранения данных). Но компромисс, на который приходится идти, добиваясь такого преимущества, выражается в отсутствии общей гибкости (способности быстро реагировать на изменения).

Многоуровневая архитектура относится к архитектурам, *разбиваемым по техническому принципу* (в отличие от *предметного разбиения*). Компоненты объединяются в группы не по предметным областям (например, по клиентам), а по своей технической архитектурной роли (например, по представлению или по бизнес-логике). В результате любая конкретная бизнес-область пронизывает все уровни архитектуры. Например, предметная область «клиент» содержится на уровне презентации, на уровне бизнес-логики, на уровне правил, на уровне сервисов и на уровне базы данных, что затрудняет внесение в нее изменений. Получается, что ко всему прочему в архитектуре многоуровневого стиля не работает и предметно-ориентированное проектирование.

Уровни изоляции

Каждый уровень в многоуровневой архитектуре может быть либо *закрытым*, либо *открытым*. Закрытый уровень означает, что при движении сверху вниз запрос не сможет пропустить ни один уровень, и для того, чтобы перейти дальше, необходимо пройти уровень, непосредственно следующий за текущим (рис. 10.3). Например, в архитектуре с закрытыми уровнями запрос, выданный на уровне представления, прежде чем добраться, наконец, до уровня базы данных, должен сначала пройти уровень бизнес-логики, затем уровень сохранения информации.

Рис. 10.3. Закрытые уровни в составе многоуровневой архитектуры

Заметьте, что для простых запросов на поиск уровню представления, показанному на рис. 10.3, было бы намного быстрее и проще получить доступ к базе данных напрямую, минуя любые ненужные уровни (что в начале 2000-х годов называлось *паттерном быстрого чтения*). Для этого уровни бизнес-логики и сохранения информации должны быть *открытыми* и позволять обходить их на пути к другим уровням. Так что же лучше: открытые или закрытые уровни? Ответ на этот вопрос кроется в ключевой концепции, которая называется *уровнями изоляции (layers of isolation)*.

Концепция *уровней изоляции* состоит в том, что изменения, сделанные на одном уровне архитектуры, как правило, не влияют на компоненты на других уровнях, при условии, что контракты между ними останутся неизменными. Каждый уровень сохраняет независимость от других, вследствие чего располагает незначительными сведениями или же вообще не осведомлен о внутренней работе других уровней архитектуры. Но для поддержки этой концепции уровни, участвующие в основном потоке запроса, должны быть обязательно закрытыми. Если уровень представления может напрямую обращаться к уровню сохранения данных, то

изменения, внесенные в уровень сохранения данных, повлияют как на уровень бизнес-логики, *так и* на уровень представления, в результате чего получится приложение с очень тесной связанностью и с межуровневыми взаимозависимостями компонентов. Из-за этого архитектура подобного типа станет хрупкой, и вносить изменения в нее будет трудно и слишком затратно.

Концепция уровней изоляции также позволяет заменять любой архитектурный уровень, не оказывая при этом влияния на остальные уровни, опять же, при условии четко определенных связей и использования паттерна делегирования (business delegate pattern). Например, концепцией уровней изоляции можно воспользоваться в многоуровневой архитектуре для замены старых уровней представления JavaServer Faces (JSF) на React.js, не затрагивая при этом никаких других уровней приложения.

Добавление уровней

Закрытые уровни облегчают поддержку уровней изоляции и помогают тем самым изолировать изменения внутри архитектуры, но есть причины, по которым некоторым уровням лучше все же быть открытыми. Предположим, к примеру, что на уровне бизнес-логики имеются совместно используемые объекты, содержащие общие функции для бизнес-компонентов (такие, как вспомогательные классы для работы с датами и строками, классы аудита, классы журналирования и т. д.). Предположим, что существует архитектурное решение, согласно которому уровень представления зависит от использования этих общих бизнес-объектов. Эта зависимость показана на рис. 10.4 пунктирной линией, идущей от компонента представления к общему бизнес-объекту на уровне бизнес-логики. Такой сценарий трудно поддается управлению и контролю, поскольку *архитектурно* уровень представления имеет доступ к уровню бизнес-логики и, следовательно, может обращаться к общим объектам внутри этого уровня.

Одним из способов архитектурного закрепления этой зависимости является добавление к архитектуре нового уровня сервисов, содержащего все общие бизнес-объекты. В этом случае ограничивается доступ уровня представления к общим бизнес-объектам, поскольку уровень бизнес-логики будет закрытым (рис. 10.5). Но новый уровень сервисов должен быть помечен как *открытый*: иначе для получения доступа к уровню сохранения данных запросы от бизнес-уровня будет вынуждены проходить через уровень сервисов. Когда уровень сервисов помечен как открытый, уровень бизнес-логики может либо обращаться к нему напрямую (что показано сплошной стрелкой), либо обходить его, сразу направляясь к уровню, расположенному ниже (что на рис. 10.5 показано пунктирной стрелкой).

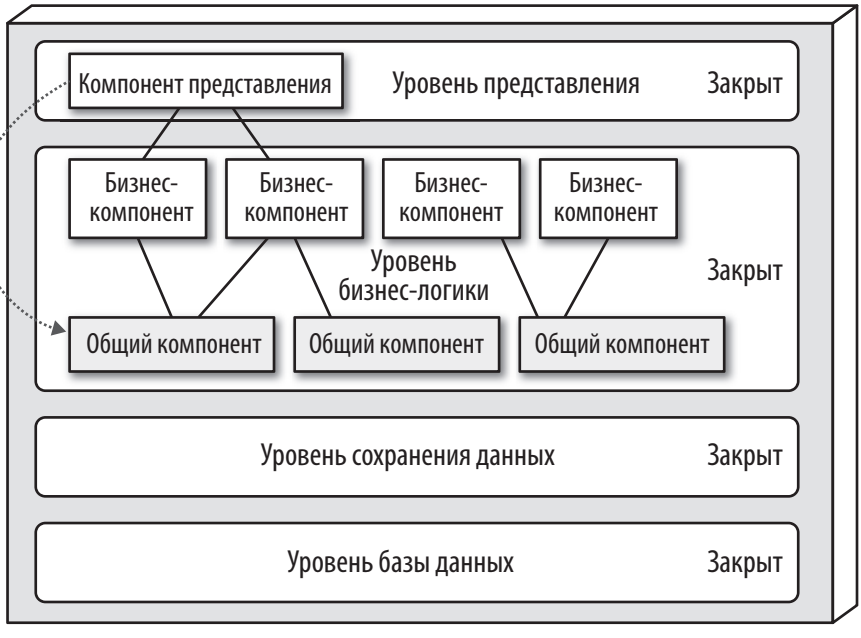


Рис. 10.4. Общие объекты внутри бизнес-уровня

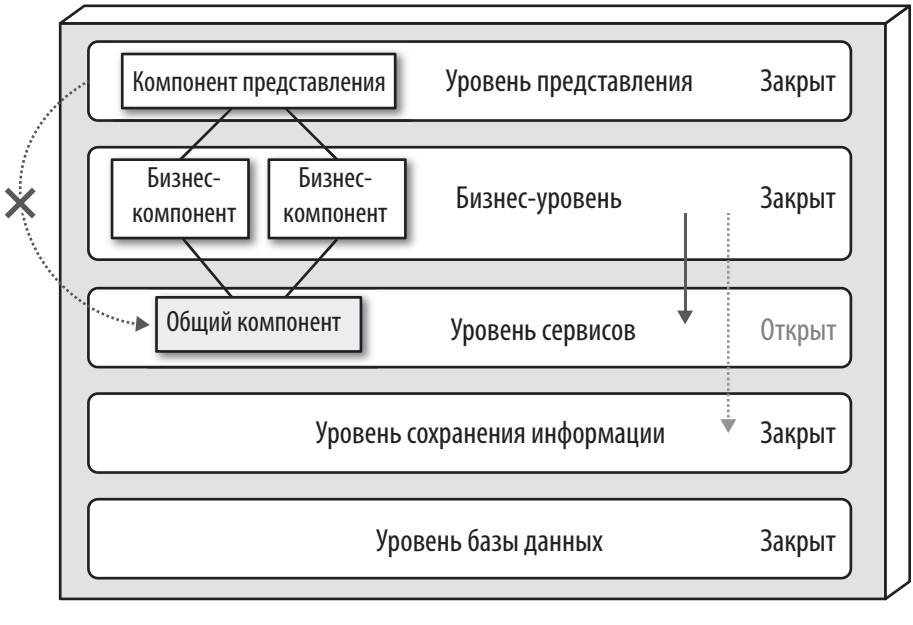


Рис. 10.5. Добавление к архитектуре нового уровня

Использование концепции открытых и закрытых уровней помогает определить взаимосвязь между архитектурными уровнями и потоками запросов. Кроме того, разработчикам предоставляется необходимая информация и ориентиры для понимания имеющихся в архитектуре ограничений доступа к тем или иным уровням. Если не документировать или должным образом не сообщать, какие уровни в архитектуре открыты, а какие закрыты (и почему), то, как правило, это приводит к созданию сильно связанных и хрупких архитектур, плохо поддающихся тестированию, сопровождению и развертыванию.

Другие факторы, заслуживающие внимания

Многоуровневая архитектура является неплохой отправной точкой для большинства приложений, когда еще точно неизвестно, какой в конечном итоге будет использован архитектурный стиль. Обычная практика: архитекторы все еще не определились, стоит ли выбирать микросервисы, а разработка уже должна начаться. Но при использовании данного метода нужно убедиться, что повторное использование кода сведено к минимуму, а иерархия объектов (глубина дерева наследования) не слишком сложная, так что можно поддерживать хороший уровень модульности. Впоследствии это облегчит переход к другому архитектурному стилю.

При использовании многоуровневой архитектуры нужно также остерегаться антипаттерна *архитектурная воронка (architecture sinkhole)*. Он возникает, когда запросы переходят от уровня к уровню с простой сквозной обработкой без выполнения бизнес-логики на каждом уровне. Предположим, к примеру, что уровень представления отвечает за простой пользовательский запрос на получение основных данных о клиенте (например, имени и адреса). Уровень представления передает запрос на уровень бизнес-логики, который ничего не делает, кроме передачи запроса дальше на уровень правил, который, в свою очередь, ничего не делает, кроме передачи запроса на уровень сохранения данных, который затем делает простой SQL-вызов к уровню базы данных для извлечения данных клиента. Затем данные проходят весь обратный путь по стеку уровней, не подвергаясь никакой дополнительной обработке или логике объединения, вычисления, применения правил или преобразования. В результате получается ненужное создание экземпляра объекта и его обработка, влияющие как на потребление памяти, так и на производительность.

У каждой многоуровневой архитектуры будет по крайней мере несколько сценариев, подпадающих под этот антипаттерн. Определить, есть ли архитектурная воронка, поможет анализ процентной доли запросов, соответствующих условиям антипаттерна. Обычно приемлемой практикой считается правило 80:20. Например, вполне допустимо, если антипаттерну соответствует всего лишь 20 %

запросов. А вот если в воронку попадает 80 % запросов, то многоуровневая архитектура будет неудачным выбором для решения задачи предметной области. Можно избежать возникновения архитектурной воронки, если использовать только открытые уровни архитектуры, учитывая, конечно, что при этом повысится сложность управления изменениями.

Зачем выбирать этот архитектурный стиль

Многоуровневый архитектурный стиль считается вполне подходящим для небольших простых приложений или веб-сайтов. Он также является удачным архитектурным выбором, особенно в качестве отправной точки, в условиях весьма жесткого бюджета и лимитированного времени. Из-за своей простоты и освоенности разработчиками и архитекторами многоуровневая архитектура является, наверное, одним из наименее затратных архитектурных стилей, упрощающих разработку небольших приложений. Многоуровневый архитектурный стиль считается также неплохим выбором, когда архитектор все еще находится в процессе анализа потребностей бизнеса и не знает, какой из стилей окажется эффективнее других.

По мере увеличения количества приложений с многоуровневым архитектурным стилем прослеживается его негативное влияние на такие свойства, как сопровождаемость, гибкость, тестируемость и развертываемость. Поэтому крупным приложениям и системам, использующим многоуровневую архитектуру, лучше подойдут другие, более модульные архитектурные стили.

Оценки архитектурных свойств

Одна звезда в оценочной таблице свойств (показанной на рис. 10.6) означает, что данное архитектурное свойство плохо поддерживается в архитектуре, а пять звезд — что свойство является одной из самых сильных сторон архитектурного стиля. Определение каждого свойства, указанного в таблице, можно найти в главе 4.

Общие затраты и простота являются основными сильными сторонами многоуровневого архитектурного стиля. Будучи по своей природе монолитными, многоуровневые архитектуры не обладают сложностью, характерной для распределенных стилей, и являются простыми и понятными, а кроме того, их построение и сопровождение также обходятся относительно дешево. И тем не менее следует предупредить, что эти высокие оценки начинают быстро снижаться, стоит только монолитным многоуровневым архитектурам разрастись и, следовательно, стать сложнее.

Архитектурное свойство	Рейтинг в звездах
Тип разбиения	Технический
Количество квантов	1
Развертываемость	★
Адаптируемость	★
Эволюционность	★
Отказоустойчивость	★
Модульность	★
Общие затраты	★★★★★
Производительность	★★
Надежность	★★★
Масштабируемость	★
Простота	★★★★★
Тестируемость	★★

Рис. 10.6. Оценки свойств многоуровневой архитектуры

У этого архитектурного стиля весьма низкие оценки развертываемости и тестируемости. Низкая оценка развертываемости объясняется церемонией развертывания (усилиями, прилагаемыми к развертыванию), высоким риском и отсутствием частых развертываний. Простое трехстрочное изменение файла класса в приложении с многоуровневым архитектурным стилем требует повторного развертывания всего модуля и, возможно, повлечет за собой изменения базы данных, конфигурации или другие неочевидные исправления программного кода. Помимо этого, такому простому трехстрочному изменению обычно сопутствуют десятки других корректировок, что еще больше увеличивает вероятность развертывания (а также возрастает частота развертываний). Низкая оценка тестируемости также является результатом этого сценария: при простом трехстрочном изменении большинство разработчиков вряд ли станут тратить время на многочасовое тестирование, добиваясь прохождения всего набора регрессионных тестов (даже если бы этому изначально придавалось первосте-

пенное значение), особенно наряду с десятками других изменений, внесенных в монолитное приложение в то же время. Мы оценили тестируемость двумя звездами (а не одной) из-за возможности имитировать компоненты (или даже целый уровень) или использовать вместо них заглушки, что в целом упрощает тестирование.

Общая надежность этого архитектурного стиля имеет среднюю оценку (три звезды), главным образом из-за отсутствия сетевого трафика и таких понятий, как полоса пропускания и время задержки, имеющихся у большинства распределенных архитектур. Многоуровневая архитектура получила от нас за надежность только три звезды из-за присущего ей монолитного развертывания в сочетании с низкой оценкой тестируемости (полноты тестирования) и рисками повторного развертывания.

Адаптируемость и масштабируемость многоуровневой архитектуры оценены очень низко (одной звездой) в основном из-за монолитного развертывания и отсутствия архитектурной модульности. Хотя в масштабе монолита некоторые функции по сравнению с остальными могут получать расширенную реализацию, эта работа обычно требует очень сложных методов проектирования, включая многопоточность, внутренний обмен сообщениями и другие методы параллельной обработки, то есть те методы, для которых данная архитектура плохо приспособлена. Но поскольку многоуровневая архитектура благодаря монолитному пользовательскому интерфейсу, внутренней обработке и монолитной базе данных всегда представлена единым системным квантом, приложения, основанные на одном кванте, могут масштабироваться только до определенного предела.

Для многоуровневой архитектуры неизменный интерес представляет оценка производительности. Мы дали ей всего две звезды, поскольку этот архитектурный стиль просто не подходит для высокопроизводительных систем из-за отсутствия параллельной обработки, закрытости уровней и наличия антипаттерна архитектурной воронки. По аналогии с масштабируемостью производительность может быть повышена за счет кэширования, многопоточности и похожих приемов, но эти свойства данному архитектурному стилю не присущи: для их получения архитекторам и разработчикам придется серьезно потрудиться.

Из-за монолитных развертываний и отсутствия архитектурной модульности многоуровневая архитектура не поддерживает отказоустойчивость. Если одна небольшая часть многоуровневой архитектуры вызывает нехватку памяти, это влияет на весь модуль приложения и приводит к его сбою. Кроме того, на общую доступность оказывает влияние высокое среднее время восстановления (mean-time-to-recovery, MTTR), обычно наблюдаемое в большинстве монолитных приложений, при этом время запуска варьируется от двух минут для небольших приложений до 15 и более минут для большинства крупных приложений.

ГЛАВА 11

Конвейерная архитектура

Одним из фундаментальных стилей архитектуры ПО, не теряющим своей актуальности, является конвейерная архитектура, известная также как архитектура *каналов* и *фильтров* (*pipes and filters*). Этот стиль появился, когда разработчики и архитекторы решили разбить функциональность на отдельные части. Большинству разработчиков эта архитектура знакома в качестве базового принципа, лежащего в основе языков командной оболочки Unix, таких как Bash[1] и Zsh[2].

Разработчики, предпочитающие языки функционального программирования, без труда заметят схожие черты между конструкциями языка и элементами этой архитектуры. Фактически такой топологии следуют многие инструменты, использующие модель распределенных вычислений MapReduce[3]. Хотя указанные примеры являются низкоуровневой реализацией конвейерного архитектурного стиля, сам стиль может также использоваться и для бизнес-приложений высокого уровня.

Топология

Топология конвейерной архитектуры, показанная на рис. 11.1, состоит из каналов и фильтров.

Каналы и фильтры взаимодействуют определенным образом, при этом каналы формируют одностороннюю связь между фильтрами, обычно работающую по принципу «точка — точка» (point-to-point).

---

[1] https://ru.wikipedia.org/wiki/Bash

[2] https://ru.wikipedia.org/wiki/Zsh

[3] https://ru.wikipedia.org/wiki/MapReduce

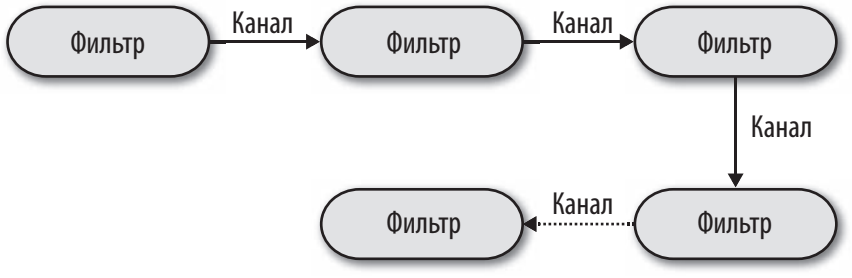


Рис. 11.1. Базовая топология конвейерной архитектуры

Каналы

Каналы представляют собой архитектурную форму линии связи между фильтрами. Обычно из соображений производительности каждый канал является однонаправленной и двухточечной (а не широковещательной) линией связи, принимающей входные данные от одного источника и неизменно перенаправляющей выходные данные приемнику. Полезная нагрузка, передаваемая по каналам, может быть в каком угодно формате, но для достижения высокой производительности архитекторы отдают предпочтение меньшим объемам данных.

Фильтры

Фильтры характеризуются автономностью, независимостью от других фильтров и, как правило, отсутствием состояния. Каждый фильтр должен выполнять только одну задачу. Составные задачи должны решаться последовательностью фильтров, а не одним.

В данном архитектурном стиле используются четыре типа фильтров:

Производитель (producer)

Начальная точка процесса, являющаяся исключительно исходной точкой, иногда называемой *источником*.

Преобразователь (transformer)

Принимает входные данные, может выполнять преобразование некоторых или всех данных, после чего пересылает их в исходящий канал. Сторонники функциональной парадигмы усмотрят в этой функции *отображение*.

Тестер (tester)

Принимает входные данные, тестирует по одному или нескольким критериям, после чего может, если нужно, выдать на основе теста выходные данные. Программисты, работающие на функциональных языках, могут заметить, что это похоже на *свертку*.

Потребитель (consumer)

Конечная точка конвейера. Иногда потребители сохраняют конечный результат или данные, полученные при прогоне по конвейеру, в базе данных либо показывают конечный результат на экране пользовательского интерфейса.

Однонаправленность и простота каналов и фильтров способствуют повторному использованию композиции. Многим разработчикам эта возможность открылась благодаря работе с оболочками. Эффективность таких абстракций наглядно демонстрируется известной историей из блога «More Shell, Less Egg»[1] («Больше скорлупы, меньше яиц»). Однажды Дональда Кнута (Donald Knuth) попросили написать программу для решения следующей задачи, касающейся текста: нужно считать текстовый файл и найти в нем *n* самых частых слов, а затем вывести на печать отсортированный список этих слов с указанием частоты их появления. Он написал программу, состоящую более чем из десяти страниц на Паскале, попутно придумав (и задокументировав) новый алгоритм. А затем Даг Макилрой (Doug McIlroy) показал скрипт оболочки, запросто уместившийся в твиттерный пост и позволявший решить эту же задачу намного проще, элегантнее и понятнее (если, конечно, читатель разбирается в командах оболочки):

```
tr -cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed ${1}q
```

Такому творческому применению простых, но мощных составных абстракций порой удивляются даже сами разработчики оболочек операционной системы Unix.

[1] http://www.leancrew.com/all-this/2011/12/more-shell-less-egg/. Игра слов: shell — одновременно «оболочка» и «скорлупа». — *Примеч. ред.*

Пример

Схема конвейерной архитектуры применяется в самых разных приложениях, особенно при решении задач с простой односторонней обработкой. Например, этот паттерн используется многими инструментами электронного обмена данными (Electronic Data Interchange, EDI), позволяя с помощью каналов и фильтров создавать преобразования документа из одного типа в другой. Инструменты ETL (extract, transform, load — извлечение, преобразование, загрузка) также используют конвейерную архитектуру для потока и изменения данных при их перемещении из одной базы или другого источника данных в другую. Оркестраторы и медиаторы процессов, такие как Apache Camel[1], используют конвейерную архитектуру для передачи информации от одного этапа бизнес-процесса к другому.

В качестве иллюстрации возможного использования конвейерной архитектуры рассмотрим следующий пример, показанный на рис. 11.2, где различная служебная телеметрическая информация отправляется из сервисов посредством потоковой передачи в Apache Kafka[2].

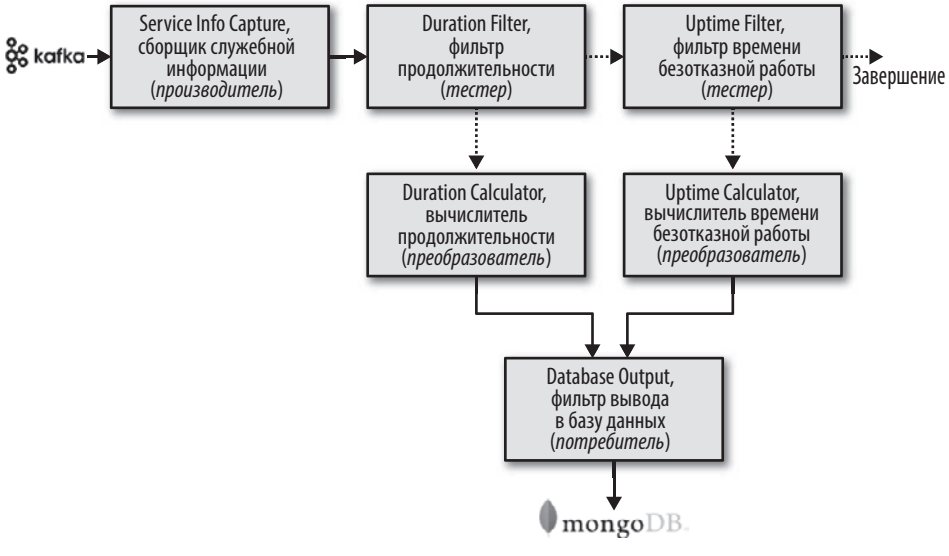


Рис. 11.2. Пример конвейерной архитектуры

[1] https://camel.apache.org/

[2] https://kafka.apache.org/

Изучая рис. 11.2, обратите внимание на использование архитектуры конвейерного стиля для обработки различных типов данных, передаваемых в Kafka. Фильтр — сборщик служебной информации, `Service Info Capture` (производитель) подписывается на топик Kafka и получает служебную информацию. Затем он отправляет собранные данные к фильтру-тестеру продолжительности, `Duration Filter`, чтобы определить, имеют ли данные, полученные из Kafka, отношение к продолжительности запроса на обслуживание (в миллисекундах). Обратите внимание на разделение задач между фильтрами: фильтр — сборщик служебной информации `Service Info Capture` занят только подключением к топику Kafka и получением потоковых данных, а фильтр продолжительности `Duration Filter` занимается лишь квалифицированием данных и, при необходимости, направлением их в следующий канал. Если данные имеют отношение к продолжительности запроса на обслуживание (в миллисекундах), фильтр продолжительности `Duration Filter` передает их фильтру преобразования: вычислителю продолжительности `Duration Calculator`, а если нет, то передает их фильтру — тестеру времени безотказной работы `Uptime Filter` для проверки, имеют ли данные отношение к показателям безотказной работы. В противном случае конвейер завершается, поскольку именно этому потоку обработки данные становятся неинтересны. Если же данные относятся к показателям времени безотказной работы, фильтр передает их фильтру — вычислителю времени безотказной работы `Uptime Calculator` для подсчета показателей безотказной работы сервиса. После вычислений фильтры-преобразователи передают измененные данные фильтру-потребителю, занимающемуся выводом в базу данных `Database Output`, который затем их сохраняет в базе данных MongoDB.

В этом примере показаны свойства расширяемости конвейерной архитектуры. Например, в структуру, показанную на рис. 11.2, после фильтра времени безотказной работы `Uptime Filter` легко можно добавить новый фильтр для обслуживания новой собранной метрики, например времени ожидания подключения к базе данных.

Оценки архитектурных свойств

Одна звезда в оценочной таблице свойств (показанной на рис. 11.3) означает, что данное архитектурное свойство плохо поддерживается в архитектуре, а пять звезд — что архитектурное свойство является одной из самых сильных сторон архитектурного стиля. Определение каждого свойства, указанного в оценочной таблице, можно найти в главе 4.

Архитектура конвейерного стиля характеризуется разбиением по техническому принципу, поскольку логика приложения разбивается по типам фильтров (про-

изводитель, тестер, преобразователь и потребитель). К тому же конвейерная архитектура обычно реализуется в виде монолитного модуля развертывания, поэтому в ней всегда только один архитектурный квант.

Архитектурное свойство	Оценка в звездах
Тип разбиения	Технический
Количество квантов	1
Развертываемость	★★
Адаптируемость	★
Эволюционность	★★★
Отказоустойчивость	★
Модульность	★★★
Общие затраты	★★★★★
Производительность	★★
Надежность	★★★
Масштабируемость	★
Простота	★★★★★
Тестируемость	★★★

Рис. 11.3. Оценки свойств конвейерной архитектуры

Сильными сторонами архитектуры конвейерного стиля являются общая стоимость и простота в сочетании с модульностью. В монолитной конвейерной архитектуре нет сложностей, характерных для распределенных архитектурных стилей, она проста и легка в понимании и не требует больших затрат в создании и сопровождении. Архитектурная модульность достигается путем разделения задач между фильтрами и преобразователями различных типов. Любые из этих фильтров могут быть изменены или заменены без какого-либо влияния на другие фильтры. Например, в примере Kafka, показанном на рис. 11.2, фильтр преобразования `Duration Calculator` может быть модифицирован для изменения вычисления продолжительности, и это никак не повлияет на любой другой фильтр.

Развертываемость и тестируемость хотя и находятся на среднем уровне, все же немного выше, чем у многоуровневой архитектуры, благодаря уровню модульности, достигаемому за счет особенностей фильтров. И тем не менее этот архитектурный стиль является монолитным, поэтому на конвейерную архитектуру также оказывают влияние все присущие монолитам особенности, риски, частота развертывания и сложности прохождения тестов.

В этом архитектурном стиле, как и в многоуровневой архитектуре, общая надежность оценивается на среднем уровне (тремя звездами), в основном из-за отсутствия сетевого трафика и таких понятий, как полоса пропускания и время задержки, имеющихся у большинства распределенных архитектур. Эта архитектура получила от нас за надежность только три звезды из-за монолитного развертывания и проблем тестируемости и развертываемости (необходимости тестирования всего монолита и полного развертывания этого монолита при любых вносимых изменениях).

Адаптируемость и масштабируемость конвейерной архитектуры оценены очень низко (одной звездой) опять же из-за монолитного развертывания. Хотя в масштабе монолита некоторые функции по сравнению с остальными могут получать расширенную реализацию, эта работа обычно требует очень сложных методов проектирования, включая многопоточность, внутренний обмен сообщениями и другие методы параллельной обработки, то есть тех методов, для которых данная архитектура плохо приспособлена. Но поскольку конвейерная архитектура благодаря монолитному пользовательскому интерфейсу, внутренней обработке и монолитной базе данных всегда представлена единым системным квантом, приложения, основанные на одном кванте, могут масштабироваться только до определенного предела.

Из-за монолитных развертываний и отсутствия архитектурной модульности конвейерная архитектура не поддерживает отказоустойчивость. Если одна небольшая часть конвейерной архитектуры испытывает нехватку памяти, это влияет на весь модуль приложения и приводит к сбою. Кроме того, на общую доступность оказывает влияние высокое среднее время восстановления (mean-time-to-recovery, MTTR), обычно наблюдаемое в большинстве монолитных приложений, при этом время запуска варьируется от двух минут для небольших приложений до 15 и более минут для большинства крупных приложений.

ГЛАВА 12

Микроядерная архитектура

Микроядерный (microkernel) архитектурный стиль (также известный как архитектура *плагинов*) был придуман несколько десятилетий назад и широко используется в настоящее время. Этот архитектурный стиль отлично подходит для приложений на основе готовых продуктов (упакованных и доступных для загрузки и установки в виде единого монолитного модуля развертывания, обычно устанавливаемого на сайте заказчика как сторонний продукт), но широко используется и во многих коммерческих пользовательских приложениях.

Топология

Микроядерный архитектурный стиль относится к сравнительно простой монолитной архитектуре, состоящей из двух архитектурных компонентов: основной системы, или ядра, и подключаемых компонентов (плагинов). Логика приложения делится между независимыми подключаемыми компонентами и основной системой, что обеспечивает расширяемость, адаптируемость и изолированность функций приложения, а также настраиваемую логику обработки. Основная топология микроядерного архитектурного стиля показана на рис. 12.1.

Ядро системы

Ядро системы формально определяется как минимальная функциональность, необходимая для работы системы. Хорошим примером может послужить IDE-среда Eclipse. Ядро Eclipse представлено простым текстовым редактором, позволяющим открыть файл, изменить текст и сохранить файл. И только когда будут добавлены плагины, Eclipse начинает превращаться в полезный продукт. А еще ядро — это «счастливый путь» (общий поток обработки) через приложение,

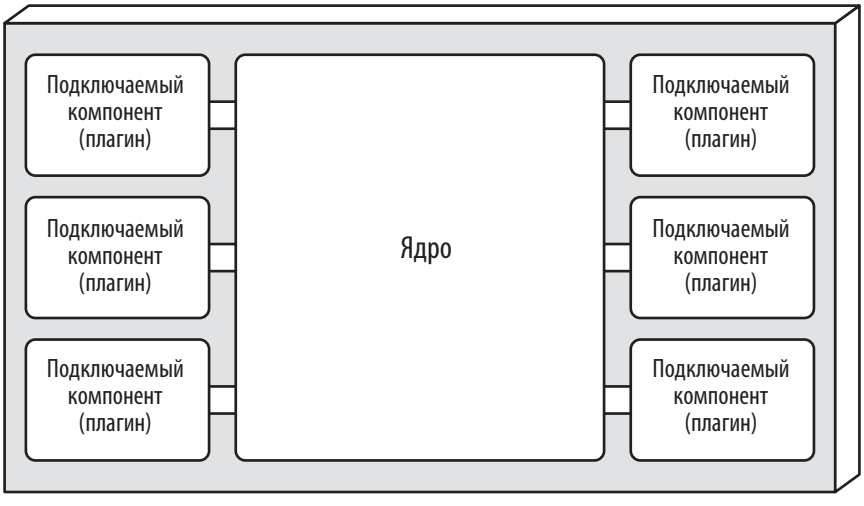


Рис. 12.1. Основные компоненты микроядерного архитектурного стиля

с небольшой или незначительной обработкой. Устранение цикломатической сложности из ядра и размещение ее в отдельных подключаемых компонентах позволяет улучшить расширяемость и сопровождаемость, а также повысить тестируемость. Предположим, к примеру, что приложение для утилизации подержанных электронных устройств должно выполнять конкретные правила оценки для каждого полученного устройства. Код на Java для такого рода обработки может выглядеть следующим образом:

```
public void assessDevice(String deviceID) {
   if (deviceID.equals("iPhone6s")) {
      assessiPhone6s();
   } else if (deviceID.equals("iPad1"))
      assessiPad1();
   } else if (deviceID.equals("Galaxy5"))
      assessGalaxy5();
   } else ...
      ...
   }
}
```

Вместо того чтобы размещать всю кастомизированную клиентскую настройку в ядре, существенно увеличивая цикломатическую сложность, стоит создать отдельный подключаемый компонент для каждого оцениваемого электронного устройства. Специальные клиентские подключаемые компоненты смогут не только изолировать независимую логику устройства от остальной части потока обработки, но и обеспечить возможность расширения. Добавление

нового оцениваемого устройства сведется к простому подключению нового компонента и к обновлению реестра. Как показано в следующем измененном исходном коде, в микроядерном архитектурном стиле задача ядра при оценке электронного устройства сводится к нахождению и вызову соответствующего плагина устройства:

```
public void assessDevice(String deviceID) {
        String plugin = pluginRegistry.get(deviceID);
        Class<?> theClass = Class.forName(plugin);
        Constructor<?> constructor = theClass.getConstructor();
        DevicePlugin devicePlugin =
                (DevicePlugin)constructor.newInstance();
        DevicePlugin.assess();
}
```

В этом примере все сложные правила и инструкции для оценки конкретного устройства содержатся в автономном, независимом подключаемом компоненте, код которого в общем случае может быть выполнен из ядра.

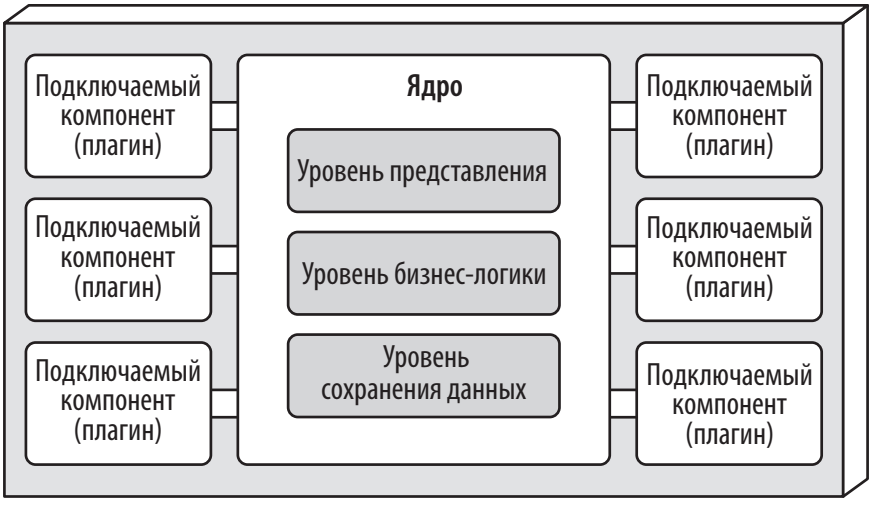
В зависимости от размера и сложности ядро может быть реализовано в виде многоуровневой архитектуры или в виде модульного монолита (рис. 12.2). В ряде случаев ядро может быть разбито на отдельно развертываемые сервисы предметных областей, где каждый сервис содержит конкретный подключаемый компонент, специфичный для определенной предметной области. Предположим, к примеру, что обработка платежа `Payment Processing` является сервисом ядра. Все методы платежа (кредитная карта, PayPal, кредит магазина, подарочная карта или заказ на покупку) будут представлены отдельными подключаемыми компонентами, подходящими исключительно для конкретного вида платежа. Во всех этих случаях одна база данных обычно распределяется на все монолитное приложение.

Уровень представления ядра может быть встроен в само ядро или же реализован в виде отдельного пользовательского интерфейса, при этом ядро предоставляет бэкенд-сервисы. Собственно говоря, отдельный пользовательский интерфейс может быть также реализован в микроядерном архитектурном стиле. Все эти варианты уровней представления в привязке к ядру показаны на рис. 12.3.

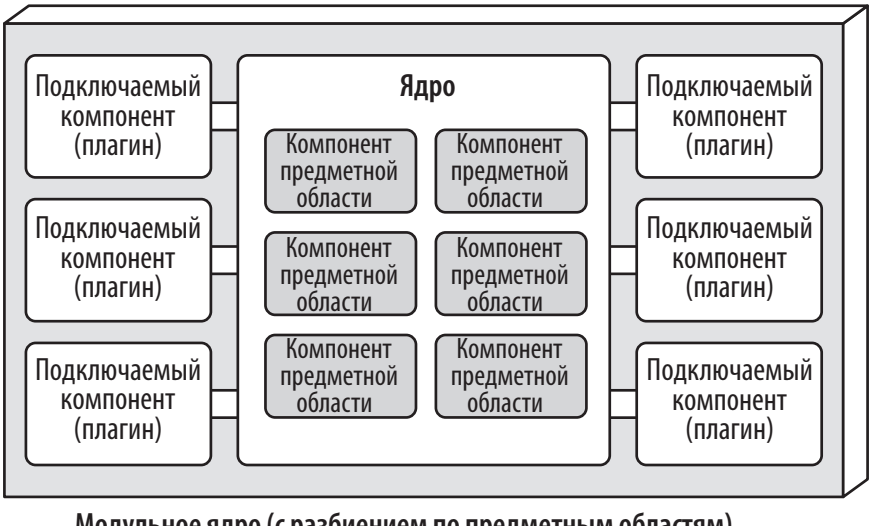
Подключаемые компоненты (плагины)

Подключаемые компоненты обладают автономностью и независимостью и содержат функции специализированной обработки, дополнительные средства и настраиваемый код, предназначенный для улучшения и расширения ядра. Кроме того, они могут использоваться для изоляции нестабильного кода, а также

для повышения сопровождаемости и тестируемости приложения. В идеале подключаемые компоненты должны быть абсолютно автономными и между ними не должно быть никаких зависимостей.



Многоуровневое ядро (с техническим разбиением)



Модульное ядро (с разбиением по предметным областям)

Рис. 12.2. Разновидности ядер с микроядерной архитектурой

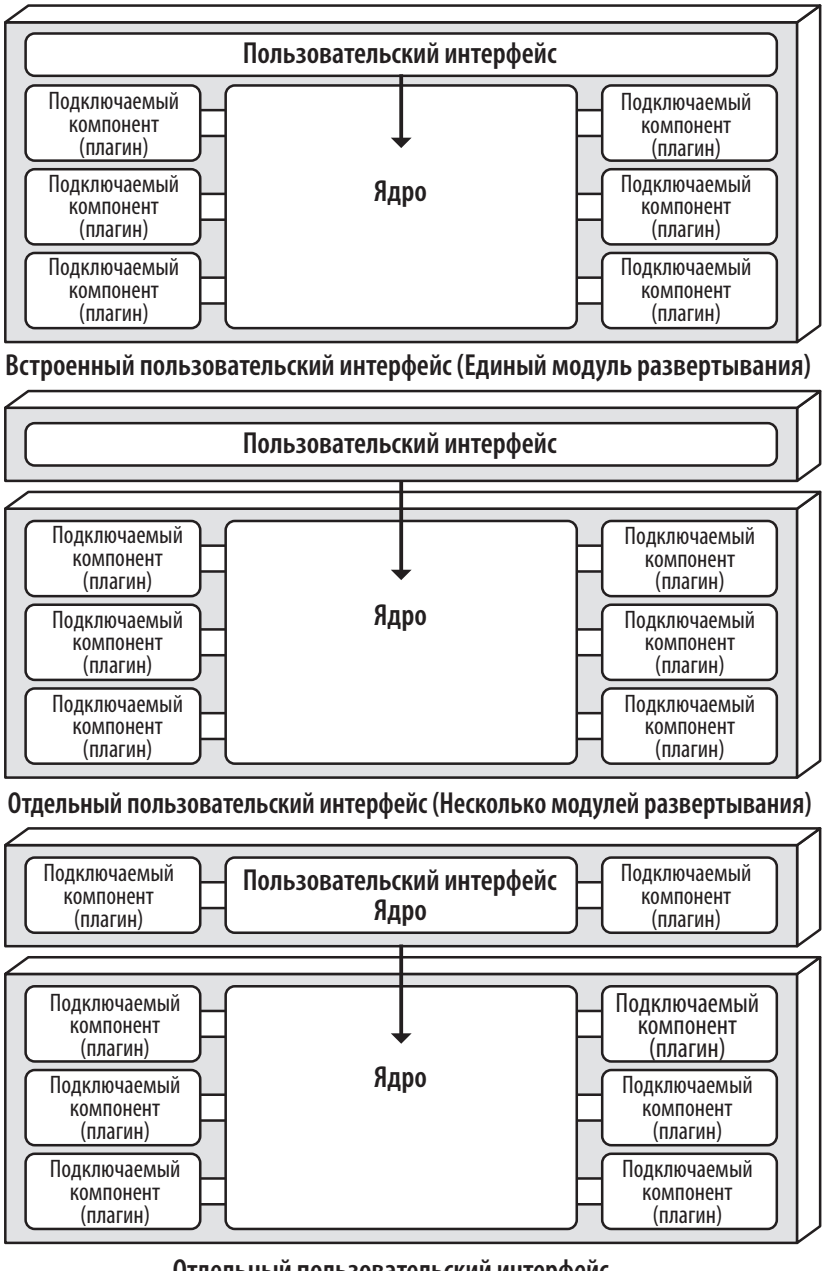


Рис. 12.3. Варианты пользовательского интерфейса

Обмен данными между подключаемыми компонентами и ядром обычно организуется по схеме «точка — точка», то есть «канал», подключающий плагин к ядру, обычно представлен вызовом метода или функции класса точки входа подключаемого компонента. Кроме того, подключаемый компонент может создаваться либо на этапе компиляции, либо в ходе выполнения приложения. Подключаемые компоненты, создаваемые в ходе выполнения приложения, могут добавляться или удаляться на ходу, не требуя повторного развертывания ядра или других плагинов, и управление ими обычно осуществляется через такие среды, как Open Service Gateway Initiative (OSGi)[1] для Java, Penrose (Java)[2], Jigsaw (Java)[3] или Prism (.NET). Подключаемые компоненты, создаваемые при компиляции, гораздо проще поддаются управлению, но требуют повторного развертывания всего монолитного приложения при модификации, добавлении или удалении.

Подключаемые по схеме «точка — точка» компоненты могут реализовываться в виде общих библиотек (например, JAR, DLL или Gem), имен пакетов в Java или же пространств имен в C#. Если снова обратиться к примеру оценки в приложении для утилизации электронных устройств, плагин для каждого электронного устройства может быть написан и реализован в виде JAR, DLL или Ruby Gem (или общей библиотеки любого другого типа), и как показано на рис. 12.4, с именем независимой общей библиотеки, совпадающим с названием устройства.

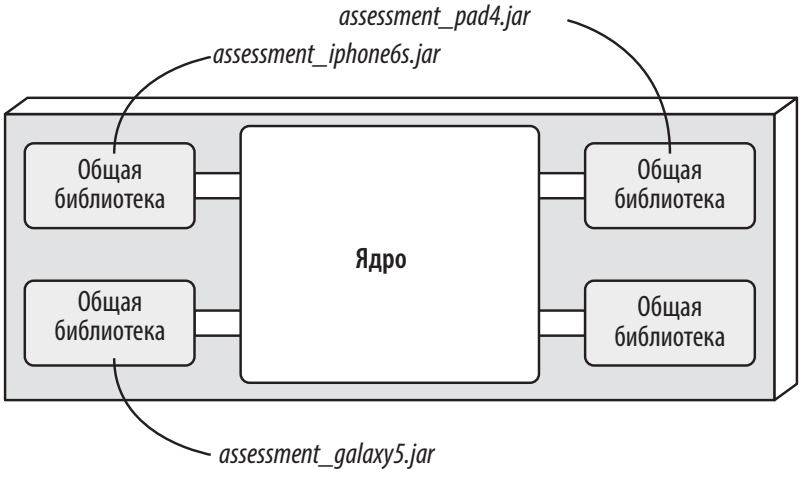


Рис. 12.4. Совместно используемая библиотека, реализованная в виде плагина

[1] https://www.osgi.org/

[2] https://openjdk.org/projects/penrose/

[3] https://openjdk.org/projects/jigsaw/

В качестве альтернативы при более простом подходе, показанном на рис. 12.5, каждый подключаемый компонент реализован в виде отдельного пространства имен или пакета с собственным именем в пределах одной и той же кодовой базы или IDE-проекта. При создании пространства имен рекомендуется придерживаться следующей семантики: `app.plugin.<domain>.<context>`. Рассмотрим, к примеру, пространство имен `app.plugin.assessment.iphone6s`. Второй узел (`plugin`) поясняет, что компонент является подключаемым, и поэтому нужно строго придерживаться основных правил, касающихся подключаемых компонентов (они должны быть автономны и отделены от других плагинов). Третий узел является описанием предметной области (в данном случае оценки — `assessment`), тем самым позволяя организовывать и группировать подключаемые компоненты по общему назначению. Четвертый узел (`iphone6s`) является описанием конкретного контекста плагина, упрощая поиск плагина конкретного устройства для внесения изменений или тестирования.

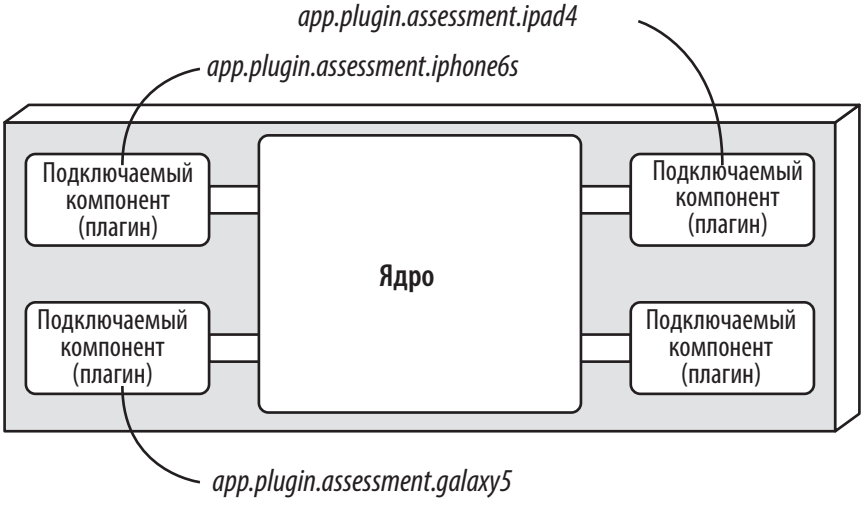


Рис. 12.5. Пакет или пространство имен, реализованное в виде плагина

Обмен данными подключаемого компонента с ядром не должен вестись исключительно по схеме «точка — точка». Есть и другие варианты, такие как использование в качестве вызова функций плагина REST-технологии или системы обмена сообщениями при условии, что каждый плагин работает как автономный сервис (или даже микросервис, реализованный с помощью контейнера). Хотя это может показаться неплохим способом повысить общую масштабируемость, не следует забывать, что такая топология (показанная на рис. 12.6) из-за моно-

литного ядра по-прежнему представлена одним-единственным архитектурным квантом. Каждый запрос, прежде чем добраться до подключаемого сервиса, должен сначала пройти через ядро.

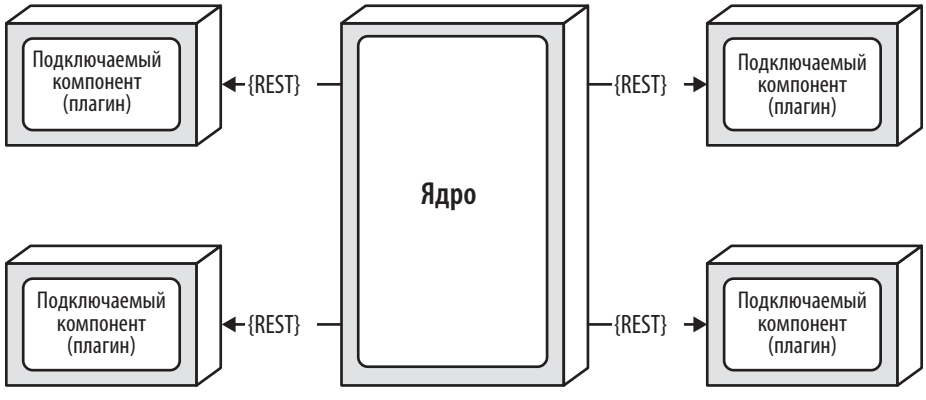


Рис. 12.6. Удаленное обращение к плагинам с использованием REST-технологии

Преимущества, получаемые за счет удаленного доступа к подключаемым компонентам, реализованным в виде отдельных сервисов, заключаются в улучшении общего развязывания (decoupling) компонентов, улучшении масштабируемости и пропускной способности, а также возможности вносить изменения в ходе выполнения приложения без каких-либо специализированных сред вроде OSGi, Jigsaw или Prism. Кроме того, появляется возможность асинхронного обмена данными с плагинами, что, в зависимости от сценария, способно существенно повысить для пользователя общую скорость отклика. В примере с утилизацией электроники вместо вынужденного ожидания запуска оценки электронного устройства ядро могло бы отправить асинхронный *запрос* для запуска оценки конкретного устройства. Как только она завершится, плагин может уведомить ядро с помощью еще одного канала асинхронного обмена сообщениями, а система, в свою очередь, даст понять пользователю, что оценка завершена.

Но эти преимущества не обходятся без компромиссов. Удаленный доступ к плагинам превращает микроядерную архитектуру из монолитной в распределенную, затрудняя ее реализацию и развертывание для большинства сторонних, локально устанавливаемых продуктов. Более того, возрастает общая сложность, растут затраты и усложняется общая топология развертывания. Если плагин перестает отвечать на запросы или не работает, особенно при использовании REST-технологии, то запрос не может быть выполнен.

А с монолитным развертыванием такого не происходит. На чем остановить свой выбор — на обращении к подключаемым компонентам со стороны ядра по схеме «точка — точка» или же на варианте с удаленным доступом — нужно решать на основе конкретных требований, а следовательно, по результатам тщательного анализа компромиссов и всех достоинств и недостатков того или иного подхода.

Непосредственное обращение подключаемых компонентов к общей базе данных обычно не практикуется. Чаще всего ответственность за такое обращение берет на себя ядро, которое передает в каждый плагин все необходимые данные. Основной причиной такого поведения является развязывание (decoupling). Изменения, вносимые в базу данных, должны оказывать влияние только на ядро, но никак не на подключаемые компоненты. И все же у плагинов могут быть свои собственные, доступные только им хранилища данных. Например, у каждого плагина, выполняющего оценку электронного устройства в примере системы утилизации подержанной техники, может быть своя собственная простая база данных или обработчик правил, содержащий все конкретные оценочные правила для каждого товара. Хранилище данных, находящееся во владении того или иного подключаемого компонента, может быть как внешним (рис. 12.7), так и встроенным в виде части подключаемого компонента или модуля монолитного развертывания (как в случае использования базы данных, созданной в оперативной памяти или при использовании ее встроенного варианта).

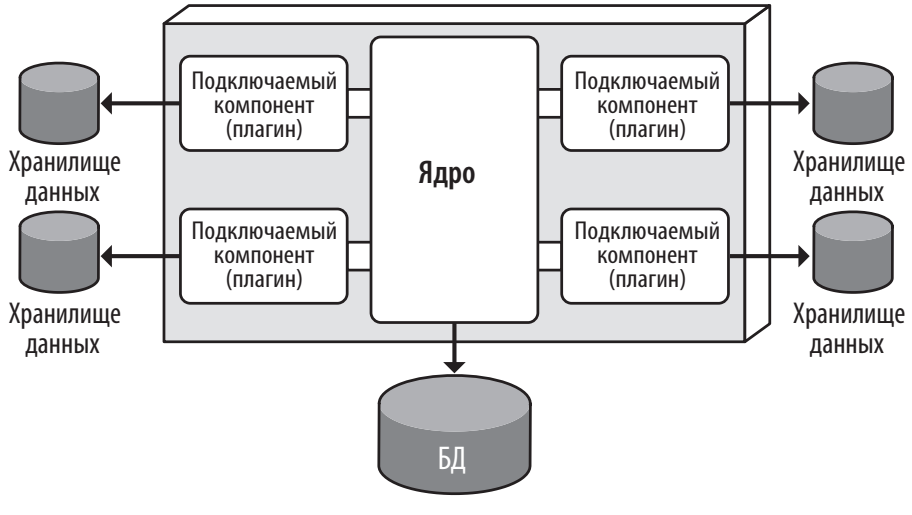


Рис. 12.7. У подключаемых компонентов могут быть свои собственные хранилища данных

Реестр

Ядру нужно знать, какие плагины доступны и как их можно получить. Одним широко распространенным вариантом решения этого вопроса является реестр плагинов. В этом реестре содержится информация о каждом плагине, включая его имя, соглашение о передаче данных и подробности протокола удаленного доступа (в зависимости от способа его подключения к ядру). Например, плагин для налогового программного обеспечения, помечающий элементы налогового аудита с высоким риском, может иметь запись реестра, содержащую имя сервиса (AuditChecker), соглашение о передаче данных (входных и выходных) и формат контракта (XML).

Реестр может быть очень простой внутренней структурой отображения, принадлежащей ядру и содержащей ключ и ссылку на подключаемый компонент, или же весьма сложным инструментом регистрации и обнаружения, либо встроенным в ядро, либо развернутым вне его (например, как Apache ZooKeeper[1] или Consul[2]). В примере с утилизацией электронных устройств следующий Java-код реализует простой реестр внутри ядра, показывая для оценки устройства типа iPhone 6S примеры входа по схеме «точка — точка», входа в обмен сообщениями и RESTful-входа:

```
Map<String, String> registry = new HashMap<String, String>();
static {
  //point-to-point access example
  registry.put("iPhone6s", "Iphone6sPlugin");

  //messaging example
  registry.put("iPhone6s", "iphone6s.queue");


  //restful example
  registry.put("iPhone6s", "https://atlas:443/assess/iphone6s");
}
```

Контракты

Контракты между подключаемыми компонентами и ядром бывают, как правило, стандартными для любых плагинов и содержат поведение, входные данные и выходные данные, возвращаемые из подключаемого компонента. Настраиваемые контракты обычно создаются в тех ситуациях, когда подключаемые компоненты

[1] https://zookeeper.apache.org/

[2] https://www.consul.io/

разрабатываются сторонними производителями, а также если с вашей стороны отсутствует контроль над контрактом, который использует плагин. В таких случаях зачастую создается конвертер между контрактом плагина и вашим стандартным контрактом, чтобы ядру не требовался отдельный код для каждого плагина.

Контракты плагинов могут быть реализованы в форматах XML, JSON или даже в виде объектов, передаваемых в обе стороны между плагином и ядром. В приложении для утилизации электронных устройств следующий контракт (реализованный в виде стандартного Java-интерфейса по имени `AssessmentPlugin`) определяет общее поведение (`assess()`, `register()` и `deregister()`), а также соответствующие ожидаемые данные на выходе из подключаемого компонента (`AssessmentOutput`):

```
public interface AssessmentPlugin {
        public AssessmentOutput assess();
        public String register();
        public String deregister();
}

public class AssessmentOutput {
        public String assessmentReport;
        public Boolean resell;
        public Double value;
        public Double resellPrice;
}
```

В этом примере контракта ожидается, что плагин оценки устройства возвратит отчет об оценке в виде отформатированной строки; флаг `resell` (в значении `true` или `false`) покажет, может ли это устройство быть продано на вторичном рынке или же безопасно утилизировано; и наконец, если оно может быть повторно продано (что является еще одной формой утилизации), какова рассчитанная стоимость товара и какой должна быть рекомендуемая цена такой продажи.

Обратите внимание на модель ролей и ответственности между ядром и подключаемыми компонентами в этом примере, особенно на поле `assessmentReport`. Ядро не несет ответственности за формат и раскрытие подробностей отчета об оценке, оно отвечает только за способ вывода информации для пользователя: распечатать или отобразить на экране.

Примеры и варианты использования

Микроядерная архитектура используется для реализации большинства инструментальных средств, применяемых для разработки и выпуска ПО. В качестве

отдельных примеров можно привести Eclipse IDE[1], PMD[2], Jira[3] и Jenkins[4]. Другими весьма распространенными продуктами, использующими микроядерную архитектуру, являются такие интернет-браузеры, как Chrome и Firefox: просмотрщики и другие плагины, которых нет в базовом браузере (ядре), добавляют дополнительные возможности. Можно привести несметное количество примеров продуктового программного обеспечения, но как в этом плане обстоят дела у крупных бизнес-приложений? Микроядерная архитектура применима и к ним. В качестве примера рассмотрим действия страховой компании, связанные с обработкой страховых исков.

Обработка таких исков представляет собой весьма сложный процесс. В каждой юрисдикции имеются свои правила и предписания относительно того, что в страховом иске разрешено, а что нет. Например, в некоторых юрисдикциях (в некоторых штатах) разрешается бесплатная замена лобового стекла, если оно повреждено камнем, а в других штатах такая практика отсутствует. Тем самым создается чуть ли не бесконечный набор условий для стандартной обработки исков.

Большинство приложений для обработки страховых претензий используют сложные системы правил, чтобы справиться с этой нелегкой задачей. Но эти механизмы могут превратиться в большой и сложный ком грязи, где изменение одного правила повлияет на другие, а для внесения простых правок придется вызывать целую армию аналитиков, разработчиков и тестировщиков, чтобы убедиться, что ничего не будет нарушено. Использование микроядерной архитектуры может решить многие из подобных проблем.

Правила обработки исков для каждой юрисдикции могут помещаться в отдельные автономные подключаемые компоненты (реализованные как исходный код или конкретный экземпляр механизма правил, к которому обращается подключаемый компонент). Тогда условия для конкретной юрисдикции могут быть добавлены, удалены или изменены без малейшего влияния на любую другую часть системы. Более того, можно будет добавлять и удалять новые юрисдикции, не затрагивая другие части системы. Ядром в этом примере будет стандартный процесс подачи и обработки иска, редко подвергаемый изменениям.

Еще одним примером большого и сложного бизнес-приложения, в котором может использоваться микроядерная архитектура, может послужить программа

---

[1] https://www.eclipse.org/ide

[2] https://pmd.github.io/

[3] https://pmd.github.io/

[4] https://www.atlassian.com/software/jira

начисления налоговых выплат. Например, в Соединенных Штатах имеется базовая двухстраничная налоговая форма, называемая формой 1040, содержащая сводку всей информации, необходимой для расчета персональных налоговых обязательств. В каждой строке налоговой формы 1040 имеется одно-единственное число, для вывода которого требуется множество других форм и учетных ведомостей (таких, как общий доход). Каждая из этих дополнительных форм и учетных ведомостей может быть реализована в виде подключаемого компонента, а итоговая налоговая форма 1040 будет ядром. В таком случае изменения в налоговом законодательстве могут быть изолированы в независимом подключаемом компоненте, упрощая внесение правок и уменьшая связанный с этим риск.

Оценки архитектурных свойств

Одна звезда в оценочной таблице свойств (показанной на рис. 12.8) означает, что данное архитектурное свойство плохо поддерживается в архитектуре, а пять звезд — что архитектурное свойство является одной из самых сильных сторон архитектурного стиля. Определение каждого свойства, указанного в оценочной таблице, можно найти в главе 4.

Сильными сторонами микроядерного архитектурного стиля, как и в многоуровневом архитектурном стиле, являются простота и общие затраты, а самыми слабыми сторонами — масштабируемость, отказоустойчивость и расширяемость. Эта слабость проявляется из-за характерной для архитектуры микроядра монолитности развертывания. Так же как и в многоуровневом архитектурном стиле, квант всегда только один, поскольку все запросы проходят через ядро, прежде чем добраться до независимых подключаемых компонентов. На этом сходство заканчивается.

Микроядерный архитектурный стиль уникален тем, что только он может иметь как предметное, так и техническое разбиение. Хотя для большинства микроядерных архитектур характерно техническое разбиение, аспект предметного разбиения присутствует в основном за счет сильного изоморфизма между предметной областью и архитектурой. Например, с этим архитектурным стилем очень хорошо сочетаются задачи, требующие различных конфигураций для каждой локализации или клиента. Еще одним примером являются продукт или приложение, в которых сильный упор делается на пользовательскую настройку и расширение функциональных возможностей (здесь можно вспомнить Jira или такую IDE-среду, как Eclipse).

Архитектурное свойство	Оценка в звездах
Тип разбиения	предметный и технический
Количество квантов	1
Развертываемость	☆☆☆
Адаптируемость	☆
Эволюционность	☆☆☆
Отказоустойчивость	☆
Модульность	☆☆☆
Общие затраты	☆☆☆☆☆
Производительность	☆☆☆
Надежность	☆☆☆
Масштабируемость	☆
Простота	☆☆☆☆
Тестируемость	☆☆☆

Рис. 12.8. Оценки свойств микроядерной архитектуры

Тестируемость, развертываемость и надежность оцениваются чуть выше среднего (тремя звездами), в основном из-за возможности изоляции функций в независимых подключаемых компонентах. При правильном подходе к делу это сократит общий объем тестирования изменений, а также снизит общий риск развертывания, особенно если подключаемые компоненты развертываются в ходе выполнения приложения.

Модульность и расширяемость также оцениваются чуть выше среднего (тремя звездами). При использовании микроядерного архитектурного стиля дополнительные функциональные возможности могут добавляться, удаляться и изменяться через независимые, автономные плагины, упрощая процесс расширения и совершенствования приложений, созданных с использованием этого архитектурного стиля, и позволяя командам разработчиков реагировать на изменения

значительно быстрее. Рассмотрим подготовку к налоговым выплатам, приведенную выше. Если налоговое законодательство США претерпит изменения (что случается с завидным постоянством) и потребует заполнения новой налоговой формы, эта новая форма может быть создана в виде подключаемого компонента и без особых усилий добавлена к приложению. Если же налоговая форма или учетная ведомость утратит актуальность, соответствующий ей плагин можно просто удалить из приложения.

Неизменный интерес в микроядерном архитектурном стиле вызывает оценка производительности. Это свойство оценено нами в три звезды (чуть выше среднего уровня), главным образом из-за того, что микроядерные приложения зачастую невелики по размеру и не разрастаются так же интенсивно, как большинство приложений с многоуровневой архитектурой. К тому же они меньше страдают от антипаттерна архитектурной воронки, рассмотренного в главе 10. И наконец, приложения, использующие микроядерную архитектуру, можно упростить, отключив ненужную функциональность, и ускорить этим их работу. Хорошим примером может послужить сервер приложений Wildfly[1] (ранее известный как JBoss Application Server). За счет отключения ненужной функциональности, например кластеризации, кэширования и обмена сообщениями, сервер приложений начинает работать значительно быстрее, чем при наличии этих функций.

ГЛАВА 13

Архитектура на основе сервисов

Архитектура *на основе сервисов (service-based)* представляет собой вариант архитектуры микросервисов и считается одной из наиболее прагматичных, главным образом из-за своей гибкости. Архитектура на основе сервисов является распределенной, но при этом у нее нет такого же уровня сложности и затратности, как, например, у микросервисов или архитектур, управляемых событиями, что делает ее оптимальным выбором для многих бизнес-приложений.

Топология

Типовая топология архитектуры на основе сервисов — это распределенная макроуровневая структура, состоящая из отдельно развернутого пользовательского интерфейса, отдельно развернутых удаленных крупномодульных сервисов и монолитной базы данных (рис. 13.1).

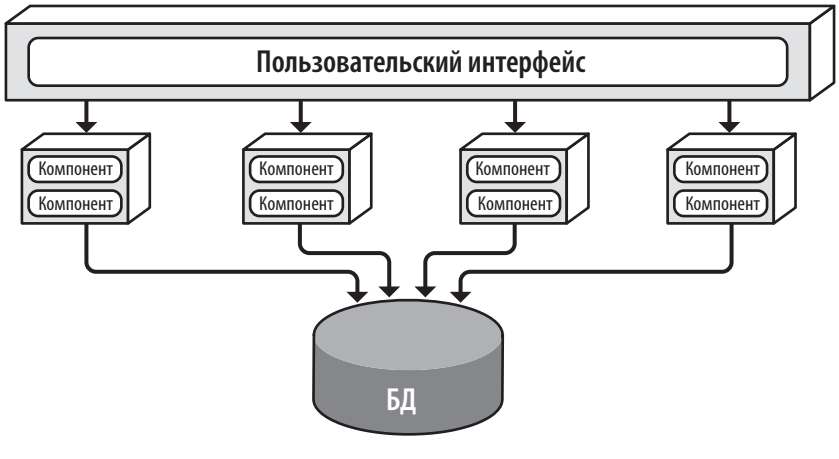


Рис. 13.1. Типовая топология архитектуры на основе сервисов

Сервисы в этом архитектурном стиле представлены, как правило, независимыми и отдельно развертываемыми крупномодульными «порциями приложения», которые обычно называют *сервисами предметной области (domain services)*. Сервисы развертываются точно так же, как и любое монолитное приложение (например, EAR-файл, WAR-файл или сборка), и поэтому не требуют контейнеризации (хотя сервис предметной области можно развернуть в таком контейнере, как Docker). Поскольку сервисы обычно совместно используют единую монолитную базу данных, их число в контексте приложения чаще всего варьируется от 4 до 12 единиц, то есть в среднем используется около 7 сервисов.

В большинстве случаев в архитектуре на основе сервисов имеется только один экземпляр каждого сервиса предметной области. Но с учетом масштабируемости, отказоустойчивости и необходимой пропускной способности может использоваться и несколько экземпляров. В таком случае обычно требуется какой-либо механизм, обеспечивающий распределение рабочей нагрузки между пользовательским интерфейсом и сервисом, чтобы интерфейс мог быть направлен на работоспособный и доступный экземпляр.

Обращение к сервисам осуществляется с помощью протокола удаленного доступа из пользовательского интерфейса. Обычно применяется REST-протокол, но может также использоваться обмен сообщениями, удаленный вызов процедур (remote procedure call, RPC) или даже SOAP-протокол. Также может встречаться и API-уровень, состоящий из прокси-сервера или шлюза (или иных внешних запросов), но в большинстве случаев пользовательский интерфейс обращается к сервисам напрямую с помощью паттерна Локатор сервисов[1], встроенного в сам интерфейс, API-шлюз или в прокси-сервер.

Одним из наиболее важных аспектов архитектуры на основе сервисов является то, что она обычно использует централизованную общую базу данных. Это позволяет сервисам применять SQL-запросы и соединения точно так же, как в мире традиционной монолитной многоуровневой архитектуры. Небольшое количество сервисов (от 4 до 12) обычно не создает в архитектуре на основе сервисов никаких сложностей с подключением к базе данных. Однако изменения в базе данных могут стать источником дополнительных проблем. Технологии, позволяющие с ними справиться, будут рассмотрены далее в разделе «Разбиение базы данных» на с. 210.

Варианты топологии

Множество вариантов топологии делают архитектурный стиль на основе сервисов, наверное, наиболее гибким архитектурным стилем. Например, показанный

[1] https://ru.wikipedia.org/wiki/Локатор_служб

на рис. 13.1 единый монолитный пользовательский интерфейс может быть разбит по предметным областям на несколько пользовательских интерфейсов, в том числе до уровня соответствия каждой предметной области. Варианты пользовательского интерфейса показаны на рис. 13.2.

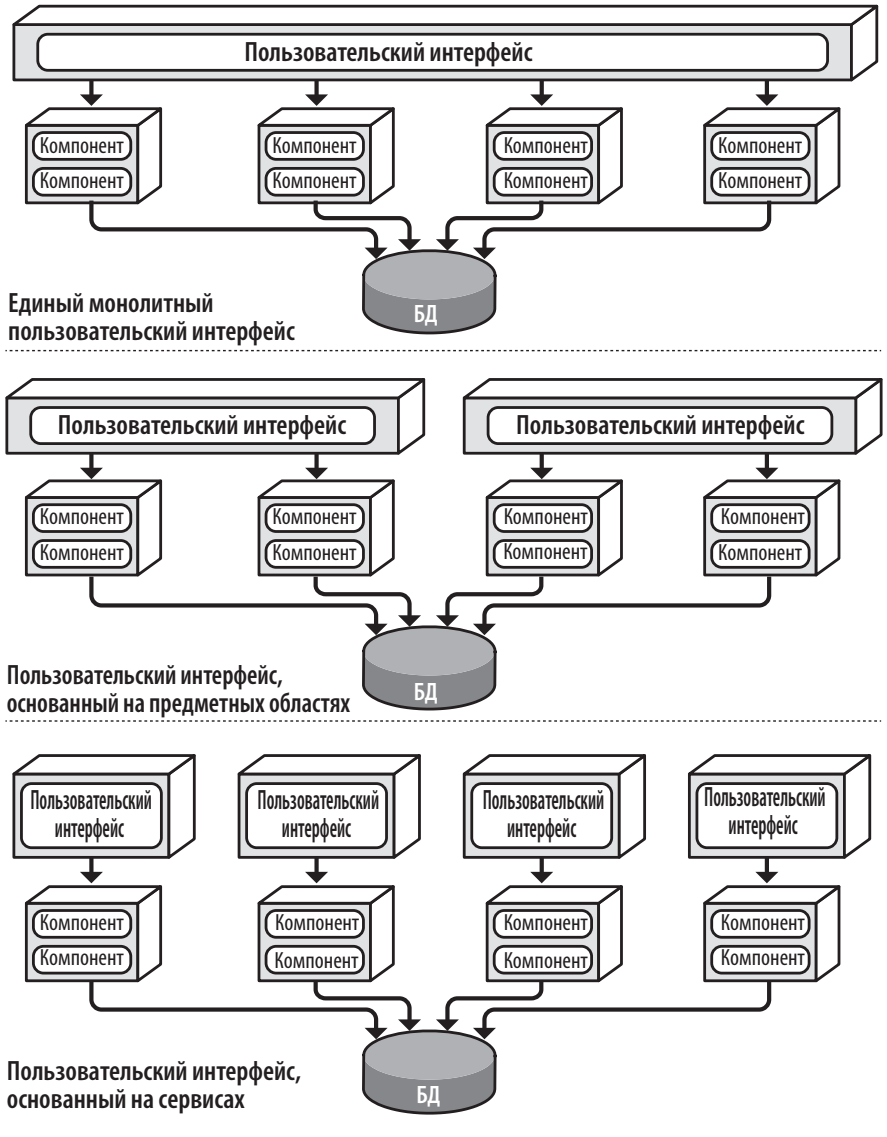


Рис. 13.2. Варианты пользовательского интерфейса

Точно так же не исключается возможность разбиения единой монолитной базы данных на отдельные базы, вплоть до баз данных, соответствующих каждому сервису предметной области (как в случае с микросервисами). При этом важно убедиться, что данные каждой отдельной базы данных не нужны никакому другому сервису. Это позволяет обойтись без межсервисного обмена данными (который при использовании архитектуры на основе сервисов совершенно недопустим), а также без дублирования информации в базах данных. Такие варианты показаны на рис. 13.3.

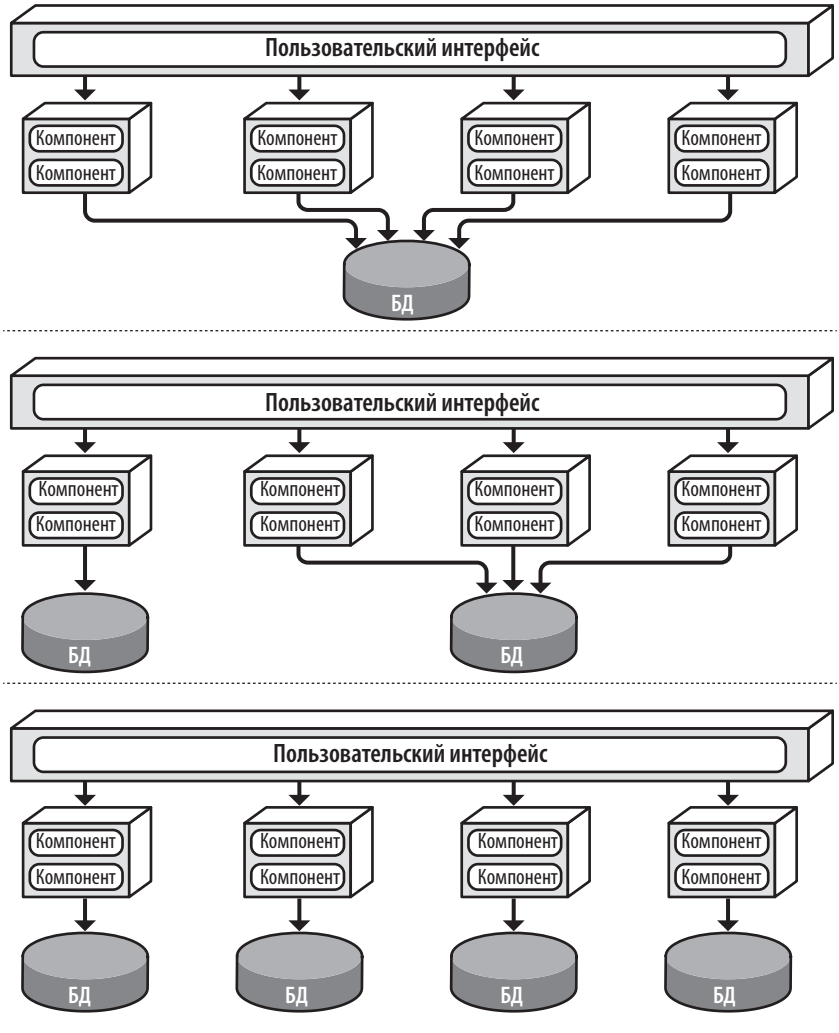


Рис. 13.3. Варианты баз данных

И наконец, можно также добавить API-уровень, состоящий из обратного прокси-сервера или шлюза между пользовательским интерфейсом и сервисами (рис. 13.4). Это хорошее решение при предоставлении внешним системам функциональности сервиса предметной области или при объединении общих сквозных проблем и переносе их за пределы пользовательского интерфейса (например, метрик, безопасности, аудита и возможности идентификации сервисов).

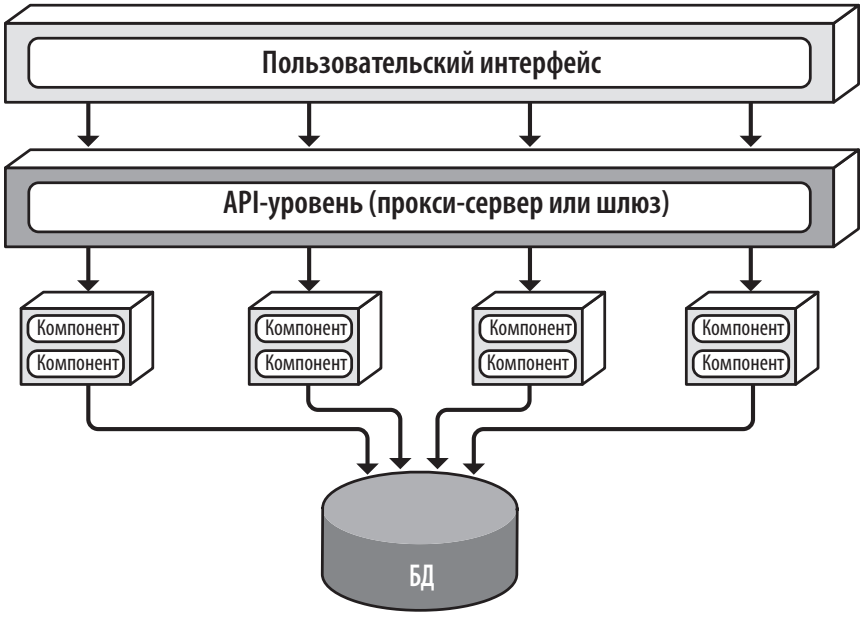


Рис. 13.4. Добавление API-уровня между пользовательским интерфейсом и сервисами предметных областей

Дизайн сервисов и гранулярность

Поскольку сервисы предметной области в архитектуре на основе сервисов обычно крупномодульные, каждый из них разрабатывается с использованием многоуровневого стиля архитектуры, состоящего из уровня внешнего API-интерфейса, уровня бизнес-логики и уровня сохранения данных. Еще один популярный подход к проектированию заключается в предметном разбиении каждого сервиса на основе предметных подобластей, что похоже на модульный монолитный архитектурный стиль. Эти подходы к проектированию сервисов показаны на рис. 13.5.

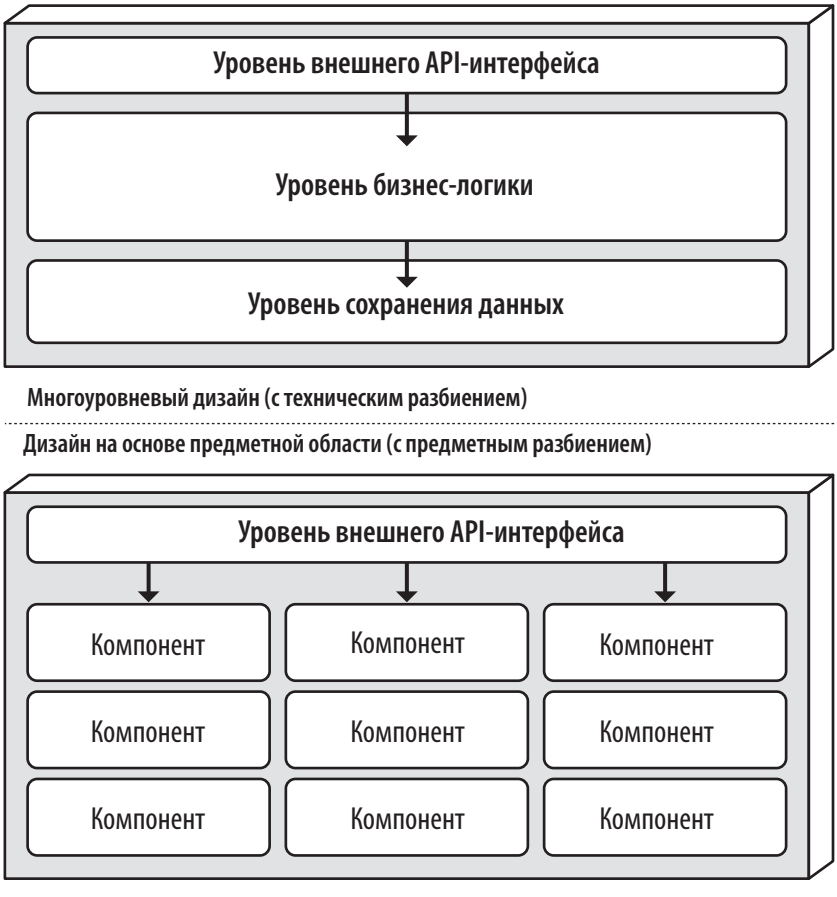


Рис. 13.5. Варианты дизайна сервисов предметной области

Независимо от дизайна, сервис предметной области должен содержать какой-нибудь интерфейс доступа к API, с которым взаимодействует пользовательский интерфейс для выполнения определенных бизнес-функций. Интерфейс доступа к API обычно отвечает за управление бизнес-запросами, поступающими от пользовательского интерфейса. Рассмотрим, к примеру, бизнес-запрос по размещению заказа (оформление заказа по каталогу). Этот единичный запрос, полученный внешним средством доступа к API-интерфейсу в предметном сервисе `OrderService`, выполняет единый бизнес-запрос: разместить заказ, создать идентификатор заказа, запустить механизм оплаты и обновить сведения о складских запасах заказанного товара. В архитектурном стиле микросервисов выполнение запроса было бы похоже на запуск управления множеством отдельно развернутых удаленных узкоспециализированных

сервисов. Такое различие между внутренним управлением на уровне классов и внешним управлением на уровне сервисов является одним из многих существенных различий между архитектурой на основе сервисов и микросервисами с точки зрения гранулярности.

Из-за крупномодульности предметных сервисов обычные ACID-транзакции базы данных (отвечающие принципам атомарности, достоверности, изолированности и живучести) связаны с подтверждениями и откатами в рамках отдельно взятого предметного сервиса. С другой стороны, архитектура с высоким уровнем распределенности, такая как микросервисы, обладает более высокой степенью детализации сервисов и использует технологию распределенной транзакции, известную как BASE-транзакция (basic availability — стандартная доступность, soft state — гибкое состояние, eventual consistency — окончательная согласованность), которая полагается на окончательную согласованность, поэтому в ней не поддерживается такой же уровень сохранности базы данных, как у ACID-транзакций в архитектуре на основе сервисов.

Проиллюстрируем это на примере оформления заказа по каталогу в приложении с архитектурой на основе сервисов. Предположим, что покупатель оформляет заказ, а срок действия кредитной карты истек. Поскольку транзакция в одном и том же сервисе носит атомарный характер, все, что было добавлено в базу данных, может быть удалено путем отката с отправкой покупателю уведомления о невозможности платежа. А теперь рассмотрим аналогичный процесс в приложении с архитектурой микросервисов, состоящем из более мелких узкоспециализированных сервисов. Сначала запрос будет принят сервисом размещения заказа, `OrderPlacement`, который создаст заказ, сгенерирует идентификатор заказа и вставит заказ в таблицы заказов. После того как он все это сделает, он отправит удаленный вызов платежному сервису `PaymentService`, который попытается провести платеж. Если платеж не пройдет из-за истечения срока действия карты, заказ нельзя будет разместить и данные окажутся в подвешенном состоянии (так как информация о заказе уже была вставлена, но еще не подтверждена). Как в таком случае следует поступить с запасом заказанного товара? Пометить товар как заказанный и уменьшить запас на единицу? А если его запас невелик и данный товар желает приобрести другой покупатель? Нужно ли разрешить приобретение этого товара другим покупателем или же следует отложить зарезервированный запас товара для того, кто пытается оформить заказ с просроченной картой? Это только лишь часть вопросов, требующих разъяснения при управлении бизнес-процессами множеством узкоспециализированных сервисов.

Будучи крупномодульными, предметные сервисы позволяют обеспечить более высокий уровень достоверности и согласованности данных, но из-за этого приходится идти на компромисс. В архитектуре на основе сервисов изменения,

внесенные в функцию размещения заказа в `OrderService`, могут потребовать тестирования всего крупномодульного сервиса (включая обработку платежа), а вот в микросервисах подобное изменение окажет влияние только на небольшой сервис размещения заказа `OrderPlacement` (и не потребует внесения изменений в сервисе платежей `PaymentService`). Кроме того, из-за развертывания более объемного кода в архитектуре на основе сервисов будет выше вероятность что-либо сломать (включая обработку платежа), а в микросервисах каждый сервис выполняет только одну задачу, следовательно, при его изменении значительно меньше шансов навредить другим функциям.

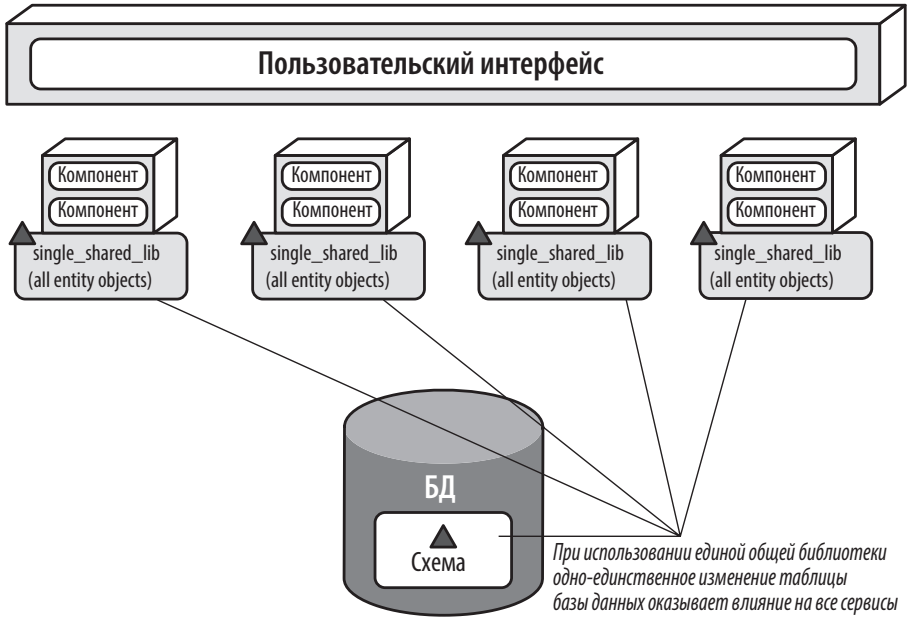
Разбиение базы данных

Хотя это и необязательно, в архитектуре на основе сервисов все сервисы приложения из-за их небольшого количества (от 4 до 12) используют единую монолитную базу данных. Такая связанность БД способна создавать проблему при изменении схемы таблиц этой базы. Неосмотрительность может привести к тому, что изменение в схеме, внесенное некорректно, повлияет на каждый сервис, и это увеличит затраты в плане трудоемкости и согласованности.

В архитектуре на основе сервисов схемы таблиц БД (обычно называемые *объектами сущностей — entity objects*), представленные в файлах совместно используемых классов, размещаются в специализированных общих библиотеках, используемых всеми предметными сервисами (например, в JAR-файле или в файле DLL). В общих библиотеках может также содержаться SQL-код. Практика создания единой общей библиотеки объектов сущностей — самый неэффективный способ реализации архитектуры на основе сервисов. Любое изменение в структурах таблиц базы данных приведет к изменениям в единой общей библиотеке, содержащей все необходимые объекты сущностей, что потребует модификации и повторного развертывания каждого сервиса независимо от того, имеются ли в каких-то из них обращения к измененной таблице. Решить эту проблему можно с помощью управления версиями общей библиотеки, но узнать, на какие именно сервисы повлияло это изменение, без ручного детального анализа будет практически невозможно. Сценарий с использованием единой общей библиотеки показан на рис. 13.6.

Один из способов уменьшения влияния и снижения риска от изменений, вносимых в БД, заключается в логическом разбиении базы данных с помощью интегрированных общих библиотек. Заметьте, что на рис. 13.7 база логически разбита на пять отдельных предметных областей (общую, клиентскую, выставления счетов, заказов и отслеживания). Обратите также внимание на то, что пять соответствующих общих библиотек, используемых предметными сервисами,

соответствуют логическому разбиению базы данных. При использовании данного приема изменения в таблице конкретной логической предметной области (в данном случае выставление счета) сопоставляются с соответствующей общей библиотекой, содержащей объекты сущностей (а также, возможно, и SQL-код), оказывая влияние только на те сервисы, которые пользуются данной общей библиотекой, то есть в данном случае на сервис выставления счетов. Ни на какие другие сервисы это изменение не влияет.



single_shared_lib — единая общая библиотека со всеми объектами сущностей

Рис. 13.6. Использование единой общей библиотеки для объектов сущностей базы данных

Обратите внимание, что на рис. 13.7 показано применение *общей (common)* предметной области и соответствующей ей общей библиотеки `common_entities_lib`, используемой всеми сервисами. Это вполне обычное явление. Такие таблицы являются общими, поэтому изменения в них требуют согласованности во всех сервисах, обращающихся к единой базе данных. Один из способов ограничить внесение изменений в эти таблицы (и в соответствующие объекты сущностей) — заблокировать общие объекты сущностей в системе управления версиями и предоставить доступ к внесению изменений только команде, работающей с базой данных. Это помогает установить контроль над изменениями и уделять особое внимание вносимым правкам.

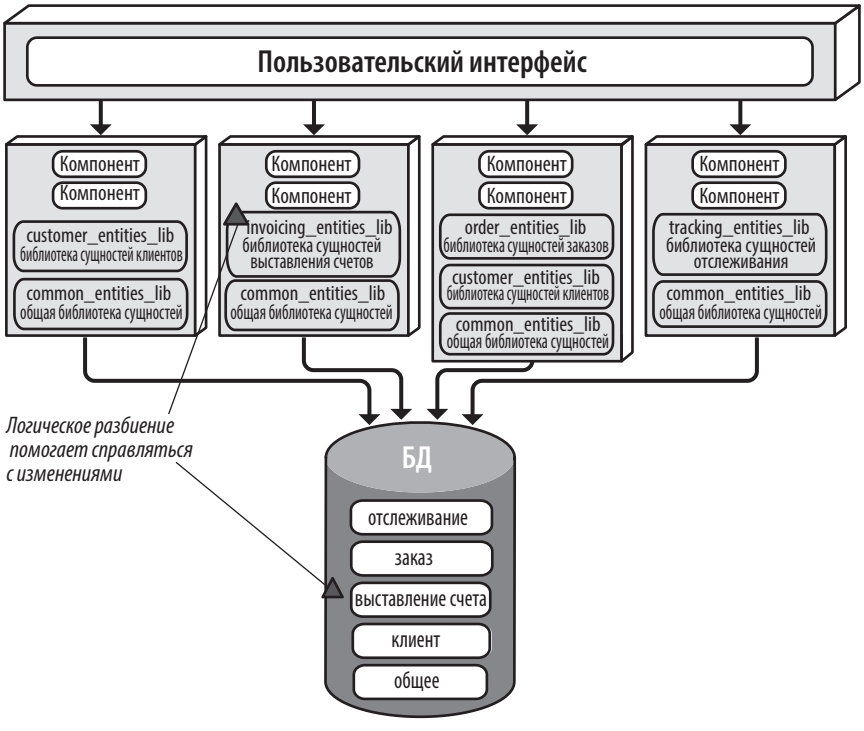


Рис. 13.7. Использование нескольких общих библиотек для объектов сущностей базы данных

Сделайте логическое разбиение базы данных как можно более детальным, сохраняя при этом четкое обозначение области данных, чтобы лучше контролировать изменения базы данных в архитектуре на основе сервисов.

Пример архитектуры

Чтобы проиллюстрировать гибкость и эффективность архитектурного стиля на основе сервисов, рассмотрим реальный пример системы утилизации подержанных электронных устройств (например, мобильных телефонов iPhone или Galaxy). Технологический процесс переработки происходит следующим образом: сначала клиент спрашивает у компании (через веб-сайт или интерактивный терминал), сколько он может выручить денег за подержанный телефон (это называется *котировкой — quoting*). Если предложение клиента устраивает, он отправляет электронное устройство компании, занимающейся утилизацией, которая, в свою очередь, получает его физически (это называется *получением — receiving*).

После получения устройства компания делает заключение, находится ли оно в хорошем рабочем состоянии или нет (это называется *оценкой — assesment*). Если устройство находится в хорошем рабочем состоянии, компания отправит клиенту обещанную сумму (это называется *расчетом — accounting*). В ходе данного процесса клиент в любое время может зайти на веб-сайт и проверить состояние товара (это называется *статусом заказа — item status*). На основании заключения устройство утилизируется либо путем безопасного уничтожения, либо путем перепродажи (это называется *утилизацией — recycling*). И наконец, на основе утилизационной деятельности компания периодически составляет специализированные и плановые финансовые и операционные отчеты (это называется *составлением отчетов — reporting*).

Такая система с архитектурой на основе сервисов показана на рис. 13.8. Обратите внимание, как каждая предметная область, указанная в предыдущем описании,

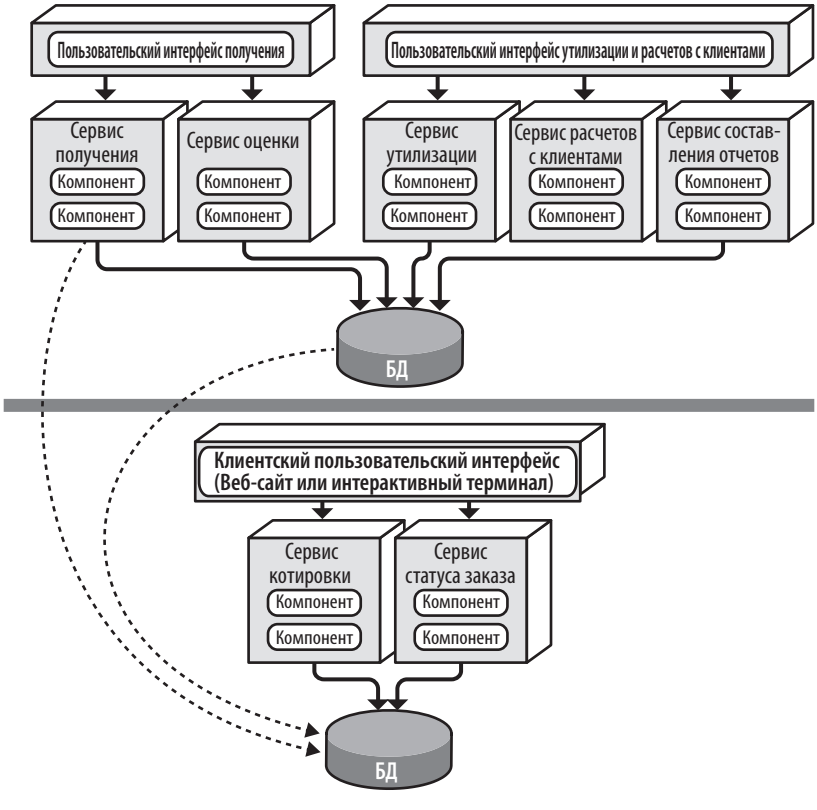


Рис. 13.8. Пример приложения для утилизации электронных устройств, использующего архитектуру на основе сервисов

реализована в виде отдельно развернутого независимого предметного сервиса. Масштабируемость может быть достигнута за счет масштабирования только тех сервисов, для которых требуется более высокая пропускная способность (в данном случае это ориентированные на клиента сервисы оценки `Quoting` и статуса заказа `ItemStatus`). Остальные сервисы не нуждаются в масштабировании, и поэтому им требуется всего один экземпляр.

Обратите внимание, как на рис. 13.8 прикладные уровни пользовательского интерфейса интегрируются в свои соответствующие предметные области: *клиентскую, получения*, *утилизации* и *расчетов*. Этим объединением обеспечивается отказоустойчивость пользовательского интерфейса, масштабируемость и безопасность (у внешних клиентов отсутствует сетевой доступ к внутренним функциям). И наконец, заметьте, что в этом примере используются две отдельные физические базы данных: одна для внешних операций по обслуживанию клиентов, а другая — для внутренних операций. Это позволяет внутренним данным и операциям размещаться в отдельной сетевой зоне, обособленной от внешних операций (что обозначено вертикальной линией), чем обеспечиваются гораздо более надежные ограничения доступа и защита данных. Односторонний доступ через брандмауэр открывает внутренним сервисам доступ к клиентской информации и к ее обновлению, но никак не наоборот. В зависимости от используемой базы данных можно также воспользоваться внутренним зеркальным отображением таблиц и их синхронизацией.

Этим примером иллюстрируются многие преимущества архитектуры на основе сервисов: масштабируемость, отказоустойчивость и безопасность (защита и доступ к данным и функциям) в дополнение к гибкости, тестируемости и развертываемости. Например, служба оценки `Assessment` подвергается постоянным изменениям из-за добавления правил оценки по мере поступления новых товаров. Эти частые изменения происходят в рамках одного-единственного предметного сервиса, обеспечивая гибкость (способность быстро реагировать на изменения), а также тестируемость (простоту и полноту тестирования) и развертываемость (простоту, частоту и низкий риск развертываний).

Оценки архитектурных свойств

Одна звезда в оценочной таблице свойств (показанной на рис. 13.9) означает, что данное архитектурное свойство плохо поддерживается в архитектуре, а пять звезд — что архитектурное свойство является одной из самых сильных сторон архитектурного стиля. Определение каждого свойства, указанного в оценочной таблице, можно найти в главе 4.

Архитектурное свойство	Оценка в звездах
Тип разбиения	Предметный
Количество квантов	От одного до нескольких
Развертываемость	★★★★
Адаптируемость	★★
Эволюционность	★★★
Отказоустойчивость	★★★★
Модульность	★★★★
Общие затраты	★★★★
Производительность	★★★
Надежность	★★★★
Масштабируемость	★★★
Простота	★★★
Тестируемость	★★★★

Рис. 13.9. Оценки свойств архитектуры на основе сервисов

Архитектура на основе сервисов относится к тем, которые разбиваются *по предметному признаку*, то есть структура формируется по предметным областям, а не из технических соображений (таких, как логика представления или логика сохранения данных). Рассмотрим предыдущий пример приложения для утилизации электронных устройств. Каждый сервис, являясь отдельно развернутым модулем программы, ограничен конкретной предметной областью (например, оценкой устройства). Изменения, внесенные в рамках этой предметной области, оказывают влияние только на конкретный сервис, его пользовательский интерфейс и связанную с ним базу данных. Подвергать изменениям что-либо еще с целью поддержки конкретного изменения, внесенного в процесс оценки, не нужно.

Так как архитектура относится к распределенным системам, количество квантов может быть от одного и больше. Даже при наличии от 4 до 12 отдельно разверну-

тых сервисов, система будет представлять собой только один квант при условии, что все эти сервисы совместно используют одну и ту же базу данных или единый пользовательский интерфейс. Но как было показано выше в разделе «Варианты топологии» на с. 204, пользовательский интерфейс и база данных могут быть интегрированы, и тогда в рамках всей системы будет несколько квантов. В примере приложения по утилизации электронных устройств в системе имеются два кванта (рис. 13.10): один для той части приложения, что связана с клиентом и содержит отдельный клиентский пользовательский интерфейс, а также набор сервисов (котировки `Quoting` и статуса заказа `Item Status`); и другой, предназначенный для внутренних операций получения устройства, оценки и утилизации. Заметьте, что даже при этом квант внутренних операций состоит из отдельно развернутых сервисов и двух отдельных пользовательских интерфейсов, которые совместно используют одну и ту же базу данных, формируя единый квант из той части приложения, которая занимается внутренними операциями.

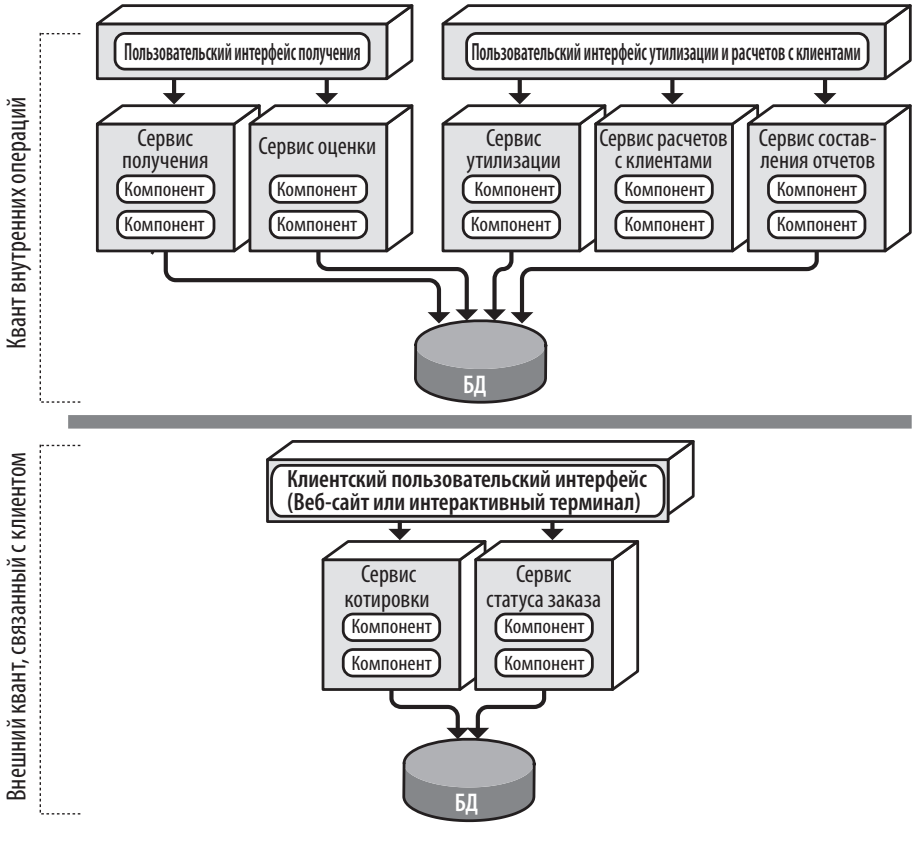


Рис. 13.10. Отдельные кванты в архитектуре на основе сервисов

Хотя пятизвездочных оценок в архитектуре на основе сервисов нет, она все же довольно высоко (четыре звезды) оценивается во многих жизненно важных областях. Разбиение приложения на несколько отдельно развертываемых предметных сервисов при использовании данного архитектурного стиля позволяет оперативнее вносить изменения (гибкость); ограниченность предметной области обеспечивает лучшее покрытие тестами (тестируемость); а возможность более частых развертываний уменьшает связанные с ними риски по сравнению с крупной монолитной системой (развертываемость). Эти три свойства позволяют сократить время вывода продукта на рынок, помогая организации внедрять новые функции и устранять недочеты в относительно короткие сроки.

Также в архитектуре на основе сервисов высоко оценены отказоустойчивость и общая доступность приложения. Даже при том, что у предметных сервисов наблюдается склонность к крупномодульности, причиной оценки в четыре звезды является автономность сервисов и отсутствие межсервисного обмена данными из-за совместно используемого кода и общей базы данных. В результате если один сервис выходит из строя (например, сервис получения `Receiving` в примере приложения по утилизации электронных устройств), это не оказывает никакого влияния на любой из шести остальных сервисов.

Масштабируемость из-за крупномодульности сервисов оценивается только в три звезды, а у адаптируемости, соответственно, только две звезды. Повысить эти показатели программными средствами, конечно, можно, но при этом дублируется больше функций, чем при использовании узкоспециализированных сервисов (например, микросервисов), что снижает эффективность масштабирования в плане потребления машинных ресурсов и повышает затраты. Обычно в архитектуре на основе сервисов достаточно иметь только по одному экземпляру каждого сервиса, если только не понадобится более широкая пропускная способность или высокая отказоустойчивость. Хорошим примером в данном случае может послужить приложение по утилизации электронных устройств: в масштабировании для поддержки большого наплыва клиентов нуждаются только сервисы котировки `Quoting` и статуса заказа `Item Status`, а другим рабочим сервисам нужны только единственные экземпляры. Это упрощает поддержку единого кэширования в оперативной памяти и создания пула подключений к базе данных.

К двум другим факторам, отличающим этот стиль архитектуры от более затратных и сложных распределенных архитектур (микросервисов, архитектур, ориентированных на события, и даже архитектур на основе пространства), относятся простота и низкие общие затраты. Благодаря этому архитектура на основе сервисов является одной из самых простых и наименее затратных из всех распределенных архитектур. Несмотря на всю привлекательность данного

обстоятельства, во всех свойствах, оцененных в четыре звезды, все же прослеживается некий компромисс между экономией и простотой. Чем выше затраты и сложность, тем выше становятся оценки.

Как правило, благодаря крупномодульности предметных сервисов, архитектуры на основе сервисов отличаются более высокой надежностью по сравнению с другими распределенными архитектурами. Для более крупных сервисов характерен меньший сетевой трафик на маршрутах, как направленных к сервисам, так и проложенных между ними, меньший объем распределенных транзакций и меньшая пропускная способность, что увеличивает общую сетевую надежность.

Когда выбирать этот архитектурный стиль

Гибкость этого архитектурного стиля (см. выше раздел «Варианты топологии» на с. 204) в сочетании с количеством трех- и четырехзвездочных оценок переводит архитектуру на основе сервисов в разряд наиболее прагматичных и доступных архитектурных стилей. Разумеется, существуют иные, гораздо более эффективные стили распределенной архитектуры, но некоторые компании считают, что эта эффективность обойдется им слишком дорого, а другие — что столь высокий уровень эффективности им просто не нужен. Это сродни тому, как мощность, скорость и маневренность автомобиля марки Феррари использовались бы только для езды в час пик на скорости 50 километров в час. Выглядит, конечно, круто, но каков расход ресурсов и денег!

Архитектура на основе сервисов также хорошо вписывается в предметно-ориентированное проектирование. Из-за крупномодульности сервисов и их функциональной ограниченности каждая предметная область вполне удачно помещается в отдельно развернутом предметном сервисе. Каждый сервис в такой архитектуре охватывает конкретное направление (например, утилизацию в приложении по утилизации электронных устройств), поэтому извлечение конкретной функциональности в отдельный модуль приложения упрощает внесение изменений в охватываемую им предметную область.

Поддержание и координация транзакций баз данных всегда были проблемным вопросом распределенных архитектур, поскольку в них обычно полагаются на окончательную согласованность, а не на традиционные ACID-транзакции (отвечающие принципам атомарности, достоверности, изолированности и живучести). Но благодаря крупномодульности предметных сервисов, архитектура на основе сервисов соблюдает принципы ACID-транзакций строже, чем любая другая распределенная архитектура. Случается, что пользовательский интерфейс или шлюз API может управлять двумя или более предметными сервисами, и тогда

транзакция должна полагаться на череду событий и на BASE-принципы. И все же в большинстве случаев транзакция привязана к определенному предметному сервису, что позволяет воспользоваться традиционными функциями коммита и отката транзакции, имеющимися в большинстве монолитных приложений.

Наконец, архитектура на основе сервисов станет удачным выбором для достижения высокого уровня архитектурной модульности без всех сложностей и тонкостей гранулярности. Как только сервисы приобретают более узкоспециализированный характер, начинают проявляться проблемы, связанные с управлением и согласованием действий, условно называемые оркестровкой (orchestration) и хореографией (choreography). И то и другое требуется, когда возникают потребности в скоординированной работе нескольких сервисов для завершения конкретной бизнес-транзакции. Оркестровка представляет собой координацию работы нескольких сервисов посредством использования отдельного сервиса-посредника, контролирующего рабочий поток транзакции и управляющего этим потоком (наподобие дирижера в оркестре). А хореография, по сути, является координацией работы нескольких сервисов, при которой все сервисы общаются друг с другом без использования центрального посредника (как танцоры в танце). По мере сужения специализации сервисов возрастает и необходимость в оркестровке и хореографии при связывании сервисов воедино для завершения бизнес-транзакции. Но поскольку сервисы в обсуждаемом стиле архитектуры чаще всего бывают более крупномодульными, им не нужен такой серьезный уровень координации, без которого не обходится практически ни одна другая распределенная архитектура.

ГЛАВА 14

Архитектура, управляемая событиями

Стиль архитектур, *управляемых событиями* (event-driven architecture), является популярным стилем распределенной асинхронной архитектуры, которая используется для создания высокопроизводительных масштабируемых приложений. Он отличается также высокой адаптируемостью и может использоваться как для небольших приложений, так и для крупных и сложных. Архитектура, управляемая событиями, состоит из отдельных компонентов, которые асинхронно получают и обрабатывают события. Она может использоваться как сама по себе, так и встраиваться в другие архитектурные стили (например, при создании микросервисной архитектуры, ориентированной на события).

Обычно используется так называемая *модель, основанная на запросах* (*request-based model*), показанная на рис. 14.1. В этой модели все запросы на выполнение

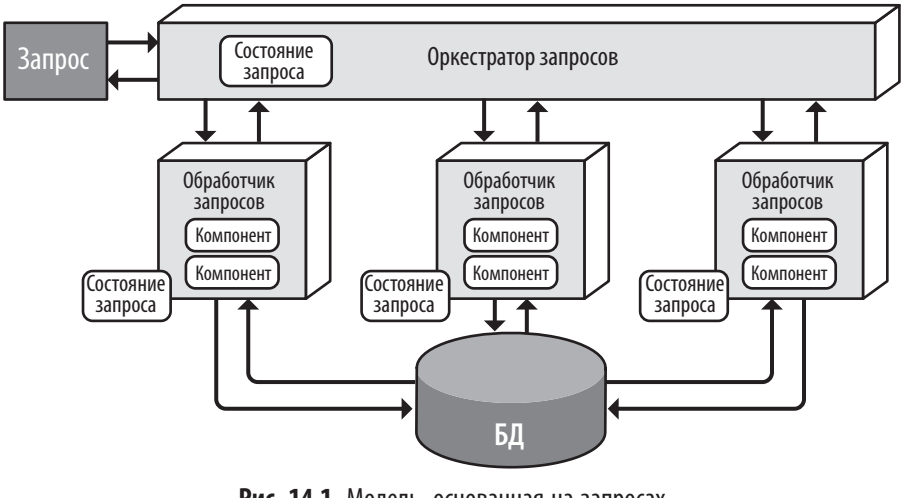


Рис. 14.1. Модель, основанная на запросах

каких-либо действий, направляемые в адрес системы, отправляются к *оркестратору запросов (request orchestrator)*. Таким оркестратором обычно является пользовательский интерфейс, но он может быть также реализован в виде API-уровня или в виде сервисной шины предприятия. Задачей оркестратора запросов является предопределенное и синхронное направление запросов к различным *обработчикам запросов (request processors)*. В свою очередь, их задача заключается либо в извлечении информации из базы данных, либо в ее обновлении.

Хорошим примером модели, основанной на запросах, будет запрос от клиента на получение истории своих заказов за последние шесть месяцев. Получение информации об истории заказов — это управляемый данными, строго определенный запрос к системе для получения конкретной информации, а не событие, на которое система должна реагировать.

Модель, основанная на событиях, напротив, реагирует на конкретную ситуацию, в результате чего на базе того или иного события предпринимается конкретное действие. Примером такой модели является отправка ставки на конкретный лот онлайн-аукциона. Эта ставка является не запросом, адресованным системе, а событием, случившимся после объявления текущей цены предложения. Система должна ответить на это событие сравнением с другими ставками, поданными в это же время, чтобы определить самую высокую выставленную цену.

Топология

В архитектуре, ориентированной на события, используются две основные топологии: *медиатора* (*mediator topology*) и *брокера* (*broker topology*). Топология медиатора чаще всего используется там, где требуется контроль над рабочим процессом событий, а топология брокера — там, где требуется быстрая реакция и динамический контроль над обработкой событий. Поскольку архитектурные свойства и стратегии реализации этих двух топологий отличаются друг от друга, важно разобраться в том, какая из них больше подходит для конкретной ситуации.

Топология брокера

Топология брокера отличается от топологии медиатора тем, что в ней нет центрального медиатора событий. Поток сообщений распределяется по компонентам обработчиков событий в виде цепочки сообщений, проходящей через небольшой почтовый агент (например, RabbitMQ, ActiveMQ, HornetQ и т. д.). Эта

топология применяется при наличии относительно простой обработки событий и отсутствии необходимости в централизованной оркестровке и координации.

В топологии брокера имеются четыре основных архитектурных компонента: инициирующее событие (initiating event), брокер событий (event broker), обработчик событий (event processor) и событие обработки (processing event). *Инициирующее событие* является исходным событием, запускающим весь поток операций, будь то простое событие, например размещение заявки в онлайн-аукционе, или же более сложные события для системы медицинского страхования, такие как смена работы или женитьба. Инициирующее событие отправляется для обработки в канал событий в *брокере событий*. В этой топологии нет компонента-медиатора, управляющего событием и контролирующего его прохождение, поэтому от брокера инициирующее событие передается единственному *обработчику событий*. Обработчик, принявший инициирующее событие, выполняет конкретную задачу, связанную с обработкой этого события, а затем асинхронно сообщает остальной части системы о том, что он сделал, выдавая то, что называется *событием обработки*. После этого событие обработки, если нужно, асинхронно отправляется к брокеру событий для дальнейшей обработки. Другие обработчики событий прослушивают событие обработки, реагируют на него каким-либо действием, а затем объявляют с помощью нового события обработки о том, что они сделали. Этот процесс продолжается, пока никто не заинтересуется тем, что было сделано последним обработчиком событий. Такой рабочий процесс показан на рис. 14.2.

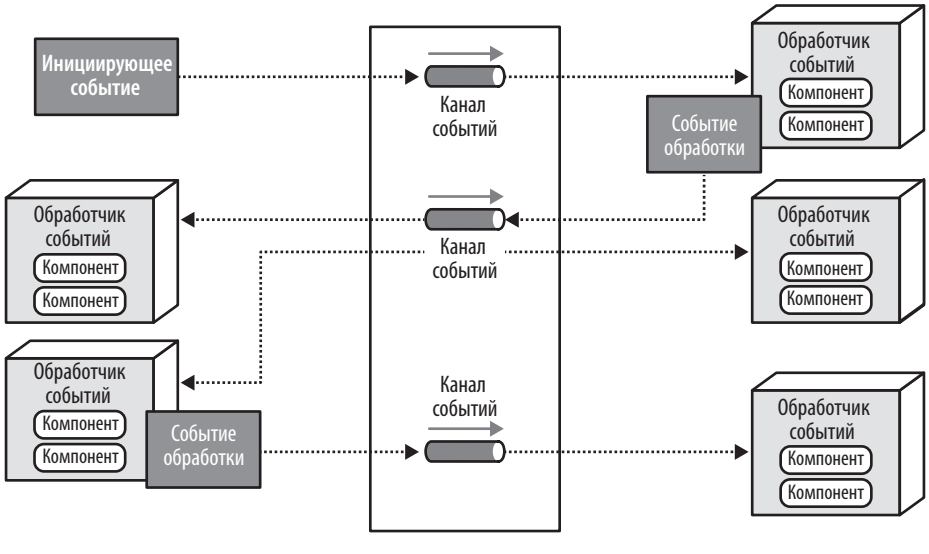


Рис. 14.2. Топология брокера

Компонент брокера событий обычно является интегрированным (состоящим из нескольких сгруппированных по предметной области экземпляров), где каждый подключенный брокер содержит все каналы событий, используемые в потоке событий для данной конкретной предметной области. Поскольку топология брокера имеет предопределенную асинхронную природу вещания по принципу «выстрелил и забыл», топики (или обмен топиками в случае с AMQP), обычно применяемые в топологии брокера, используют модель сообщений с публикацией и подпиской (publish-and-subscribe messaging model).

В топологии брокера каждый обработчик событий сообщает о своих действиях остальной части системы независимо от того, интересен или нет результат его работы любому другому обработчику событий. Тем самым обеспечивается расширяемость архитектуры, если для обработки события потребуется дополнительная функциональность. Предположим, к примеру, что часть сложной обработки события (рис. 14.3) заключается в создании электронного письма и его отправке клиенту в качестве уведомления о предпринятом конкретном действии. Обработчик события уведомления `Notification` создаст и отправит электронное письмо, а затем объявит об этом всей остальной системе через новое событие обработки, отправленное в топик. Но в данном случае события этого топика никакими другими обработчиками событий не прослушиваются, и поэтому сообщение просто исчезает.

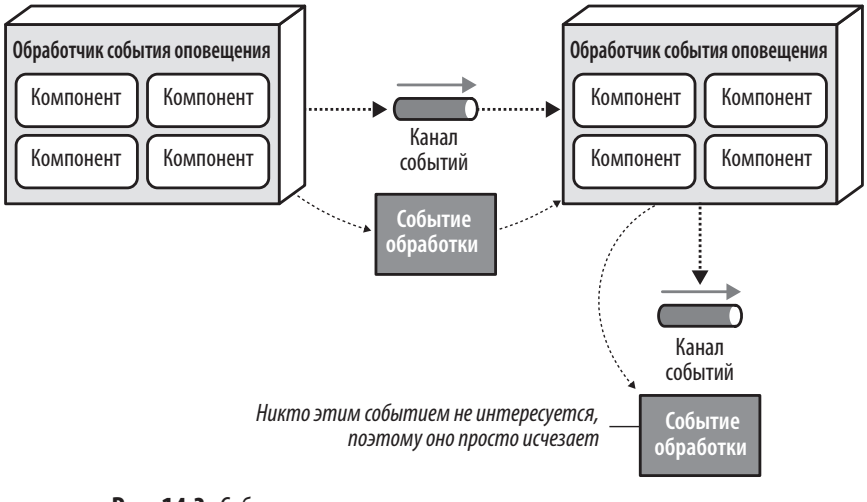


Рис. 14.3. Событие оповещения отправлено, но проигнорировано

Перед нами хороший пример *архитектурной расширяемости*. Отправка игнорируемых сообщений может кому-то показаться напрасной тратой ресурсов, но это не так. Предположим, что поступило новое требование по анализу электронных писем, отправленных клиентам. Новый обработчик событий может быть легко

добавлен к системе, поскольку информация доступна через соответствующую тему электронной почты каждому новому анализатору без дополнительной поддержки или внесения изменений в другие обработчики событий.

Чтобы показать, как работает топология брокера, рассмотрим процесс обработки в обычной системе ввода розничных заказов (рис. 14.4), где размещен заказ на товар (скажем, на такую же книгу, как эта). В данном примере обработчик события размещения заказа `OrderPlacement` получает инициирующее событие, `PlaceOrder`, вставляет заказ в таблицу базы данных и возвращает клиенту идентификационный номер заказа. Затем вся остальная система оповещается о том, что был создан заказ, через событие обработки `order-created`. Заметьте, что данным событием интересуются три обработчика событий: уведомление — `Notification`, платеж — `Payment` и запас товаров — `Inventory`. Все они выполняют свои задачи в параллельном режиме.

Обработчик события уведомления `Notification` получает событие обработки `order-created`, свидетельствующее о создании заказа, и отправляет электронное письмо клиенту. Затем он выдает другое событие обработки, `email-sent` (отправка электронного письма). Заметьте, что это событие не прослушивается никакими другими обработчиками событий. Так и должно быть, тем самым подтверждается предыдущий пример с описанием расширяемости архитектуры — готовый к использованию хук, позволяющий другим обработчикам при необходимости подключиться к этому потоку событий.

Обработчик события изменения запасов товара `Inventory` также отслеживает событие обработки `order-created` и сокращает на единицу запас (количество экземпляров) нашей книги. Затем он объявляет об этом действии с помощью выдачи события обработки `inventory-updated`, которое, в свою очередь, воспринимается обработчиком события изменения складских остатков `Warehouse` для управления запасами товара на разных складах, перераспределяя товары при недопустимом снижении запаса.

Обработчик события платежа `Payment` также получает событие обработки `order-created` и списывает средства с кредитной карты клиента в соответствии с только что созданным заказом. Заметьте, что в результате действий, предпринятых обработчиком событий `Payment`, выдается одно из двух событий: первое предназначено для уведомления всей остальной системы, что платеж был принят (`payment-applied`), а второе является событием обработки, оповещающим всю остальную систему об отклонении платежа (`payment-denied`). Обратите внимание, что обработчик события оповещения `Notification` заинтересован в событии обработки `payment-denied`, поскольку он должен, в свою очередь, отправить электронное сообщение клиенту с предупреждением о необходимости обновить информацию о кредитной карте или выбрать другой метод оплаты.

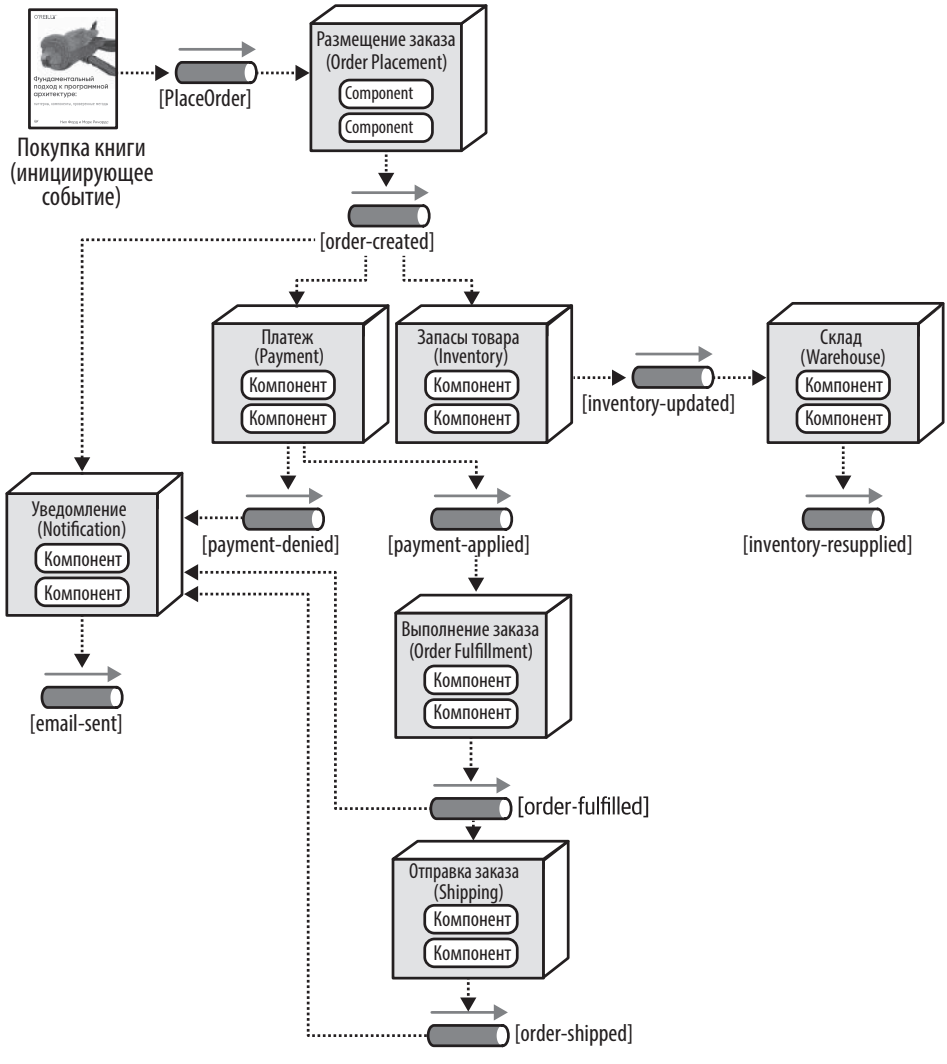


Рис. 14.4. Пример топологии брокера (в квадратных скобках указаны события, а в прямоугольниках — обработчики событий)

Обработчик события выполнения заказа `OrderFulfillment` отслеживает событие обработки `payment-applied` и контролирует комплектацию и упаковку заказа. Как только они будут завершены, он оповещает всю остальную систему о выполнении заказа, выдавая событие обработки `order-fulfilled`. Заметьте, что это событие отслеживается модулями обработки уведомления и отправки — `Notification` и `Shipping` соответственно. Одновременно обработчик

события `Notification` уведомляет клиента о выполнении заказа и его готовности к отправке, а обработчик события `Shipping` выбирает метод отправки. Обработчик события `Shipping` контролирует отправку заказа и выдает событие обработки `order-shipped`, которое также отслеживается обработчиком события `Notification` для оповещения клиента об изменении статуса заказа.

Изучая предыдущий пример, следует отметить, что все обработчики событий разделены (decoupled) и независимы друг от друга. Понять топологию брокера лучше всего на примере эстафеты. Бегуны в эстафете держат эстафетную палочку и пробегают определенную дистанцию (скажем, полтора километра), а затем передают палочку следующему бегуну, и так далее по цепочке, пока последний бегун не пересечет финишную черту. В эстафетах, как только бегун передаст палочку, его участие в забеге завершается и он принимается за другие дела. Точно так же происходит и в топологии брокера. Как только обработчик события передает событие дальше, он больше не занимается им и может реагировать на другие события инициирования или обработки. Кроме того, каждый обработчик событий может масштабироваться независимо от других обработчиков, чтобы адаптироваться к различным условиям нагрузки.

Большими преимуществами топологии брокера являются производительность, отзывчивость и масштабируемость, но в то же время существует ряд негативных моментов. Прежде всего, это отсутствие контроля над всем рабочим процессом, связанным с инициирующим событием (в данном случае с событием `PlaceOrder`). Рабочий процесс очень динамичен, зависит от различных условий, и к тому же в нем отсутствуют уведомления о фактическом завершении бизнес-транзакции по размещению заказа. Также в рамках топологии брокера весьма непросто проводить обработку ошибок. Поскольку промежуточное отслеживание или же обеспечение контроля над бизнес-транзакцией отсутствует, при сбое (например, при выходе из строя обработчика события `Payment` и невыполнении им своей задачи) нигде в системе об этом ничего не будет известно. Бизнес-процесс застопорится и не сможет продолжиться без какого-либо автоматического или ручного вмешательства. Более того, все остальные процессы пойдут без учета ошибки. Например, обработчик события `Inventory` как ни в чем не бывало снизит запас товара, и все остальные обработчики событий продолжат работать так, будто все в порядке.

Топологией брокера также не поддерживается и перезапуск бизнес-транзакций (возможность их восстановления). Поскольку исходная обработка инициирующего события запустила асинхронное выполнение других действий, повторный запуск инициирующего события невозможен. Ни один компонент в топологии брокера не осведомлен о состоянии дел или даже не владеет состоянием исходного бизнес-запроса, и поэтому ни один из компонентов в этой топологии не отвечает за перезапуск бизнес-транзакции (инициирующего события) и за

сведения о том, где эта транзакция застопорилась. Все достоинства и недостатки топологии брокера сведены в табл. 14.1.

Таблица 14.1. Достоинства и недостатки топологии брокера

Достоинства	Недостатки
Обработчики событий разделены	Отсутствие контроля над рабочим процессом
Высокая масштабируемость	Проблема обработки ошибок
Высокая отзывчивость	Отсутствие возможности восстановления
Высокая производительность	Невозможность перезапуска
Высокая отказоустойчивость	Угроза возникновения несогласованности данных

Топология медиатора

Топология медиатора в архитектуре, управляемой событиями, избавлена от ряда недостатков топологии брокера. Центральное место занимает медиатор событий, который управляет процессом и контролирует весь процесс обработки инициирующих событий, требующий согласованной работы нескольких обработчиков. Топология медиатора состоит из следующих архитектурных компонентов: инициирующее событие, очередь событий, медиатор событий, каналы событий и обработчики событий.

Как и в топологии брокера, инициирующее событие дает старт всему процессу обработки событий. Но в отличие от топологии брокера, инициирующее событие отправляется в очередь, которая принимается медиатором событий. Медиатору известно только то, что связано с обработкой события, и поэтому он генерирует соответствующие события обработки, которые отправляются в выделенные каналы событий (обычно по очереди) в режиме обмена сообщениями «точка — точка». Затем обработчики событий отслеживают выделенные каналы событий, обрабатывают событие и, как правило, возвращают медиатору оповещение о завершении своей работы. В отличие от топологии брокера, обработчики событий в топологии медиатора не объявляют всей системе о своих действиях. Топология медиатора показана на рис. 14.5.

Чаще всего в этой топологии имеется не один, а несколько медиаторов, связанных с конкретными предметными областями или группами событий. Это уменьшает типичную для данной топологии проблему, при которой отказ в одном-единственном месте приводит к отказу всей системы, а также расширяет общую пропускную способность и повышает производительность системы.

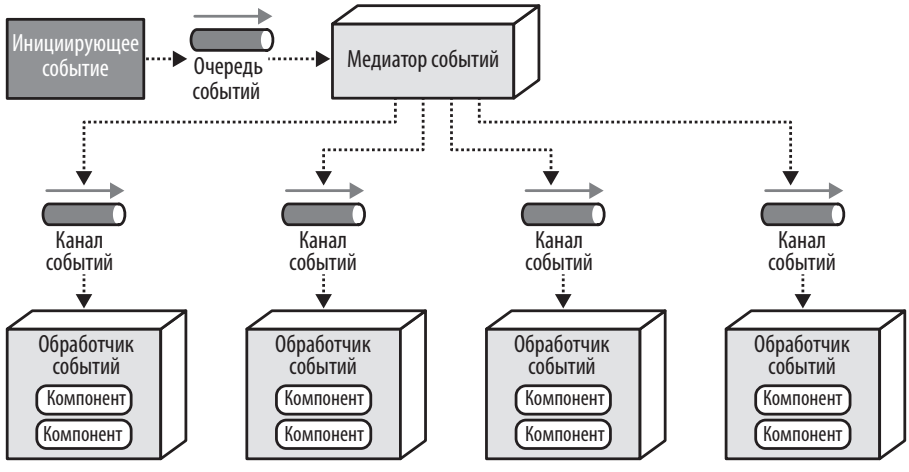


Рис. 14.5. Топология медиатора

Так, в системе может быть медиатор клиента, обрабатывающий все клиентские события (к примеру, при регистрации нового клиента и при обновлении профиля), и еще один медиатор, обрабатывающий все, что связано с заказами (при добавлении товара в корзину покупателя и при выборе товара по каталогу).

В зависимости от природы и сложности обрабатываемых событий, медиатор может быть реализован несколькими способами. Например, для событий, не требующих сложной обработки ошибок и оркестровки, обычно вполне хватает таких медиаторов, как Apache Camel[1], Mule ESB[2] или Spring Integration[3]. В медиаторах этих типов обычно имеются собственные потоки и маршруты сообщений, написанные на Java или C# для управления рабочим процессом обработки событий.

Но если процесс обработки событий требует большого объема разветвленной обработки и нескольких динамических маршрутов со сложными директивами обработки ошибок, то понадобятся такие медиаторы, как Apache ODE[4] или Oracle BPEL Process Manager[5]. Эти медиаторы основаны на использовании языка выполнения бизнес-процессов — Business Process Execution Language (BPEL), XML-подобной структуры, предоставляющей описание шагов, задействованных

[1] https://camel.apache.org/

[2] https://www.mulesoft.com/

[3] https://spring.io/projects/spring-integration

[4] https://ode.apache.org/

[5] https://www.oracle.com/middleware/technologies/bpel-process-manager.html

в обработке события. BPEL-артефакты также содержат структурированные элементы, используемые для обработки ошибок, перенаправления, многоадресной рассылки и т. д. BPEL довольно эффективный, но относительно сложный в изучении язык, поэтому его код обычно создается с помощью графического интерфейса, поставляемого в комплекте средств разработки BPEL.

BPEL может пригодиться для сложных и динамичных рабочих процессов, но он пасует перед процессами обработки событий, требующими длительных транзакций с участием человека. Предположим, к примеру, что сделка размещается посредством инициирующего события `place-trade`. Медиатор событий принимает это событие, но в ходе обработки выясняется, что сделка требует ручного подтверждения из-за превышения количества акций. Тогда медиатор событий должен будет остановить обработку события, отправить уведомление старшему трейдеру для ручного подтверждения и дождаться этого подтверждения. В таких случаях потребуется механизм управления бизнес-процессами — Business Process Management (BPM), например jBPM[1].

Чтобы выбрать подходящую реализацию медиатора событий, важно знать типы обрабатываемых событий. Если для сложных и длительных событий с вмешательством человека выбрать Apache Camel, то будет чрезвычайно тяжело писать и сопровождать программы. По тем же соображениям использование механизма BPM для простых потоков событий потребует месяцев напрасных усилий, когда то же самое в Apache Camel можно сделать за несколько дней.

Учитывая, что довольно редко все события относятся к одному классу сложности, мы рекомендуем классифицировать события как простые, сложные или комплексные и всегда проводить каждое событие через медиатор простых событий (например, Apache Camel или Mule). Затем этот медиатор может запросить классификацию события и на ее основе обработать событие самостоятельно или же перенаправить его другому медиатору, предназначенному для более сложных событий. Тогда можно будет выполнять эффективную обработку всех типов событий, привлекая для этого необходимый для конкретного события тип медиатора. Такая модель делегирования событий разным медиаторам показана на рис. 14.6.

На рис. 14.6 показано, что медиатор простых событий `Simple Event Mediator` генерирует и отправляет событие обработки, когда процесс обработки достаточно прост. Но при поступлении в `Simple Event Mediator` инициирующего события, классифицируемого как сложное или комплексное, этот медиатор перенаправляет исходное событие другим медиаторам (BPEL или BMP). Медиатор простых событий `Simple Event Mediator`, перехватив исходное событие,

[1] https://www.jbpm.org/

может по-прежнему отвечать за информацию о завершении его обработки или же просто делегировать весь рабочий процесс (включая уведомление клиента) другим медиаторам.

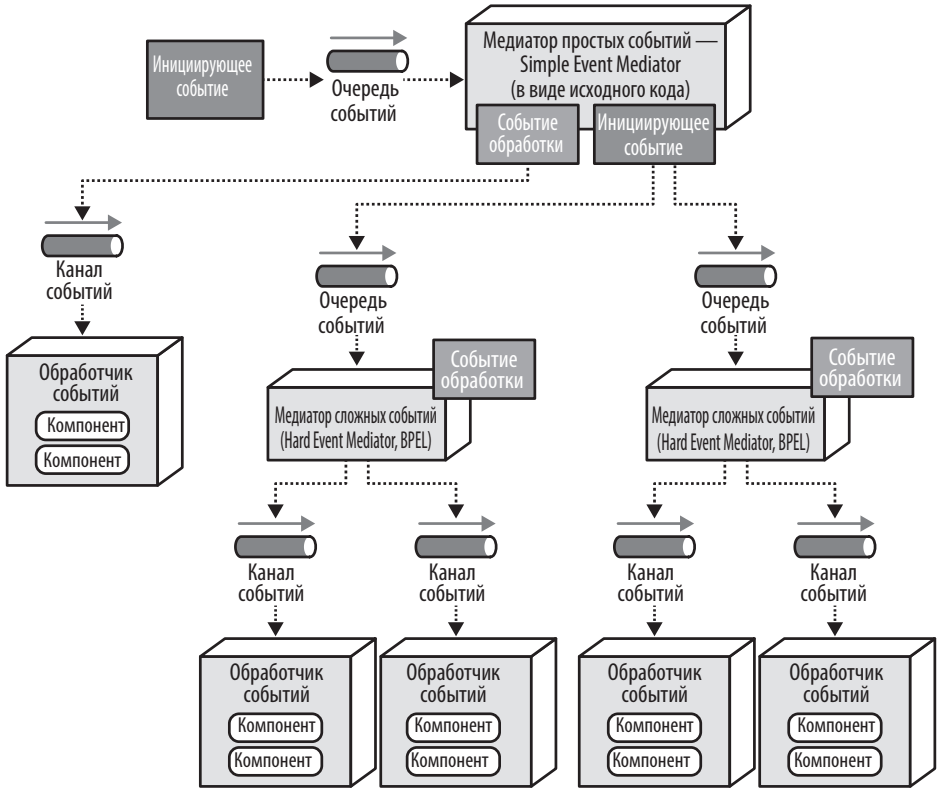


Рис. 14.6. Делегирование события подходящему типу медиатора событий

Чтобы показать, как работает топология медиатора, рассмотрим тот же пример системы ввода розничных заказов из раздела о топологии брокера, но на этот раз уже с применением топологии медиатора. В этом примере медиатору известны шаги обработки данного конкретного события. Этот процесс обработки события (происходящий внутри компонента медиатора) показан на рис. 14.7.

То же самое инициирующее событие, что и в предыдущем примере (`PlaceOrder`), отправляется для обработки в очередь клиентских событий `customer-event-`

queue. Медиатор Customer забирает это инициирующее событие и на основе процесса, показанного на рис. 14.7, начинает генерировать события обработки. Обратите внимание, что множественные события, показанные на шагах 2, 3 и 4, происходят параллельно в рамках шага, но последовательно между шагами. Другими словами, шаг 3 (выполнение заказа) должен быть завершен с соответствующим оповещением до того, как клиент на шаге 4 будет уведомлен о том, что заказ готов к отправке (отправка заказа).

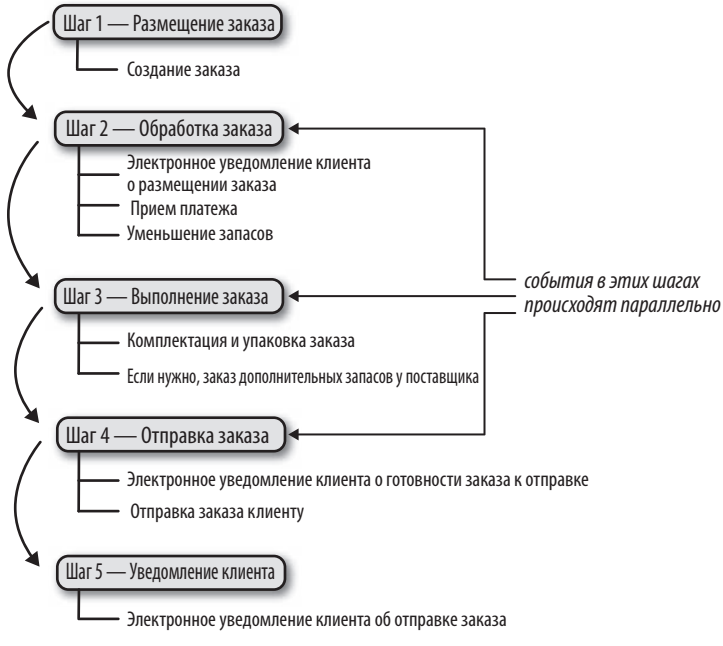


Рис. 14.7. Пошаговые действия медиатора для размещения заказа

Как только будет получено инициирующее событие, медиатор Customer сгенерирует событие обработки create-order и отправит соответствующее сообщение в очередь размещения заказа order-placement-queue (рис. 14.8). Обработчик событий OrderPlacement получает это событие, выполняет проверку и создает заказ, возвращая медиатору уведомление вместе с идентификатором заказа. Теперь медиатор может отправить этот идентификатор заказа клиенту, подтвердив размещение заказа, или же продолжить работу, пока не завершит выполнение всех шагов (это будет зависеть от определенных бизнес-правил размещения заказов).

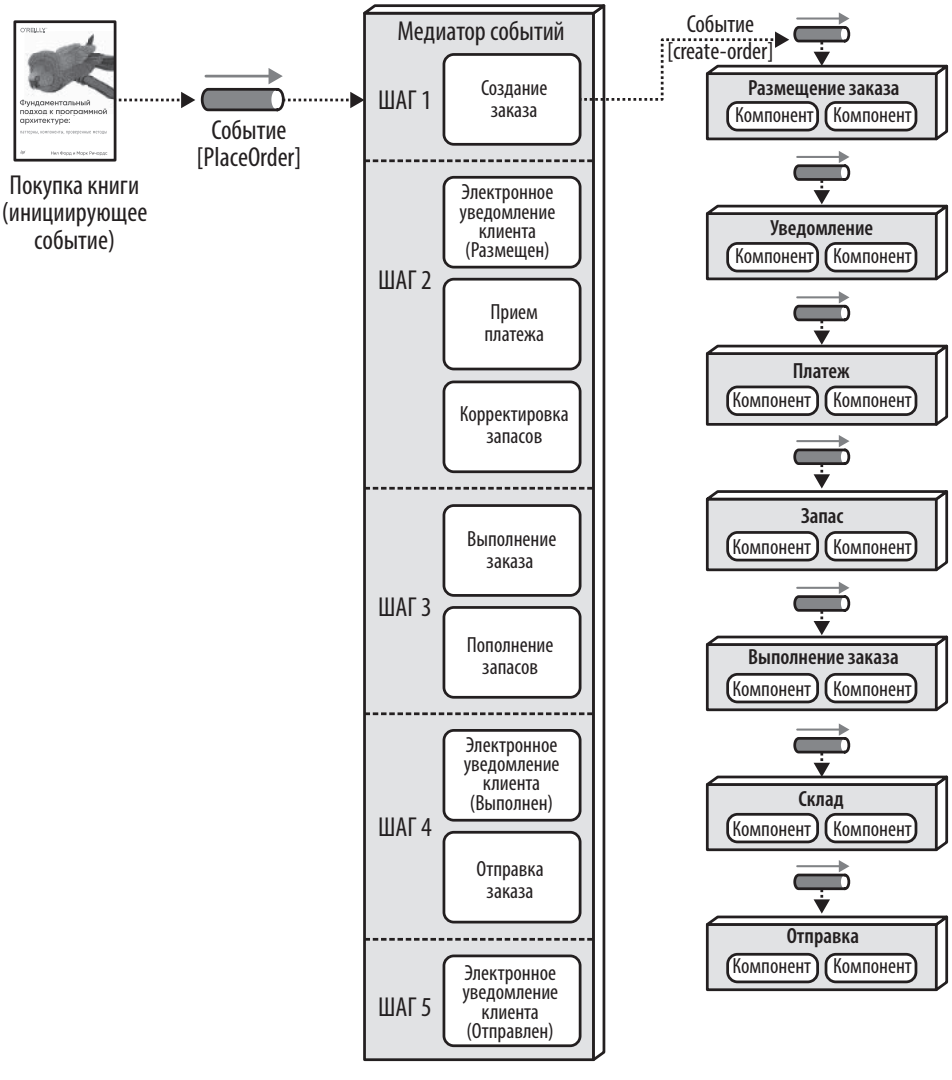


Рис. 14.8. Медиатор, шаг 1

Завершив работу по шагу 1, медиатор переходит к шагу 2 (рис. 14.9) и одновременно генерирует три сообщения: `email-customer`, `apply-payment` и `adjust-inventory`. Все эти события обработки отправляются в соответствующие очереди. Все три обработчика событий получают эти сообщения, выполняют соответствующие задачи и уведомляют медиатор о завершении своей обработки. Заметьте, что медиатор, прежде чем перейти к шагу 3, должен дождаться подтверждения от всех трех параллельных процессов. Теперь, если ошибка возникнет в одном

из параллельно функционирующих обработчиков событий, медиатор сможет предпринять корректирующие действия для устранения проблемы (чуть позже мы более подробно рассмотрим этот момент).

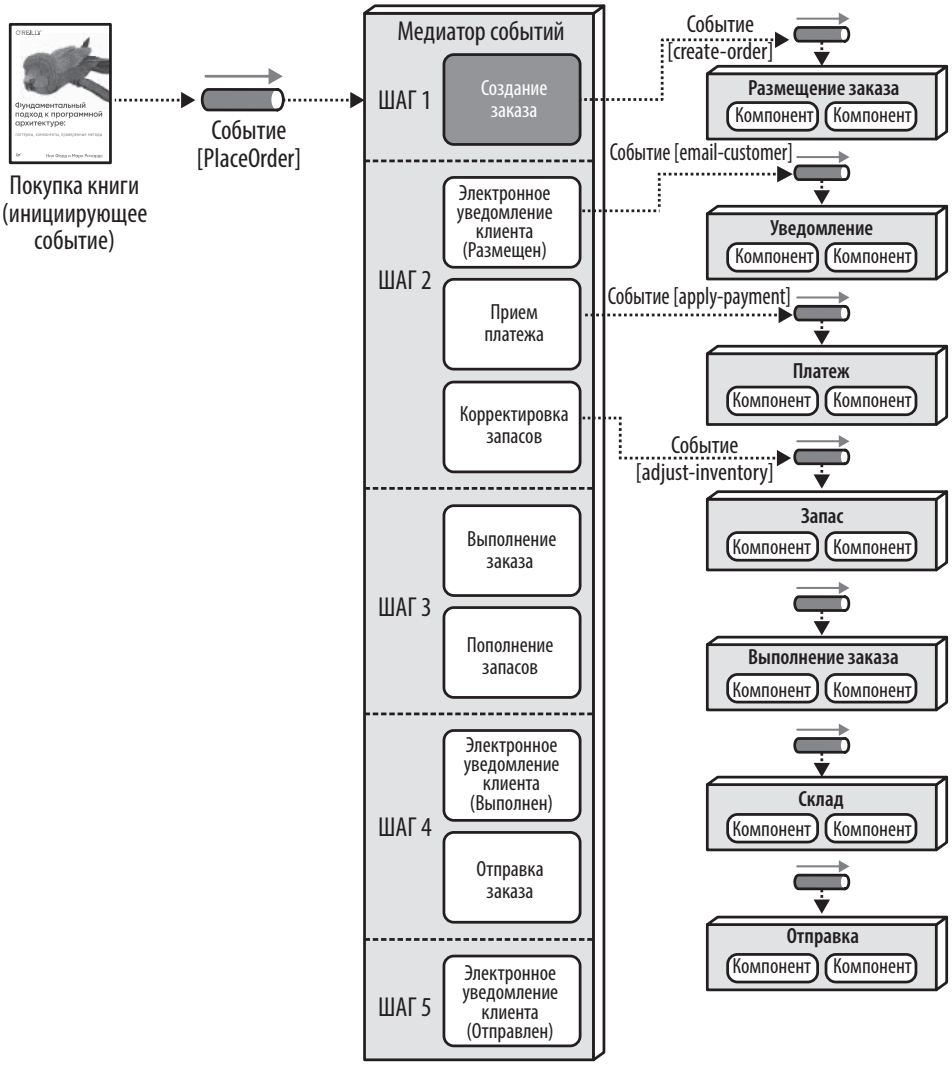


Рис. 14.9. Медиатор, шаг 2

Как только медиатор получит подтверждение об успешном выполнении задач от всех обработчиков событий из шага 2, он сможет перейти к шагу 3 для вы-

полнения заказа (рис. 14.10). Обратите внимание еще раз, что оба этих события (`fulfill-order` и `order-stock`) могут происходить одновременно. Данные события принимаются обработчиками событий `OrderFulfillment` и `Warehouse`, которые выполняют свою работу и возвращают медиатору подтверждение.

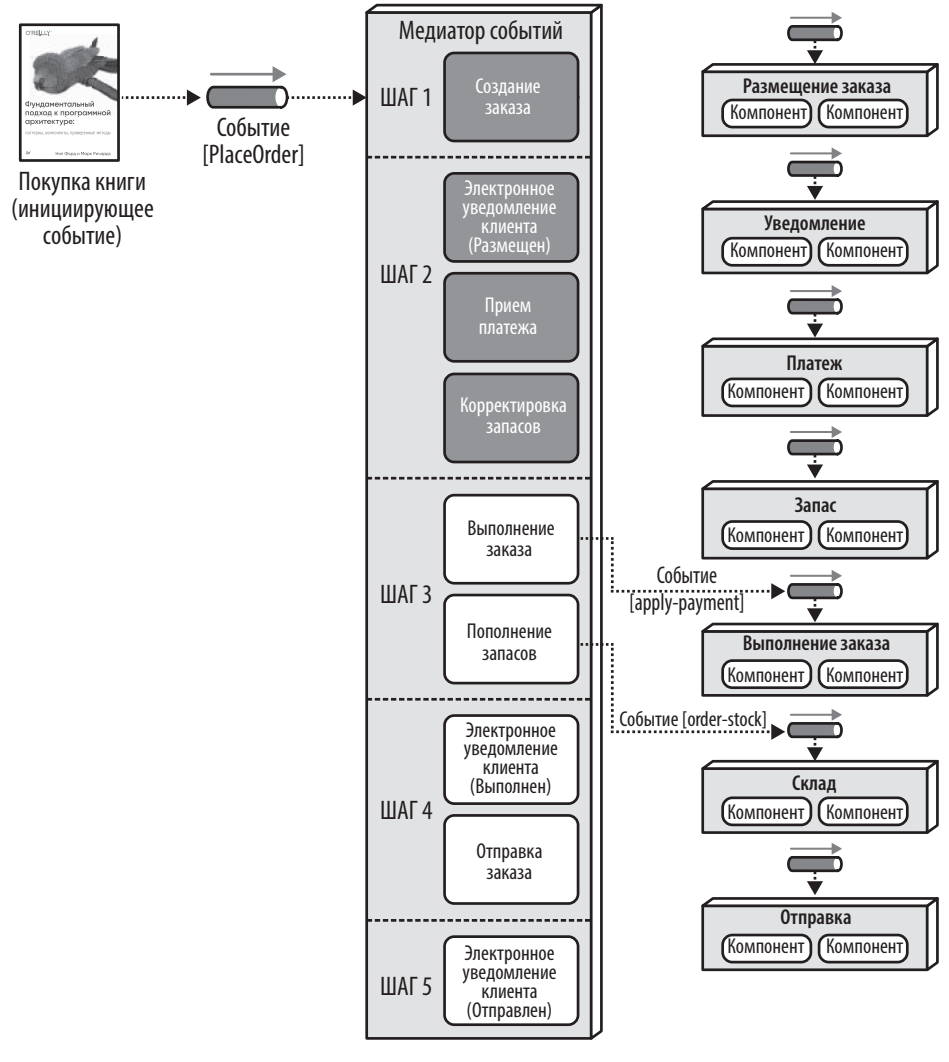


Рис. 14.10. Медиатор, шаг 3

Как только обработка этих событий завершится, медиатор перейдет к отправке заказа на шаге 4 (рис. 14.11). На этом шаге будет сгенерировано еще одно со-

бытие обработки, `email-customer`, с конкретной информацией о том, что нужно сделать (в данном случае уведомить клиента, что заказ готов к отправке), а также событие `ship-order`.

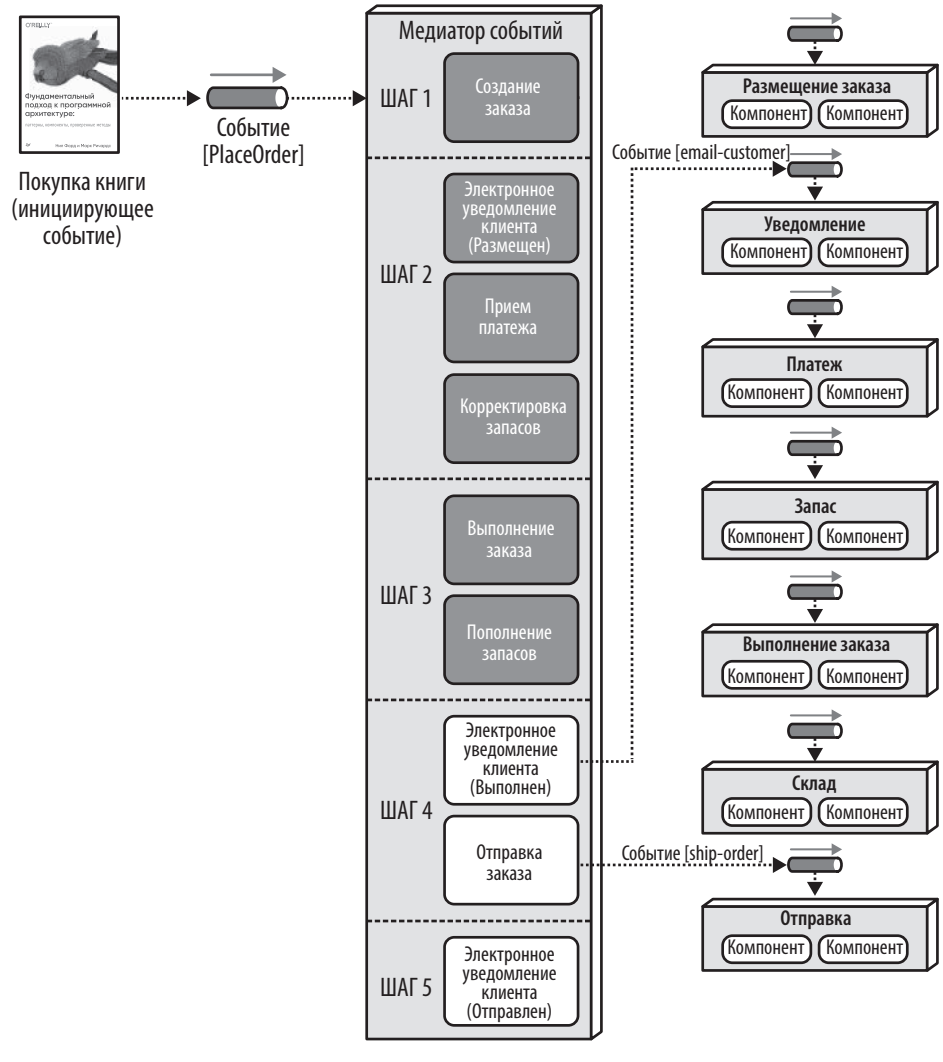


Рис. 14.11. Модератор, шаг 4

И наконец, медиатор перейдет к шагу 5 (рис. 14.12) и сгенерирует еще одно контекстное событие `email_customer` для уведомления клиента, что заказ отправлен. Теперь рабочий процесс завершится, и медиатор пометит процесс обработки

инициирующего события завершенным и удалит все состояния, связанные с инициирующим событием.

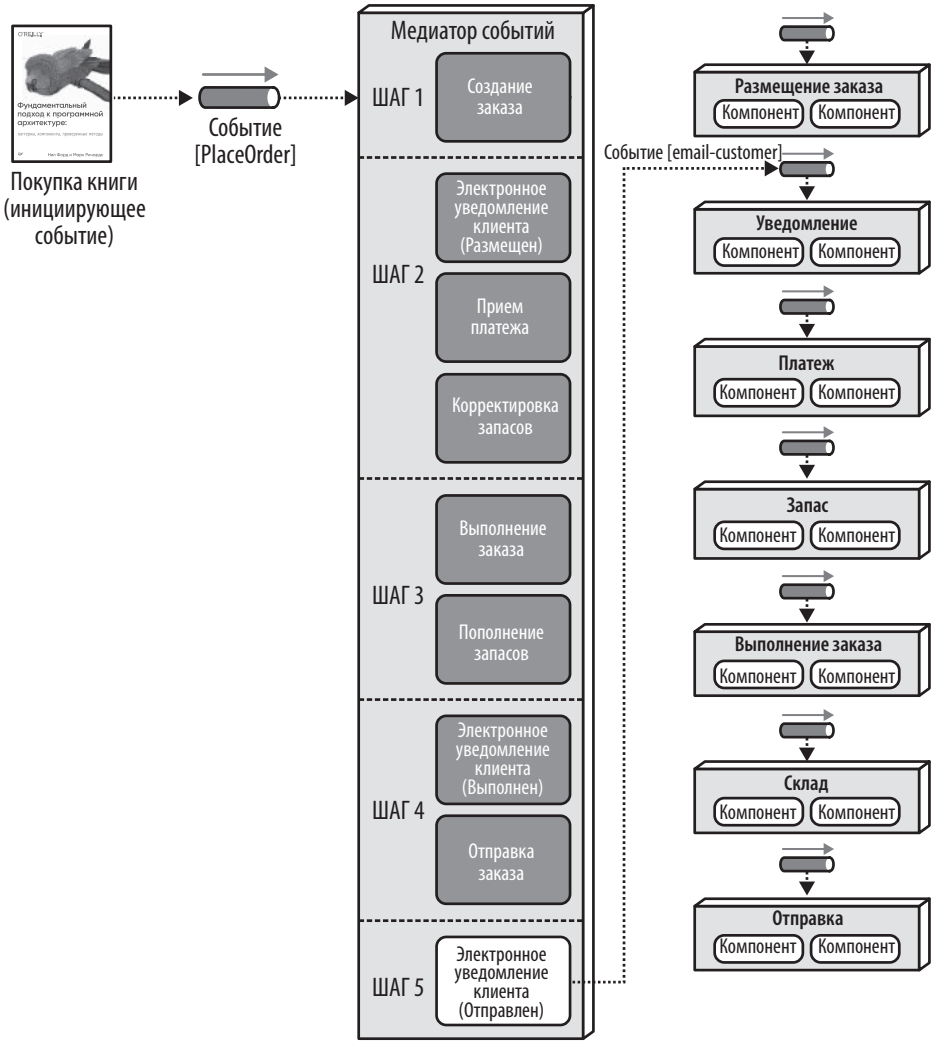


Рис. 14.12. Медиатор, шаг 5

Компонент-медиатор управляет рабочим процессом, чего нет в топологии брокера. Контролируя рабочий процесс, медиатор может поддерживать состояние события и управлять обработкой ошибок, выполнять восстановление и перезапуск. Представим, что в предыдущем примере платеж не прошел из-за просроченной

кредитной карты. В этом случае медиатор получает условие этой ошибки и понимает, что заказ не может быть выполнен (шаг 3), пока платеж не будет принят. Он останавливает процесс и записывает состояние запроса в свое собственное постоянное хранилище данных. Как только платеж пройдет, рабочий процесс может быть перезапущен с места остановки (в данном случае с начала шага 3).

Еще одно характерное различие топологий брокера и медиатора — то, как события обработки отличаются с точки зрения значения и использования. В топологии брокера события обработки публикуются как события, произошедшие в системе (например, `order-created`, `payment-applied` и `email-sent`). Обработчики событий выполняют определенные действия, а другие обработчики событий на это реагируют. А в топологии медиатора события обработки (например, `place-order`, `send-email` и `fulfill-order`) являются *командами* (то есть тем, что должно произойти) в отличие от *событий* (то есть того, что уже произошло). К тому же в топологии медиатора событие обработки должно быть обязательно обработано (команда), а в топологии брокера может быть проигнорировано (реакция).

Топология медиатора, конечно же, решает ряд проблем топологии брокера, но у нее имеются и свои негативные стороны. Прежде всего, очень трудно смоделировать динамическую обработку, происходящую в сложном потоке событий. В результате многие рабочие процессы в медиаторе выполняют только общую обработку, а гибридная модель, объединяющая топологии медиатора и брокера, используется для динамической обработки сложных событий (например, состояния отсутствия товара на складе или других нетипичных ошибок). Кроме того, хотя обработчики событий могут легко расширяться так же, как и в топологии брокера, медиатор должен тоже масштабироваться, что иногда приводит к образованию узких мест в общем потоке обработки событий. И наконец, обработчики событий в топологии медиатора разделены не так сильно, как в топологии брокера, а из-за контроля со стороны медиатора не достигается высокая производительность. Все достоинства и недостатки топологии медиатора сведены в табл. 14.2.

Таблица 14.2. Достоинства и недостатки топологии медиатора

Достоинства	Недостатки
Контроль над рабочим процессом	Более сильная связанность обработчиков событий
Возможность обработки ошибок	Низкая масштабируемость
Восстанавливаемость	Низкая производительность
Возможность перезапуска	Низкая отказоустойчивость
Более простая организация согласованности данных	Проблема моделирования сложных рабочих процессов

Выбор между топологиями брокера и медиатора сводится, по сути, к компромиссу между управлением рабочим процессом и возможностью обработки ошибок, с одной стороны, и высокой производительностью с масштабируемостью — с другой. Хотя в топологии медиатора показатели производительности и масштабируемости вполне приемлемы, они все же ниже, чем в топологии брокера.

Возможности работы в асинхронном режиме

В отличие от других архитектурных стилей, стиль архитектур, управляемых событиями, имеет уникальное свойство, заключающееся в исключительно асинхронном обмене данными, как для обработки по принципу «выстрелил и забыл» (при которой ответ не требуется), так и для обработки по принципу запрос — ответ (при которой требуется ответ от потребителя события). Асинхронный обмен данными может быть весьма эффективным приемом сокращения времени отклика системы.

Рассмотрим пример, показанный на рис. 14.13, где пользователь публикует комментарий на веб-сайте, оставляя отзыв на конкретный товар. Предположим, что сервису комментариев в этом примере требуется на публикацию 3000 мс, поскольку текст комментария проходит через несколько механизмов синтаксического анализа: контроль бранных слов, контроль грамматики, позволяющий убедиться, что по структуре текст не похож на оскорбление, и наконец, контроль

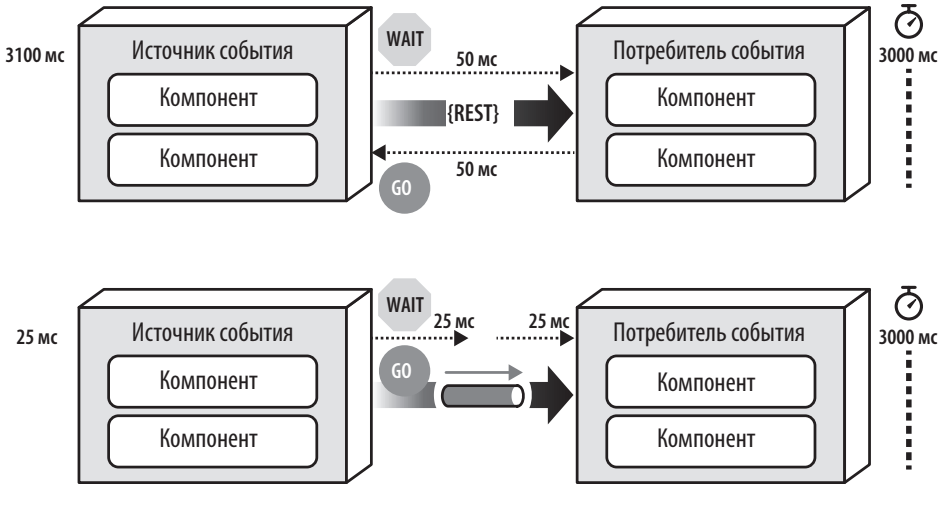


Рис. 14.13. Сравнение синхронного и асинхронного обмена данными (WAIT — ожидание, GO — пуск)

контекста, позволяющий понять, что это отзыв на конкретный товар, а не болтовня про политику. Заметьте, что на рис. 14.13 на верхнем пути публикации комментария используется синхронный RESTful-вызов: сервис получает комментарий с задержкой 50 мс, 3000 мс уходит на публикацию комментария, и 50 мс составляет сетевая задержка на подтверждение пользователю, что комментарий опубликован. Получается, что время отклика при публикации комментария составляет для пользователя 3100 мс. Теперь посмотрим на нижний маршрут и заметим, что при использовании асинхронного обмена сообщениями время отклика на публикацию комментария на веб-сайте для конечного пользователя составляет всего 25 мс (по сравнению с 3100 мс). Фактически на публикацию комментария все же уходит 3025 мс (25 мс на получение сообщения и 3000 мс на саму публикацию), но для пользователя публикация уже состоялась.

Перед нами наглядный пример разницы между *отзывчивостью* и *производительностью*. Если пользователю не нужен ответ (кроме подтверждения или стандартной благодарности), то зачем заставлять его ждать? Отзывчивость заключается в уведомлении пользователя о том, что действие принято и будет тут же обработано, а производительность отвечает за скорость всего процесса. Заметьте, что ничего не было сделано для оптимизации способа обработки текста сервисом работы с комментариями: в обоих случаях на это уходили те же 3000 мс. Повышение *производительности* было бы связано с оптимизацией сервиса работы с комментариями и параллельным запуском всех механизмов текстового и грамматического разбора с использованием кэширования или других подобных приемов. В нижнем примере, показанном на рис. 14.13, решается задача повышения общей отзывчивости системы, а не ее производительности.

Разница между временами отклика двух примеров на рис. 14.13 (3100 мс и 25 мс) просто поразительна. Но есть один нюанс. На синхронном маршруте верхней схемы пользователь получает гарантию публикации комментария. А на нижнем маршруте существует всего лишь подтверждение о приеме поста к публикации, с обещанием, что в конечном итоге пост будет опубликован. С точки зрения конечного пользователя, комментарий принят. А что, если в сообщении будет что-либо неподобающее? Тогда пост будет отклонен, но способа его возврата конечному пользователю нет. Или есть? В этом примере, при условии, что пользователь зарегистрирован на веб-сайте (что является непременным условием постинга), ему может быть отправлено сообщение о том, что именно не так с его комментарием, и предложения по его исправлению. Это был простой пример. А если взять более сложный пример с покупкой в асинхронном режиме сразу нескольких акций (при биржевой торговле), когда нет возможности возвращения к пользователю?

Основная проблема асинхронного обмена данными заключается в обработке ошибок. Отзывчивость существенно повышается, а вот исправлять ошибки становится значительно труднее, что ведет к усложнению системы, ориентированной на работу с событиями. В следующем разделе решение этой задачи рассматривается с применением паттерна реактивной архитектуры, который называется *паттерном событий рабочего процесса* (*workflow event pattern*).

Обработка ошибок

Применение паттерна событий рабочего процесса является одним из способов решить проблему обработки ошибок в асинхронном рабочем процессе. Этот паттерн построен по схеме реактивной архитектуры, одновременно решающей вопросы отказоустойчивости и отзывчивости. Иными словами, система может быть отказоустойчивой с точки зрения обработки ошибок, не оказывая при этом негативного влияния на отзывчивость.

Как показано на рис. 14.14, в паттерне событий рабочего процесса используются делегирование, удержание и восстановление с помощью *делегата рабочего процесса* (*workflow delegate*). Источник события в асинхронном режиме передает данные через канал сообщений потребителю события. Если при обработке данных потребитель события обнаруживает ошибку, он тут же направляет ее *обработчику рабочего процесса* (*workflow processor*) и переходит в очереди событий к следующему сообщению. Таким образом, за счет немедленной обработки не оказывается никакого негативного влияния на отзывчивость. Если бы потребитель события занимался поиском ошибки, то не происходило бы чтения следующего по очереди сообщения, что повлияло бы не только на реакцию на это следующее сообщение, но и на обработку всех других сообщений, ожидающих в очереди.

Как только обработчик рабочего процесса получает ошибку, он пытается выяснить, что не так с сообщением. Это может быть статическая, однозначная ошибка, или же понадобится использовать алгоритмы машинного обучения для анализа сообщения и выявления аномалии в данных. В любом случае обработчик рабочего процесса программным образом (без вмешательства человека) вносит изменения в исходные данные в попытке их восстановления, а затем отправляет их обратно в исходную очередь. Потребитель события видит это сообщение как новое и пытается обработать его еще раз в надежде, что на этот раз все пройдет гладко. Конечно, довольно часто обработчик рабочего процесса не в состоянии определить, чем сообщение не устраивает потребителя. В таких случаях он отправляет сообщение в другую очередь, и его получает так называемый дашборд,

то есть приложение, похожее на Microsoft Outlook или Apple Mail. Этот дашборд обычно находится на рабочем столе ответственного специалиста, который изучает сообщение, сам вносит в него исправления, а затем повторно отправляет его в исходную очередь (обычно через переменную заголовка сообщения reply-to).

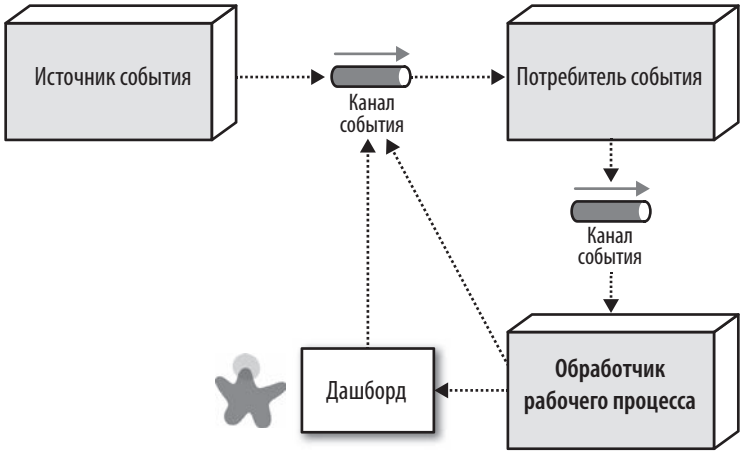


Рис. 14.14. Паттерн событий рабочего процесса, используемый в реактивной архитектуре

Чтобы проиллюстрировать работу паттерна событий рабочего процесса, предположим, что трейдер в одной части страны принимает торговые ордера (инструкции о том, какие акции покупать и в каком количестве) от крупной торговой фирмы, находящейся в другой части страны. Трейдер группирует торговые ордера (создавая то, что обычно называют корзиной) и в асинхронном режиме отправляет их в крупную торговую фирму для размещения у брокера с целью приобретения акций. Чтобы упростить пример, предположим, что контракт для торговых инструкций должен выглядеть следующим образом:

```
ACCOUNT(String),SIDE(String),SYMBOL(String),SHARES(Long)
```

Предположим, что крупная торговая фирма получает от торгового советника следующую корзину торговых ордеров Apple (AAPL):

```
12654A87FR4,BUY,AAPL,1254
87R54E3068U,BUY,AAPL,3122
6R4NB7609JJ,BUY,AAPL,5433
2WE35HF6DHF,BUY,AAPL,8756 SHARES
764980974R2,BUY,AAPL,1211
1533G658HD8,BUY,AAPL,2654
```

Обратите внимание, что в четвертой торговой инструкции (`2WE35HF6DHF,BUY, AAPL,8756 SHARES`) после количества торгуемых акций имеется слово `SHARES`. Когда эти торговые ордера, получаемые в асинхронном режиме, обрабатываются крупной торговой фирмой, не имеющей каких-либо возможностей обработки ошибок, сервисом размещения сделок выдается следующая ошибка:

```
Exception in thread “main” java.lang.NumberFormatException:
        For input string: “8756 SHARES”
        at java.lang.NumberFormatException.forInputString
        (NumberFormatException.java:65)
        at java.lang.Long.parseLong(Long.java:589)
        at java.lang.Long.<init>(Long.java:965)
        at trading.TradePlacement.execute(TradePlacement.java:23)
        at trading.TradePlacement.main(TradePlacement.java:29)
```

Когда возникает это исключение, сервис размещения сделок ничего не может поделать, поскольку запрос отправлен в асинхронном режиме, — кроме разве что регистрации состояния ошибки. Другими словами, у него нет такого пользователя, который оперативно бы реагировал на ошибку и занимался ее исправлением.

Применение паттерна событий рабочего процесса позволяет исправить такую ошибку программным путем. Поскольку у крупной торговой фирмы нет контроля над трейдером и над соответствующими данными отправляемых им торговых ордеров, ей необходимо самостоятельно исправлять ошибки (рис. 14.15). Когда возникает подобная ошибка (`2WE35HF6DHF,BUY,AAPL,8756 SHARES`), сервис размещения сделок `Trade Placement` немедленно передает ее через асинхронный обмен сообщениями в сервис `Trade Placement Error` для обработки, дополнительно передавая информацию об исключении:

```
Trade Placed: 12654A87FR4,BUY,AAPL,1254
Trade Placed: 87R54E3068U,BUY,AAPL,3122
Trade Placed: 6R4NB7609JJ,BUY,AAPL,5433
Error Placing Trade: “2WE35HF6DHF,BUY,AAPL,8756 SHARES”
Sending to trade error processor <-- delegate the error fixing and move on
Trade Placed: 764980974R2,BUY,AAPL,1211
...
```

Сервис `Trade Placement Error` (действуя в качестве делегата рабочего процесса) получает ошибку и исследует исключение. Обнаружив, что проблема связана со словом `SHARES` в поле для количества акций, сервис `Trade Placement Error` удаляет это слово и снова отправляет ордер на сделку на повторную обработку:

```
Received Trade Order Error: 2WE35HF6DHF,BUY,AAPL,8756 SHARES
Trade fixed: 2WE35HF6DHF,BUY,AAPL,8756
Resubmitting Trade For Re-Processing
```

Затем исправленный ордер на сделку успешно проходит обработку в сервисе размещения сделок:

```
...
trade placed: 1533G658HD8,BUY,AAPL,2654
trade placed: 2WE35HF6DHF,BUY,AAPL,8756 <-- this was the original trade in error
```

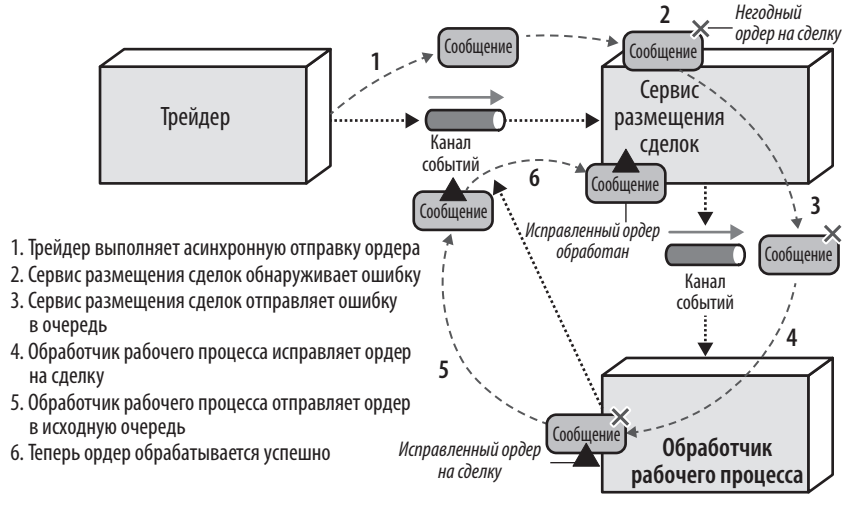


Рис. 14.15. Обработка ошибок с помощью паттерна событий рабочего процесса

Особенностью паттерна событий рабочего процесса является внеочередная обработка повторно отправленных сообщений, содержавших ошибку. В нашем примере биржевой торговли порядок отправки сообщений играет важную роль, поскольку все сделки в рамках конкретной учетной записи должны обрабатываться в определенном порядке (например, на одном и том же брокерском счете продажа (SELL) IBM должна происходить до покупки (BUY) AAPL). Поддерживать порядок следования сообщений в заданном контексте (в данном случае по номеру счета брокера) возможно, но это довольно сложная задача. Решением может стать организация очередности в сервисе размещения сделок `Trade Placement` и сохранение номера счета сделки в сообщении об ошибке. Любая сделка с точно таким же номером счета будет сохраняться для последующей обработки во временной очереди (в порядке FIFO — «первым пришел — первым вышел»). После того как изначально ошибочная сделка исправлена и обработана, сервис размещения сделок извлекает из очереди оставшиеся сделки для той же учетной записи и обрабатывает их по порядку.

Предотвращение потери данных

Главной проблемой асинхронного обмена является потеря данных. К сожалению, в архитектуре, управляемой событиями, имеется множество мест, где такое может произойти. Под потерей данных подразумевается сообщение, которое теряется или никогда не достигает конечной цели. К счастью, есть действенные готовые приемы, которыми можно воспользоваться для предотвращения потери данных при асинхронном обмене сообщениями.

Для того чтобы лучше понять проблемы, связанные с потерей данных в архитектуре, управляемой событиями, предположим, что обработчик событий `Event Processor A` асинхронно отправляет в очередь сообщение. Обработчик событий `Event Processor B` принимает сообщение и вставляет данные из него в базу данных. В этом типовом сценарии есть три места, где могут быть утеряны данные (рис. 14.16):

1. Сообщение вообще не попадает в очередь от обработчика событий `Event Processor A`; или даже если оно попадает, брокер выходит из строя еще до того, как следующий обработчик событий сможет обработать сообщение.
2. Обработчик событий `Event Processor B` извлекает из очереди следующее доступное сообщение и выходит из строя, не успевая его обработать.
3. Обработчик событий `Event Processor B` не может сохранить сообщение в базе данных из-за ошибки в самих данных.

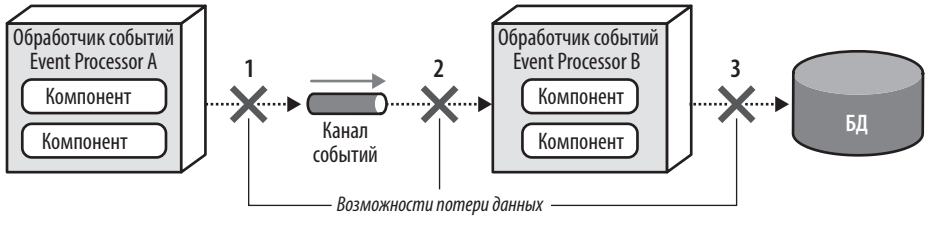


Рис. 14.16. Места, где данные могут быть потеряны в архитектуре, ориентированной на события

Каждую из этих областей потери данных можно сузить за счет основных приемов обмена сообщениями. Проблема 1 (сообщение вообще не попадает в очередь) довольно просто решается путем использования сохраняемых очередей сообщений вместе с так называемой *синхронной отправкой* (*synchronous send*). Сохраняемые очереди сообщений поддерживают так называемую гарантированную доставку. Когда брокер сообщений получает сообщение, он не только хранит его в памяти для быстрого извлечения, но и сохраняет в каком-нибудь физическом

хранилище данных (например, в файловой системе или в базе данных). Если брокер сообщений выходит из строя, сообщение физически сохраняется на диске, поэтому когда этот брокер снова заработает, сообщение будет доступно для обработки. Синхронная отправка приводит к блокирующему ожиданию в источнике сообщения до тех пор, пока брокер не подтвердит сохранение сообщения. Эти два основных метода не позволяют потерять сообщение на пути от источника события до очереди, поскольку оно либо все еще в источнике, либо уже сохранено в очереди.

Проблема 2 (обработчик событий `Event Processor B` извлекает из очереди следующее доступное сообщение и выходит из строя еще до обработки события) может быть также решена с помощью основного приема обмена сообщениями, который называется *режимом уведомления клиента* (*client acknowledge mode*). По умолчанию, когда сообщение извлекается из очереди, оно тут же удаляется из нее; это называется *режимом автоматического уведомления (auto acknowledge mode*). В режиме уведомления клиента сообщение сохраняется в очереди, и к нему прикрепляется идентификатор клиента, поэтому оно не может быть прочитано другими потребителями. Если в этом режиме обработчик событий `Event Processor B` выйдет из строя, сообщение будет все еще храниться в очереди и не будет потеряно на данном этапе работы с сообщениями.

Проблема 3 (обработчик событий `Event Processor B` не может сохранить данные в базе данных из-за имеющейся в них ошибки) решается путем использования ACID-транзакций (отвечающих принципам атомарности, достоверности, изолированности и живучести) для их коммита в базе данных. Как только произойдет коммит транзакции в базе данных, это будет означать, что данные гарантированно сохранены. При использовании так называемой *поддержки последнего участника* (*last participant support, LPS*) сообщение удаляется из сохраняемой очереди путем уведомления о завершении обработки и о сохранении сообщения. Тем самым гарантируется, что сообщение не потеряно на всем пути его перемещения от обработчика событий `Event Processor A` в базу данных. Эта технология показана на рис. 14.17.

Возможность широковещательной передачи

Еще одним уникальным свойством архитектуры, управляемой событиями, является возможность широковещательной рассылки о событиях без какой-либо информации, где (если вообще где-то) будет получено сообщение и что с ним будет сделано. Эта технология, проиллюстрированная на рис. 14.18, показывает, что когда отправитель опубликует сообщение, его получают сразу несколько подписчиков.

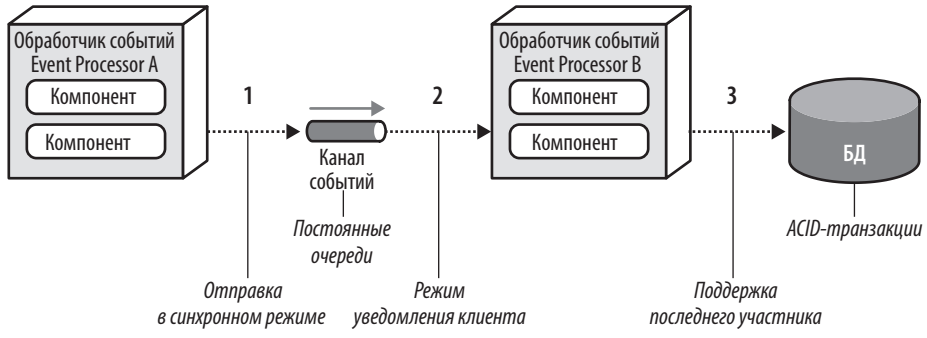


Рис. 14.17. Предотвращение потери данных в приложениях с архитектурой, управляемой событиями

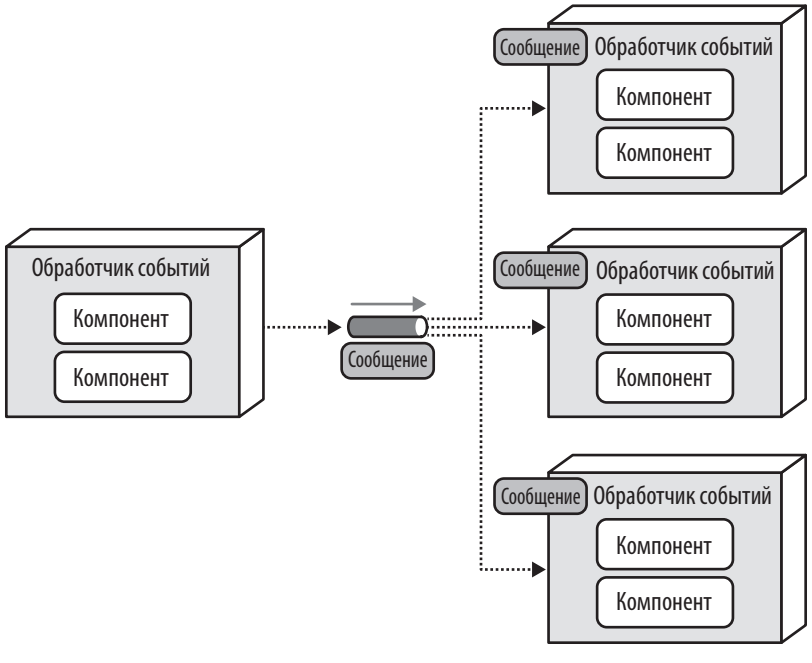


Рис. 14.18. Рассылка событий другим обработчикам событий

Широковещательная рассылка — это, наверное, самый высокий уровень разделения между обработчиками событий, поскольку производитель распространяемого сообщения, как правило, не знает, какие именно обработчики событий его получат, а еще важнее, что именно они с ним сделают. Возможности широковещательной рассылки являются важной частью паттернов для

окончательной согласованности, обработки комплексных событий (complex event processing, CEP) и разрешения множества других ситуаций. Рассмотрим частые изменения цен на акции для инструментов фондового рынка. Каждый тикер (текущая цена конкретной акции) может повлиять на множество вещей. Но сервис, публикующий последнюю цену, просто оповещает о ней, не зная, как этой информацией воспользуются.

Запрос — ответ

До сих пор в этой главе шла речь об асинхронных запросах, которым не нужен немедленный ответ от потребителя события. А что, если при заказе книги понадобится идентификатор заказа? Или при бронировании места на авиарейс нужен номер подтверждения? Здесь мы сталкиваемся с примерами взаимодействия сервисов или обработчиков событий, для которого требуется синхронный обмен данными.

В архитектуре, управляемой событиями, синхронный обмен данными выполняется через обмен сообщениями по принципу *запрос — ответ* (*request — reply*). Иногда его называют *псевдосинхронным обменом данными* (*pseudosynchronous communications*). Каждый канал событий в среде обмена сообщениями по принципу запрос — ответ состоит из двух очередей: очереди запросов и очереди ответов. Исходный запрос на получение информации отправляется в асинхронном режиме в очередь запросов, после чего управление возвращается источнику сообщения. Затем источник блокирует очередь ответов в ожидании получения ответа. Потребитель сообщения получает сообщение, обрабатывает его, а затем отправляет ответ в очередь ответов. Далее источник события получает сообщение с данными ответа. Основной ход процесса показан на рис. 14.19.

Для реализации обмена сообщениями по принципу запрос — ответ применяются две основные технологии. Первая (и более распространенная) заключается в использовании *связующего идентификатора* (*correlation ID*), содержащегося в заголовке сообщения. Он представляет собой поле в ответном сообщении, значением которого является идентификатор исходного сообщения запроса. Эта технология (рис. 14.20) с идентификатором сообщения, помеченным как ID, и связующим идентификатором, помеченным как CID, работает следующим образом:

1. Источник события отправляет сообщение в очередь запросов и записывает уникальный идентификатор сообщения (в данном случае ID 124). Заметьте, что связующий идентификатор (CID) в данном случае имеет значение `null`.

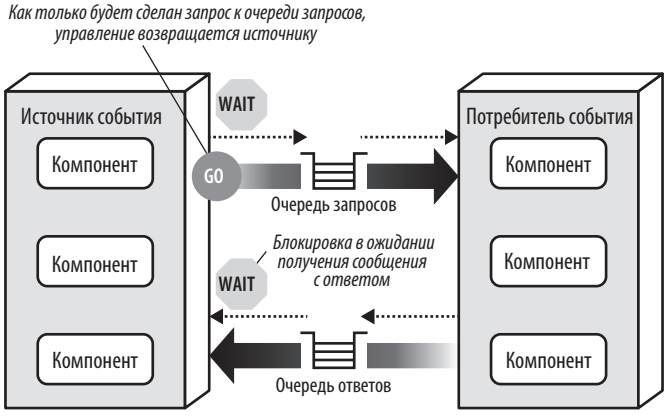


Рис. 14.19. Обработка сообщений по принципу запрос — ответ

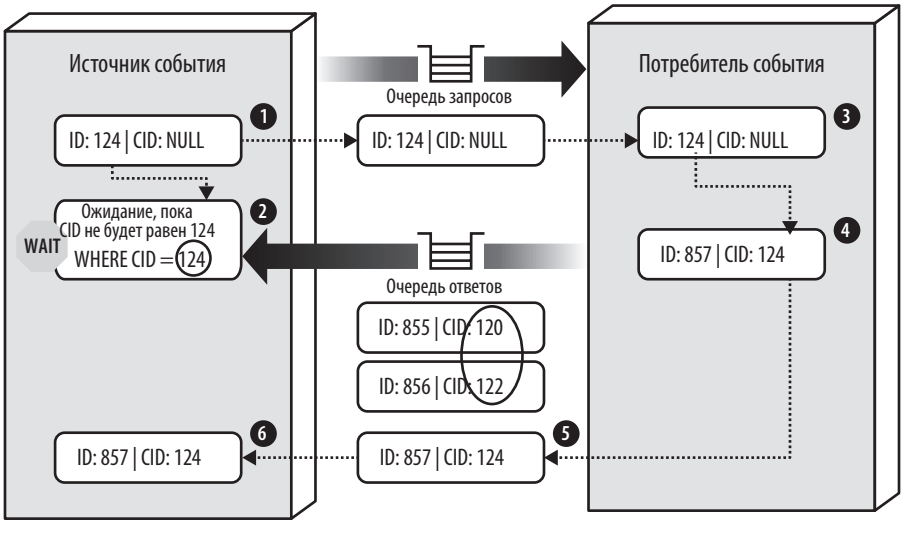


Рис. 14.20. Обработка сообщений по принципу запрос — ответ с использованием связующего идентификатора

2. Теперь источник сообщения блокируется в ожидании в очереди ответов фильтром сообщений (также называемым селектором сообщений), где связующий идентификатор в заголовке сообщения равен идентификатору исходного сообщения (в данном случае 124). Заметьте, что в очереди ответов имеются два сообщения: сообщение с идентификатором 855 со связующим идентификатором 120 и сообщение с идентификатором 856 со связующим идентификатором 122. Ни одно из них не будет отобрано, поскольку связу-

ющий идентификатор не соответствует тому, что ищет потребитель сообщения (CID 124).

3. Потребитель сообщения получает сообщение (ID 124) и обрабатывает запрос.
4. Потребитель сообщения создает ответное сообщение, содержащее данные ответа, и устанавливает для связующего идентификатора в его заголовке значение идентификатора исходного сообщения (124).
5. Потребитель события отправляет в очередь ответов новое сообщение (ID 857).
6. Источник события получает сообщение, поскольку связующий идентификатор (124) соответствует установке селектора сообщений из шага 2.

Еще одна технология, используемая для реализации обмена сообщениями по принципу запрос — ответ, заключается в создании *временной очереди* (*temporary queue*) ответов. Временная очередь предназначена для конкретного запроса, она создается, когда запрос сделан, и удаляется, когда запрос завершен. Эта технология, показанная на рис. 14.21, не требует связующего идентификатора, поскольку временная очередь является специально выделенной и известной только источнику события для конкретного запроса. Технология временной очереди работает следующим образом:

1. Источник события создает временную очередь (или же она создается автоматически, в зависимости от брокера сообщений) и отправляет сообщение в очередь запросов, передавая имя временной очереди в заголовке с адресом для ответов reply-to (или в каком-либо другом согласованном пользовательском атрибуте заголовка сообщения).
2. Источник события блокируется в ожидании на временной очереди ответов. Селектор сообщений не нужен, поскольку любое сообщение, отправленное в эту очередь, принадлежит исключительно источнику события, изначально отправившему сообщение.
3. Потребитель события получает сообщение, обрабатывает запрос и отправляет сообщение с ответом в очередь ответов, указанную в заголовке reply-to.
4. Обработчик событий получает сообщение и удаляет временную очередь.

Хотя технология временной очереди значительно проще, при ее применении брокер сообщений должен создавать временную очередь для каждого сделанного запроса, а затем без промедления ее удалять. Большое количество сообщений может существенно замедлить работу брокера и оказать негативное влияние на общую производительность и отзывчивость. Поэтому мы обычно рекомендуем использовать технологию связующего идентификатора.

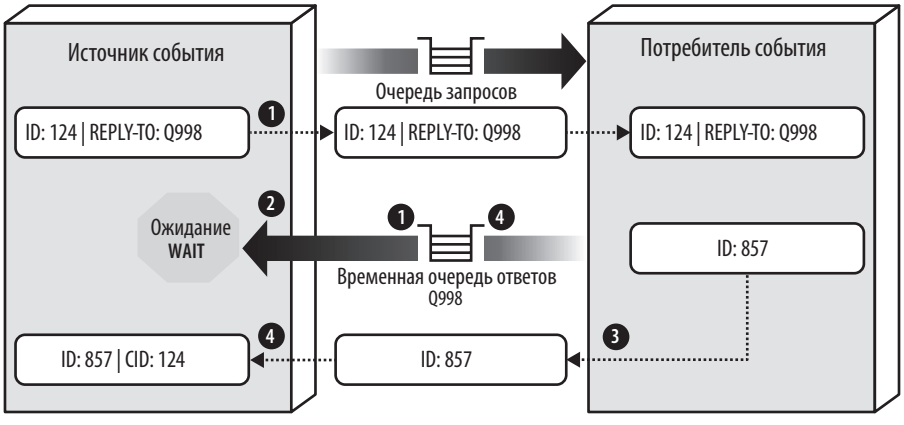


Рис. 14.21. Обработка сообщений по принципу запрос — ответ с использованием временной очереди

Выбор между моделью на основе запросов и моделью на основе событий

Модель на основе запросов и модель на основе событий являются вполне жизнеспособными подходами к разработке программных систем. Но для общего успеха системы важно выбрать правильную модель. Для четко структурированных запросов на основе данных (например, при извлечении данных профиля клиента), когда нужны определенность и контроль над рабочим процессом, мы рекомендуем останавливать выбор на модели на основе запросов. А для гибких, основанных на действиях событий, требующих высоких уровней отзывчивости и масштабируемости, со сложной и динамичной пользовательской обработкой, мы рекомендуем выбирать модель на основе событий. Анализ всех компромиссов, связанных с применением модели на основе событий, также поможет с выбором. Все достоинства и недостатки модели на основе событий, применяемой в архитектуре, управляемой событиями, сведены в табл. 14.3.

Таблица 14.3. Компромиссы, присущие модели на основе событий

Преимущества перед моделью на основе запросов	Компромиссы
Более быстрая реакция на динамический пользовательский контент	Поддержка только окончательной согласованности
Лучшая масштабируемость и приспособляемость	Менее жесткий контроль над процессом обработки

Преимущества перед моделью на основе запросов	Компромиссы
Лучшая гибкость и управление изменениями	Меньшая определенность в отношении исхода потока событий
Лучшая адаптируемость и расширяемость	Трудности тестирования и отладки
Лучшая отзывчивость и производительность	
Лучшая приспособленность к принятию решений в реальном времени	
Лучшая реакция на ситуационную осведомленность	

Гибридные архитектуры, управляемые событиями

Хотя многие приложения используют архитектуру, управляемую событиями, в качестве основной, зачастую она применяется в сочетании с другими архитектурными стилями, формируя так называемую гибридную архитектуру. К некоторым широко распространенным архитектурным стилям, использующим архитектуру, управляемую событиями, как часть другого архитектурного стиля, относятся микросервисы и архитектура на основе пространства. К другим возможным гибридам относится архитектура микроядра, управляемая событиями, и архитектура конвейера, управляемая событиями.

Добавление архитектуры, управляемой событиями, к любому архитектурному стилю помогает избавиться от узких мест, обеспечить противодавление при отклонении запросов и предоставить уровень отзывчивости на действия пользователя, недоступный в других архитектурных стилях. Как в микросервисах, так и в архитектуре на основе пространства для переноса данных используется обмен сообщениями, при котором выполняется асинхронная отправка данных к другому обработчику, который, в свою очередь, обновляет информацию в базе данных. Оба стиля также используют архитектуру, управляемую событиями, для предоставления уровня программного масштабирования сервисов (в архитектуре микросервисов) и блоков обработки (в архитектуре на основе пространства), применяя при этом обмен сообщениями для связи между сервисами.

Оценки архитектурных свойств

Одна звезда в оценочной таблице свойств (рис. 14.22) означает, что данное архитектурное свойство не имеет высокой поддержки в архитектуре, а пять звезд — что архитектурное свойство является одной из самых сильных сторон

архитектурного стиля. Определение каждого свойства, указанного в оценочной таблице, можно найти в главе 4.

Архитектурное свойство	Оценка в звездах
Тип разбиения	Технический
Количество квантов	От одного до нескольких
Развертываемость	★★★
Адаптируемость	★★★
Эволюционность	★★★★★
Отказоустойчивость	★★★★★
Модульность	★★★★
Общие затраты	★★★
Производительность	★★★★★
Надежность	★★★
Масштабируемость	★★★★★
Простота	★
Тестируемость	★★

Рис. 14.22. Оценки свойств архитектуры, управляемой событиями

Прежде всего архитектура, управляемая событиями, имеет техническую природу разбиения, где любая конкретная предметная область распределена между несколькими обработчиками событий и связана воедино посредством медиаторов, очередей и топиков. Изменения в конкретной предметной области обычно оказывают влияние на многие обработчики событий, медиаторы и другие артефакты обмена сообщениями, поэтому архитектура, управляемая событиями, не относится к предметному типу разбиения.

В архитектуре, управляемой событиями, может быть от одного до нескольких квантов, что обычно зависит от взаимодействия с базой данных в каждом об-

работчике событий и от обработки запроса — ответа. Несмотря на то что весь обмен данными в этой архитектуре асинхронный, если несколько обработчиков событий совместно используют один и тот же экземпляр базы данных, все они будут содержаться в одном и том же архитектурном кванте. Та же логика верна и для обработки запроса — ответа: даже при том, что обмен данными между обработчиками событий все еще асинхронный, если запрос требует немедленного ответа от потребителя события, он синхронно связывает обработчики событий вместе, поэтому они принадлежат одному и тому же кванту.

Чтобы проиллюстрировать этот момент, рассмотрим пример, где один обработчик событий отправляет запрос на размещение заказа другому обработчику событий. Первый обработчик событий для продолжения своей работы должен дождаться идентификатора заказа от другого обработчика. Если второй обработчик событий, размещающий заказ и генерирующий идентификатор, даст сбой, первый обработчик не сможет продолжить работу. Поэтому они являются частями одного и того же архитектурного кванта и имеют общие архитектурные свойства, несмотря на то что оба они отправляют и получают сообщения в асинхронном режиме.

Архитектура, управляемая событиями, получает пять звезд за производительность, масштабируемость и отказоустойчивость — основные сильные стороны этого архитектурного стиля. Высокая производительность достигается за счет асинхронного обмена данными в сочетании с высоким уровнем параллельной обработки. Высокая масштабируемость достигается за счет того, что нагрузка обработчиков событий (которые также называются *конкурирующими потребителями*, *competing consumers*) балансируется программно. По мере увеличения количества запросов могут быть программно добавлены дополнительные обработчики событий. Отказоустойчивость достигается за счет развязки обработчиков событий, обменивающихся данными в асинхронном режиме и обеспечивающих окончательную согласованность и конечную обработку событий рабочих процессов. Если отправляющий запрос пользовательский интерфейс или обработчик событий не требует немедленного ответа, то для обработки события на более позднем этапе в условиях недоступности последующих обработчиков могут использоваться промисы и футуры.

Архитектура, управляемая событиями, имеет относительно низкие общие оценки *простоты* и *тестируемости*, в основном из-за встречающихся в этом архитектурном стиле недетерминированных и динамических потоков событий. Протестировать детерминированные потоки в рамках модели, основанной на запросах, сравнительно легко, поскольку маршруты и результаты общеизвестны, но к модели, ориентированной на события, это не относится. Порой неизвестно,

как обработчики событий будут реагировать на динамические события и какие сообщения они при этом могут создавать. Соответствующие «диаграммы дерева событий» могут быть чрезвычайно сложными, с сотнями и даже тысячами сценариев, что существенно затрудняет управление и тестирование.

И наконец, архитектуры, управляемые событиями, относятся к чрезвычайно высокоэволюционирующим, отсюда и пятизвездочная оценка этого показателя. Добавление новых функций через уже существующие или через новые обработчики событий выполняется относительно просто, особенно в топологии брокера. Данные становятся доступными путем предоставления хуков через опубликованные сообщения, поэтому нет необходимости вносить изменения в инфраструктуру или в существующие обработчики событий для добавления какой-либо новой функциональности.

ГЛАВА 15

Архитектура на основе пространства

Большинство коммерческих веб-приложений следуют одному и тому же алгоритму: запрос от браузера попадает на веб-сервер, затем на сервер приложения и, наконец, на сервер базы данных. Эта схема отлично работает при небольшом количестве клиентов, но по мере роста числа пользователей начинают появляться узкие места, сначала на уровне веб-сервера, затем на уровне сервера приложения и, наконец, на уровне сервера базы данных. Классическим решением этой проблемы будет масштабирование веб-сервера. Оно достигается сравнительно просто, недорого и иногда помогает совсем избавиться от узких мест. Но в большинстве случаев при высокой пользовательской нагрузке масштабирование на уровне веб-сервера просто переносит узкое место на сервер приложений. Масштабирование серверов приложений может оказаться более сложной и затратной задачей, чем масштабирование веб-серверов, и обычно просто приводит к перемещению узкого места на уровень сервера базы данных, где масштабирование еще труднее и затратнее. Даже если удастся выполнить масштабирование базы данных, в конечном итоге будет получена топология, похожая на треугольник (рис. 15.1), у которого самая широкая часть — это веб-серверы (их легче всего масштабировать), а самая узкая — база данных (ее сложнее всего масштабировать).

В любом крупном приложении с большой одновременной пользовательской нагрузкой база данных обычно является последним ограничивающим фактором возможного количества одновременно обрабатываемых транзакций. Хотя различные технологии кэширования и инструменты масштабирования базы данных могут помочь, факт остается фактом: масштабирование обычного приложения для экстремальных нагрузок — задача весьма непростая.

Стиль архитектур на *основе пространства* (*space-based architecture*) разработан специально для решения проблем, связанных с высокой масштабируемостью,

адаптируемостью и высоким уровнем параллелизма. Он также может пригодиться для приложений с переменными и непредсказуемыми объемами одновременных пользовательских обращений. Решение проблемы экстремальной и переменной масштабируемости архитектурным путем часто оказывается более удачным подходом, чем попытки масштабирования базы данных или внедрение кэширования в немасштабируемую архитектуру.

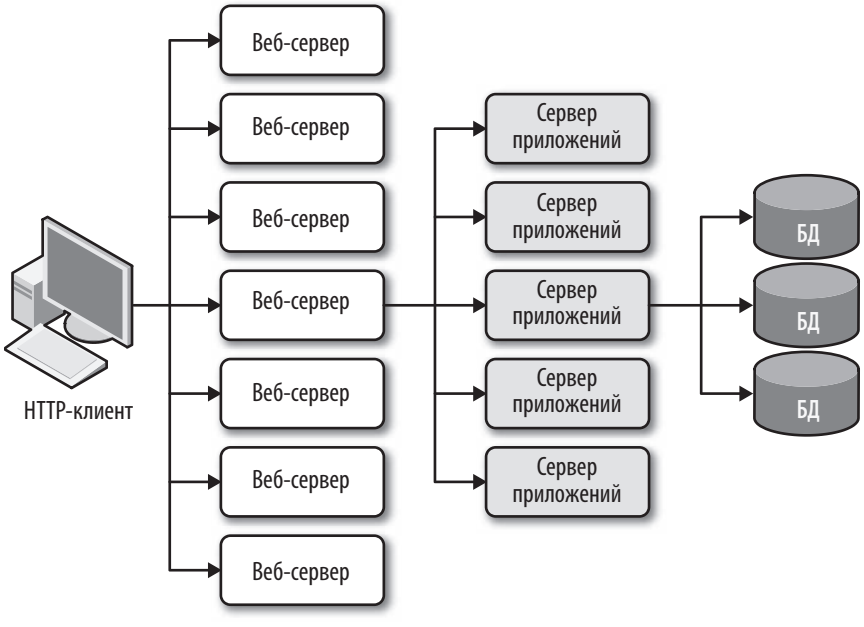


Рис. 15.1. Пределы масштабируемости в традиционной веб-топологии

Топология

Архитектура на основе пространства получила свое название от концепции *пространства кортежей*[1] (*tuple space*), технологии использования нескольких параллельных процессоров, взаимодействующих через общую память. Высокая масштабируемость, адаптируемость и производительность достигаются за счет избавления от центральной базы данных как синхронного ограничителя в системе и использования вместо нее реплицированных сеток данных в памяти (in-memory data grids). Данные приложения содержатся в оперативной памяти

[1] https://oreil.ly/XVJ_D

и реплицируются во всех активных блоках обработки данных. Когда такой блок обновляет данные, он асинхронно отправляет эту информацию в БД, обычно через обмен сообщениями с постоянными очередями. Блоки обработки запускаются и останавливаются динамически по мере увеличения и уменьшения пользовательской нагрузки, решая тем самым проблему переменной масштабируемости. Отсутствие центральной БД, участвующей в стандартной транзакционной обработке приложения, позволяет устранить связанное с ней узкое место, что обеспечивает чуть ли не бесконечную масштабируемость.

Архитектура на основе пространства составляется из нескольких компонентов: *блока обработки* (*processing unit*), содержащего код приложения, *виртуализированного связующего программного обеспечения* (*virtualized middleware*), используемого для управления блоками обработки и координации их действий, *средств переноса данных* (*data pumps*) для асинхронной отправки обновленных данных в базу данных, *средств записи данных* (*data writers*), которые выполняют обновления, полученные от средств переноса данных, и *средств чтения данных* (*data readers*), считывающих информацию из базы данных и доставляющих ее в блоки обработки при их запуске. Все эти компоненты первичной архитектуры показаны на рис. 15.2.

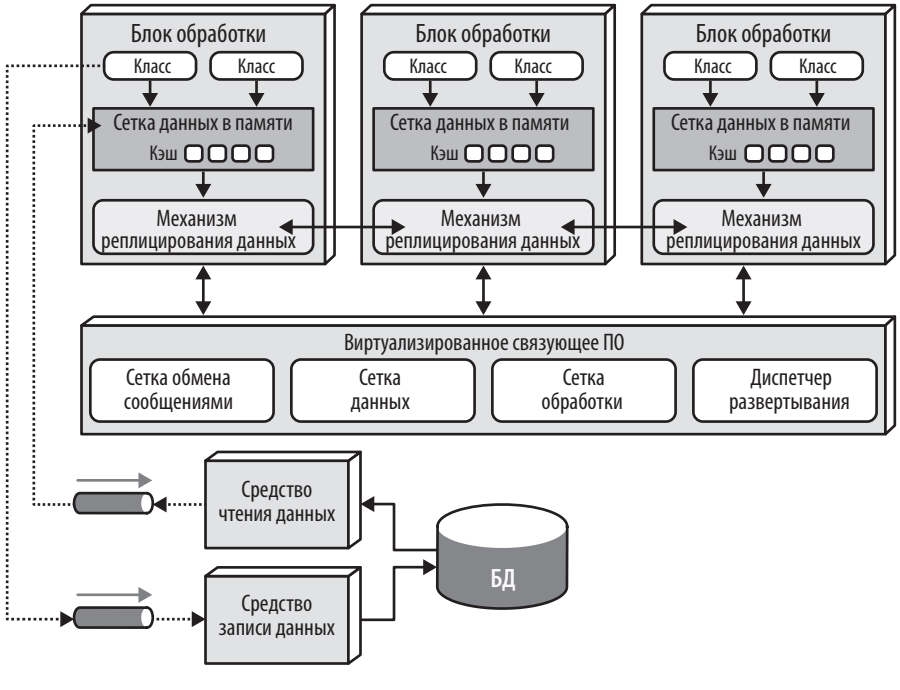


Рис. 15.2. Основная топология архитектуры на основе пространства

Блок обработки

Блок обработки (рис. 15.3) содержит логику приложения (или части такой логики). Обычно он включает в себя веб-компоненты, а также бизнес-логику бэкенда. Содержимое блока обработки зависит от типа приложения. Небольшие веб-приложения, вероятнее всего, будут развернуты в одном блоке обработки, а в более крупных приложениях функциональность может быть разделена на несколько блоков в зависимости от задач приложения. Блок обработки также может содержать небольшие узкоспециализированные сервисы (как в случае с микросервисами). В дополнение к логике приложения блок обработки также содержит сетку данных в памяти и механизм репликации, обычно реализуемый с помощью таких программных средств, как Hazelcast[1], Apache Ignite[2] и Oracle Coherence[3].

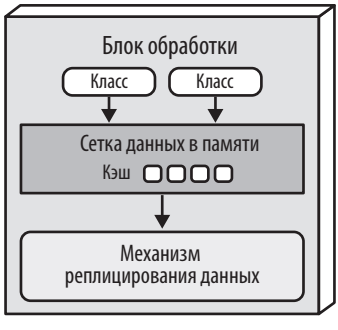


Рис. 15.3. Блок обработки

Виртуализированное связующее программное обеспечение

Виртуализированное связующее программное обеспечение решает в архитектуре проблемы инфраструктуры, связанные с синхронизацией данных и обработкой запросов. В состав виртуализированного связующего ПО входит *сетка обмена сообщениями*, *сетка данных*, *сетка обработки* и *диспетчер развертывания*. Эти компоненты, подробно описанные в следующих разделах, могут быть разработаны или приобретены как сторонние продукты.

[1] https://hazelcast.com/

[2] https://ignite.apache.org/

[3] https://www.oracle.com/java/coherence/

Сетка обмена сообщениями

Сетка обмена сообщениями (messaging grid), показанная на рис. 15.4, управляет входным запросом и состоянием сессии. Когда запрос поступает в виртуализированное связующее ПО, компонент сетки обмена сообщениями определяет, какие активные компоненты обработки доступны для получения запроса, и пересылает запрос одному из этих блоков обработки. Сложность сетки обмена сообщениями может варьироваться от простого циклического (round-robin) алгоритма до более сложного алгоритма поиска ближайшего соединения (next-available algorithm), который отслеживает, каким блоком обработки какой запрос обрабатывается. Этот компонент обычно реализуется с применением обычного веб-сервера с возможностями балансировки нагрузки (например, HA Proxy и Nginx).

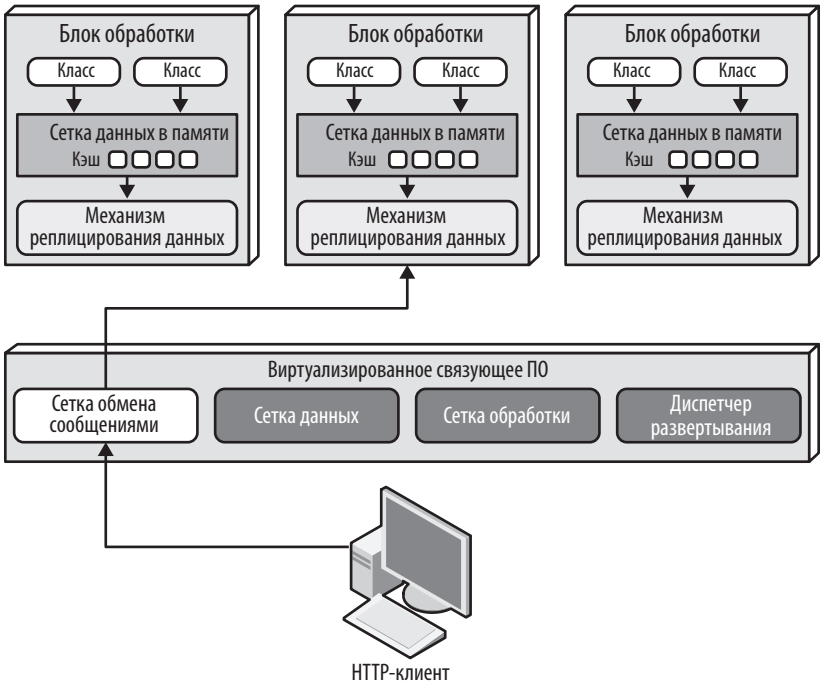


Рис. 15.4. Сетка обмена сообщениями

Сетка данных

Компонент сетки данных (data grid) является, пожалуй, самым важным и решающим в этом архитектурном стиле. В большинстве современных проектов сетка данных создается исключительно внутри блоков обработки в виде реплицированного кэша. Но в реализациях с внешним контроллером или в условиях

использования распределенного кэша эта функциональность будет находиться как в блоках обработки, так и в компоненте сетки данных в виртуализированном связующем ПО. Поскольку сетка обмена сообщениями может пересылать запрос любому из доступных блоков обработки, важно, чтобы каждый блок обработки содержал в памяти идентичные данные. Хотя на рис. 15.5 показана синхронная репликация данных между блоками обработки, в действительности она выполняется асинхронно и очень быстро, и обычно синхронизация данных завершается менее чем за 100 мс.

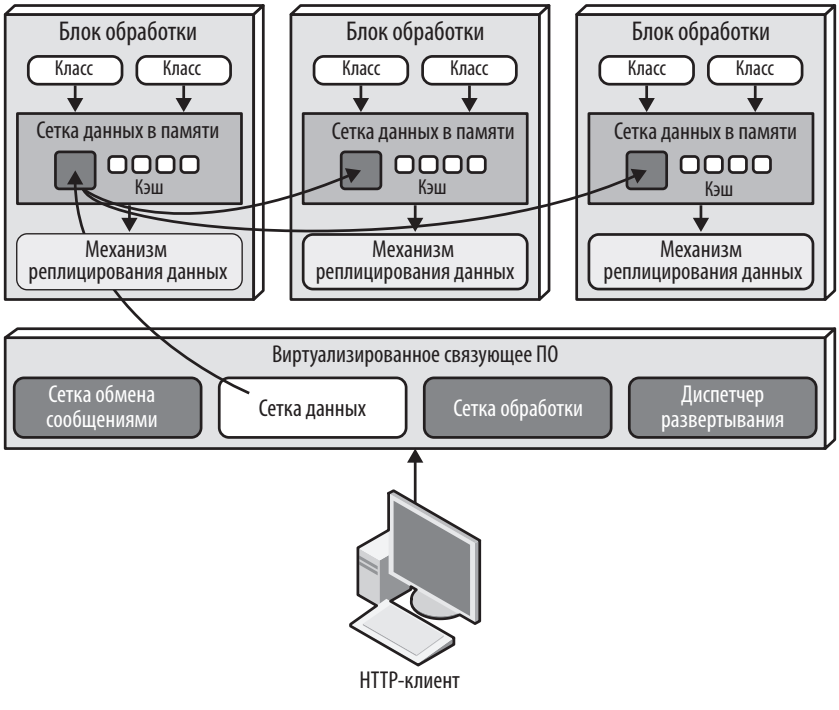


Рис. 15.5. Сетка данных

Данные синхронизируются между блоками обработки, содержащими сетку данных с одним и тем же именем. Для наглядности рассмотрим следующий код на языке Java с использованием Hazelcast, который создает внутреннюю реплицированную сетку данных для блоков обработки, содержащих информацию профиля клиента:

```
HazelcastInstance hz = Hazelcast.newHazelcastInstance();
Map<String, CustomerProfile> profileCache =
        hz.getReplicatedMap("CustomerProfile");
```

Этот код будет содержаться во всех блоках обработки, которым требуется доступ к информации профиля клиента. Изменения, внесенные в кэш с именем `CustomerProfile` из любого блока обработки, будут реплицироваться на все другие блоки обработки, содержащие кэш с таким же именем. Блок обработки может содержать столько реплицированных кэшей, сколько ему потребуется для выполнения своей работы. Возможен и вариант, при котором один блок обработки может сделать удаленный вызов другому блоку обработки, чтобы запросить данные (хореография) или использовать для оркестровки запроса сетку обработки, рассматриваемую в следующем разделе.

Репликация данных в блоках обработки также позволяет экземплярам сервиса запускаться и останавливаться без необходимости считывания данных из БД, если существует хотя бы один экземпляр с именованным реплицированным кэшем. Когда появляется экземпляр блока обработки, он подключается к провайдеру кэша (например, Hazelcast) и делает запрос на получение именованного кэша. После установления соединения с другими блоками обработки кэш будет загружен из любого другого экземпляра блока обработки.

Благодаря использованию *списка участников* (*member list*), каждому блоку обработки известно обо всех других экземплярах блока обработки. Список участников содержит IP-адреса и порты всех других блоков, использующих кэш с таким же именем. Предположим, к примеру, что существует единственный экземпляр обработки, содержащий код и реплицированные кэшированные данные для профиля клиента. В следующих строчках лога, сгенерированных с помощью Hazelcast, показан только один экземпляр, то есть список участников для данного экземпляра содержит только сам этот экземпляр:

```
Instance 1:
Members {size:1, ver:1} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 this
]
```

Когда с кэшем под тем же именем запускается еще один блок обработки, список участников обоих сервисов обновляется, и в нем отражается IP-адрес и порт каждого блока обработки:

```
Instance 1:
Members {size:2, ver:2} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 this
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316
]

Instance 2:
Members {size:2, ver:2} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316 this
]
```

Когда появляется третий блок обработки, списки участников экземпляра 1 и экземпляра 2 обновляются, чтобы отразить в составе списка новый третий экземпляр:

```
Instance 1:
Members {size:3, ver:3} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 this
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753
]

Instance 2:
Members {size:3, ver:3} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316 this
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753
]

Instance 3:
Members {size:3, ver:3} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753 this
]
```

Заметьте, что все три экземпляра знают о существовании друг друга (включая самих себя). Предположим, экземпляр 1 получает запрос на обновление информации профиля клиента. Когда экземпляр 1 обновляет кэш с помощью метода `cache.put()` или другого похожего метода обновления, сетка данных (например, Hazelcast) будет асинхронно вносить в другие реплицированные кэши те же обновления, обеспечивая постоянную синхронизацию всех трех кэшей профиля клиента.

Когда экземпляр блока обработки выходит из строя, все остальные блоки обработки автоматически обновляются, чтобы отразить потерю элемента. Например, если экземпляр 2 выходит из строя, списки участников экземпляров 1 и 3 обновляются следующим образом:

```
Instance 1:
Members {size:2, ver:4} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 this
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753
]

Instance 3:
Members {size:2, ver:4} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753 this
]
```

Сетка обработки

Сетка обработки (processing grid), показанная на рис. 15.6, является дополнительным компонентом виртуализированного связующего ПО, которое управляет согласованной обработкой запросов, если в одном бизнес-запросе задействовано сразу несколько блоков обработки. Когда поступает запрос, требующий координации между типами блоков обработки (например, блоком обработки заказа и блоком обработки платежей), медиатором и оркестратором между этими двумя блоками обработки выступает именно сетка обработки.

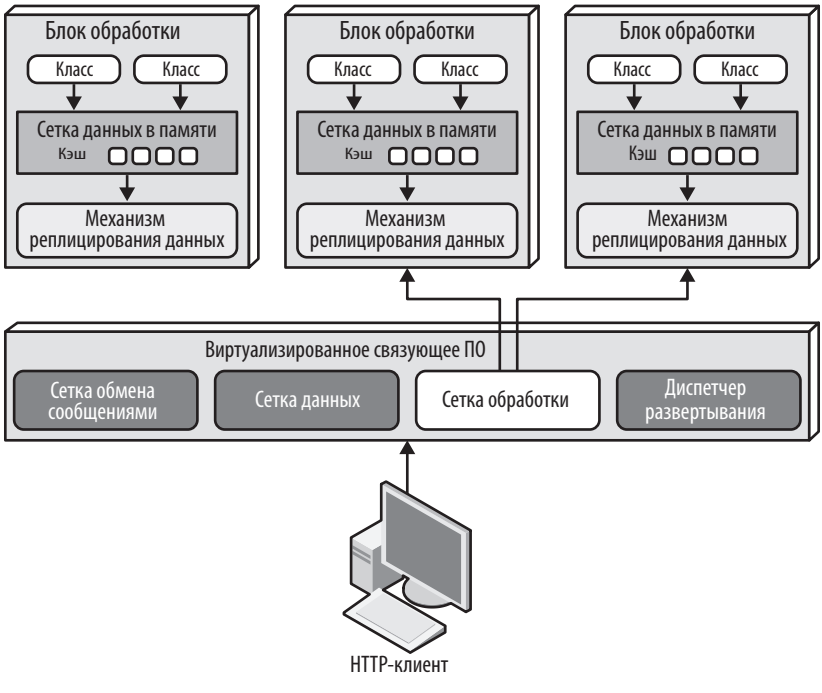


Рис. 15.6. Сетка обработки

Диспетчер развертывания

Компонент диспетчера развертывания (deployment manager) управляет динамическим запуском и завершением работы экземпляров блока обработки в зависимости от условий нагрузки. Этот компонент постоянно отслеживает время отклика и объем пользовательской нагрузки, запуская новые блоки обработки при увеличении нагрузки и выключая блоки при уменьшении нагрузки. Это наиболее важный компонент для достижения переменной масштабируемости (адаптируемости) приложения.

Средства переноса данных

Средство переноса данных (*data pump*) позволяет отправлять данные другому обработчику, который затем обновляет информацию в базе данных. Это необходимый компонент архитектуры на основе пространства, поскольку блоки обработки не выполняют чтение и запись напрямую в БД. Перенос данных в архитектуре на основе пространства всегда выполняется асинхронно, что в конечном счете обеспечивает согласованность кэша в памяти с базой данных. Когда экземпляр блока обработки получает запрос и обновляет свой кэш, этот же блок обработки становится владельцем обновления и, следовательно, отвечает за отправку данного обновления средством переноса данных для итогового обновления БД.

Средства переноса данных обычно реализуются с помощью обмена сообщениями (рис. 15.7). Обмен сообщениями — хороший выбор для переноса данных при использовании архитектуры на основе пространства. Обмен сообщениями поддерживает не только асинхронную связь, но и гарантированную доставку и сохранение порядка следования сообщений посредством организации очередей «первым пришел — первым вышел» (FIFO). Кроме того, обмен сообщениями обеспечивает разделение блока обработки и средства записи данных, что позволяет сохранять работоспособность блоков обработки, если недоступно средство записи.

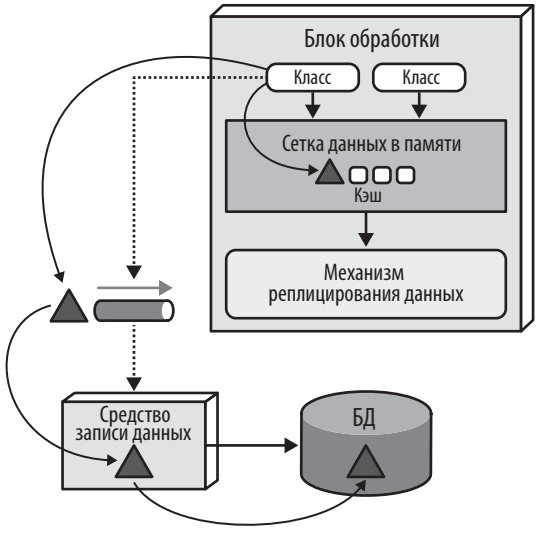


Рис. 15.7. Средство переноса данных, используемое для отправки информации в базу данных

В большинстве случаев используется несколько средств переноса данных, каждое из которых обычно посвящено определенной предметной области или ее подразделу (например, клиенту или складским запасам). Средства переноса данных могут быть выделены для каждого типа кэша (например, `CustomerProfile`, `CustomerWishlist` и т. д.), или же они могут быть предназначены для блока обработки предметной области (например, `Customer`), содержащего гораздо более объемный общий кэш.

Средства переноса данных обычно имеют связанные контракты, включающие действие с данными контракта (добавление, удаление или обновление). Контракт может быть JSON-схемой, XML-схемой, объектом или даже сообщением, содержащим пары имя — значение. Для выполнения обновления данные, передающиеся в сообщении средства переноса данных, содержат только новые значения. Например, если клиент изменяет номер телефона в своем профиле, то вместе с идентификатором клиента и запросом на обновление данных будет отправлен только новый номер.

Средства записи данных

Компонент записи данных принимает сообщения от средства переноса данных и обновляет базу данных информацией из этого сообщения (рис. 15.7). Средства записи данных могут быть реализованы в виде сервисов, приложений или концентраторов данных (например, Ab Initio[1]). Степень детализации средств записи данных может варьироваться в зависимости от сферы применения и объема средств переноса данных и блоков обработки.

Средство записи данных на основе предметной области содержит всю необходимую логику базы данных, применяемую при обработке всех обновлений в пределах определенной предметной области (например, клиента) независимо от количества средств переноса данных, от которых он принимает информацию. Заметьте, что на рис. 15.8 имеются четыре разных блока обработки и четыре разных средства переноса данных, представляющих предметную область клиента: профиль, вишлист, кошелек и предпочтения (`Profile`, `WishList`, `Wallet` и `Preferences`), но при этом используется только одно средство записи данных. Единое средство записи данных о клиентах отслеживает все четыре средства переноса данных и содержит всю необходимую логику базы данных (например, SQL) для обновления информации о клиентах в БД.

Как вариант, каждый класс блока обработки может иметь свое собственное выделенное средство записи данных (рис. 15.9). В данной модели средство записи

[1] https://www.abinitio.com/en

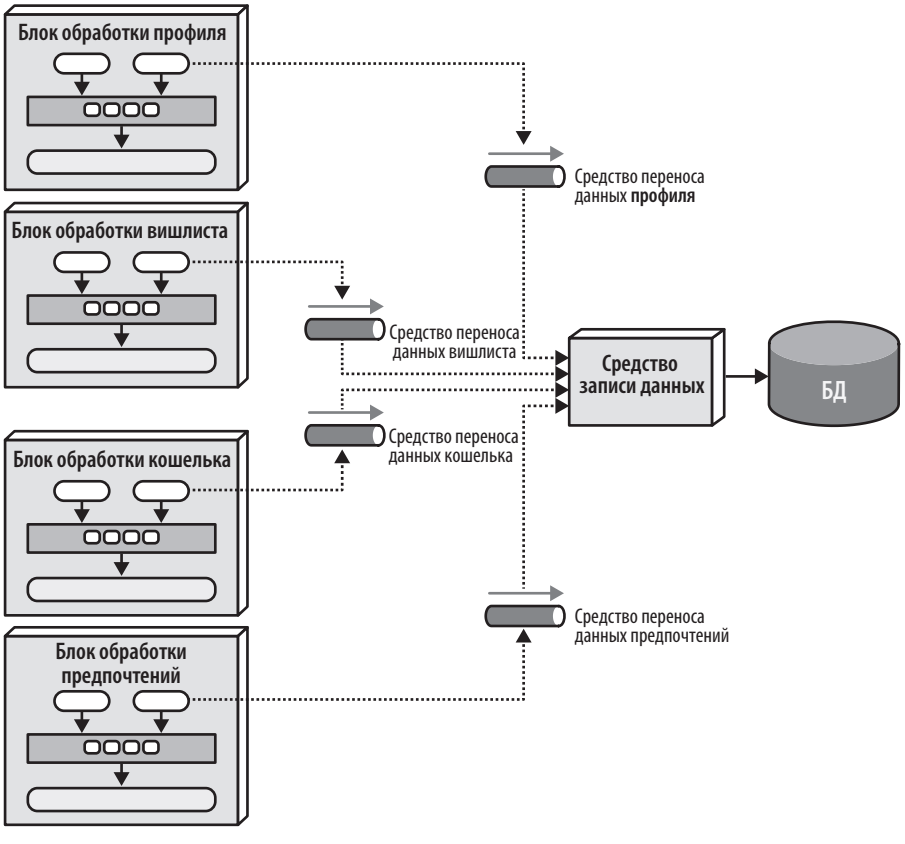


Рис. 15.8. Средство записи данных на основе предметной области

данных выделено каждому соответствующему средству переноса данных и содержит только логику обработки базы данных для этого конкретного блока обработки (например, `Wallet`). Хотя в этой модели прослеживается склонность к созданию слишком большого количества средств записи данных, она обеспечивает более высокую масштабируемость и гибкость за счет согласованности блока обработки, средства переноса данных и средства записи данных.

Средства чтения данных

Если средства записи данных отвечают за обновление базы данных, то средства чтения данных отвечают за считывание информации из БД и ее отправку в блоки обработки с помощью средства обратного переноса данных. В архитектуре на

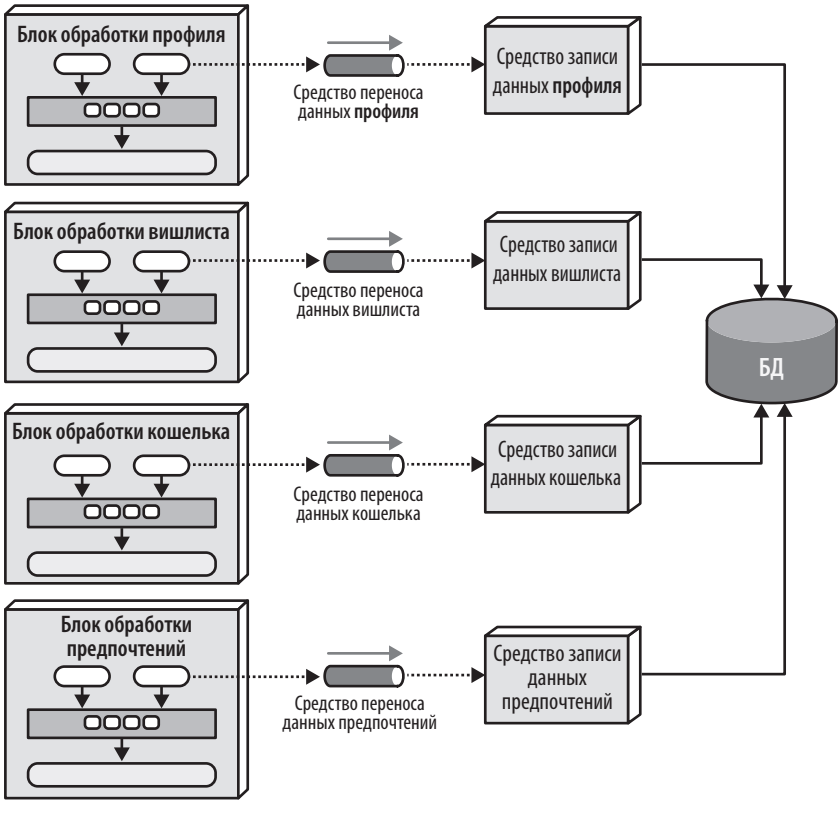


Рис. 15.9. Средства записи данных, выделенные каждому средству их переноса

основе пространства средства чтения данных вызываются только в одной из трех ситуаций: при сбое всех экземпляров блоков обработки с одним и тем же именованным кэшем, при повторном развертывании всех блоков обработки с одним и тем же именованным кэшем или при получении архивных данных, не содержащихся в реплицированном кэше.

В случае отказа всех экземпляров (из-за сбоя всей системы или повторного развертывания всех экземпляров) данные должны быть прочитаны из БД (чего в архитектуре на основе пространства обычно стараются избегать). Когда начинают появляться экземпляры класса блока обработки, каждый из них пытается захватить блокировку кэша. Первый захвативший блокировку становится временным владельцем кэша; все остальные переходят в состояние ожидания, пока блокировка не будет снята (в зависимости от типа реализации используемого кэша могут быть и другие варианты, но в данном сценарии будет только один

основной владелец кэша). Чтобы загрузить кэш, экземпляр, получивший статус его временного владельца, отправляет сообщение в очередь, запрашивая данные. Средство чтения данных принимает запрос на чтение и затем выполняет запрос к БД для получения необходимой информации, а затем отправляет эти данные в другую очередь (она называется средством обратной передачи данных). Блок обработки, владеющий временным кэшем, принимает данные от средства обратной передачи данных и загружает кэш. Как только вся информация будет загружена, временный владелец снимет блокировку кэша, а затем все остальные экземпляры синхронизируются, после чего можно приступать к обработке. Этот процесс показан на рис. 15.10.

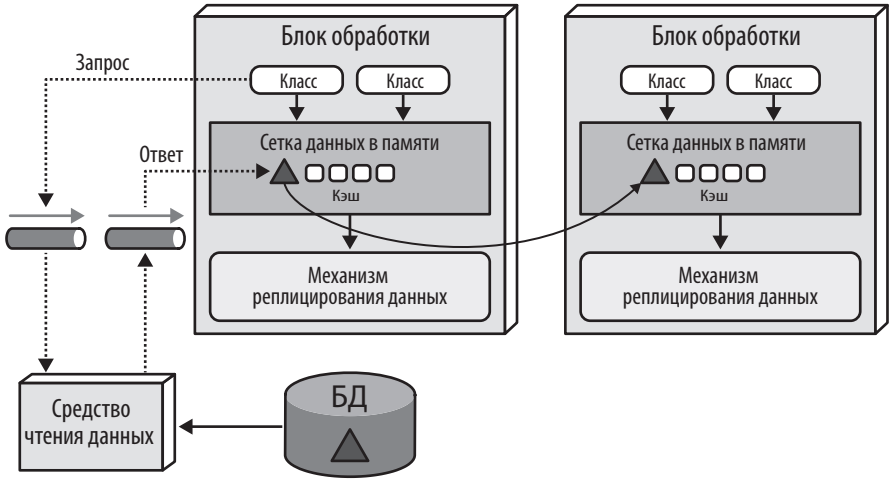


Рис. 15.10. Средство чтения данных со средством обратного переноса данных

Как и средства записи данных, средства чтения данных также могут основываться на предметной области или выделяться конкретному классу блоков обработки (что обычно и бывает). Реализуются они так же, как и средства записи данных: в виде сервисов, приложений или концентраторов данных.

Средства записи и чтения, по сути, образуют то, что обычно называют *уровнем абстракции данных* (или в некоторых случаях *уровнем доступа к данным*). Разница между ними заключается в объеме знаний, имеющихся у блоков обработки относительно структуры таблиц (или схемы) БД. Уровень доступа к данным означает, что блоки обработки связаны с исходными информационными структурами БД и используют средства чтения и записи только для скрытого доступа к БД. С другой стороны, уровень абстракции данных означает, что блок обработки с помощью отдельных контрактов изолирован от исходных структур

таблиц БД. Обычно архитектура на основе пространства полагается на модель уровня абстракции данных, поэтому схема репликации кэша в каждом блоке обработки может отличаться от исходных структур таблиц БД. Это позволяет вносить в базу данных постепенные изменения без обязательного воздействия на блоки обработки. Чтобы облегчить эти постепенные изменения, средства записи и средства чтения содержат логику преобразования, чтобы в случае изменения типа столбца или удаления столбца или таблицы они могли бы буферизовать модификацию базы данных до тех пор, пока необходимые обновления не будут внесены в кэш блока обработки.

Коллизии данных

При использовании реплицированного кэширования в режиме «активный-активный», где обновления могут происходить в любом экземпляре сервиса, содержащем кэш с одним и тем же именем, высока вероятность *коллизии данных*, вызванной задержкой репликации. Эта коллизия возникает при обновлении данных в одном экземпляре кэша (кэш А), во время репликации в другой экземпляр кэша (кэш Б), где те же самые данные обновляются в этом кэше (кэш Б). В данном сценарии локальное обновление кэша Б не состоится из-за репликации старых данных из кэша А, а обновленные данные в кэше А будут заменены такими же, но устаревшими данными из кэша Б из-за той же репликации.

Чтобы уяснить суть проблемы, предположим, что имеются два экземпляра сервиса (Сервис А и Сервис Б), содержащие реплицированный кэш товарных запасов. Проблему коллизии данных можно продемонстрировать следующей чередой событий:

- Текущий запас товара составляет 500 единиц.
- Сервис А обновляет кэш товарного запаса, устанавливая его значение в 490 единиц (10 единиц продано).
- В то время, пока происходит связанная с предыдущим действием репликация, Сервис Б обновляет кэш товарного запаса, устанавливая его значение в 495 единиц (5 единиц продано).
- Кэш Сервиса Б обновляется значением 490 из-за репликации, вызванной обновлением, выполненным Сервисом А.
- Кэш Сервиса А обновляется значением 495 из-за репликации, вызванной обновлением, выполненным Сервисом Б.

- Оба кэша, в Сервисе А и Сервисе Б, не синхронизированы и содержат неправильные значения (запас товара должен быть 485 единиц).

Есть несколько факторов, влияющих на количество возможных коллизий данных: количество экземпляров блока обработки с одним и тем же кэшем, скорость обновления кэша, размер кэша и, наконец, задержка репликации, свойственная программе кэширования. Формула, используемая для вероятностного определения числа потенциальных коллизий данных на основе этих факторов, выглядит следующим образом:

$$CollisionRate = N \times \frac{UR^2}{S} \times RL,$$

где N — количество экземпляров сервисов, использующих кэши с одинаковыми именами, UR — частота обновлений в миллисекундах (возведенная в квадрат), S — размер кэша (количество строк) и RL — задержка репликации кэширования. Эта формула пригодится для определения вероятного процента коллизии данных и, следовательно, возможности использования реплицированного кэширования. Рассмотрим, к примеру, следующие значения факторов, учитываемых в этом расчете:

Частота обновлений (UR)	20 обновлений в секунду
Количество экземпляров (N)	5
Размер кэша (S)	50 000 строк
Задержка репликации (RL)	100 мс
Обновления	72 000 в час
Частота коллизий	14,4 в час
Процентный показатель	0,02%

Применив эти коэффициенты к формуле, получим 72 000 обновлений в час с высокой вероятностью, что 14 обновлений одних и тех же данных могут привести к конфликтам. Учитывая низкий процентный показатель (0,02 %), такая репликация будет вполне жизнеспособным вариантом.

Изменение показателя задержки репликации оказывает существенное влияние на согласованность данных. Продолжительность задержки зависит от многих факторов, включая тип сети и физическое расстояние между блоками обработки. Именно поэтому значения задержки репликации редко публикуются и должны рассчитываться и получаться на основе фактических измерений в производ-

ственной среде. Значение, использованное в предыдущем примере (100 мс), является вполне подходящим показателем для планирования, если фактическая задержка, значение которой часто используется для определения количества коллизий данных, недоступна. Например, изменение задержки от 100 мс до 1 мс дает такое же количество обновлений (72 000 в час), но вероятность конфликтов снижается до 0,1 в час! Этот сценарий показан в следующей таблице:

Частота обновлений (UR)	20 обновлений в секунду
Количество экземпляров (N)	5
Размер кэша (S)	50 000 строк
Задержка репликации (RL)	1 мс (прежде было 100)
Обновления	72 000 в час
Частота коллизий	0,1 в час
Процентный показатель	0,0002%

Количество блоков обработки, содержащих кэш с одним и тем же именем (представленное в виде фактора *количества экземпляров*) также напрямую влияет на количество возможных конфликтов данных. Например, уменьшение количества блоков обработки с 5 экземпляров до 2 доводит частоту конфликтов данных до 6 в час при 72 000 обновлений в час:

Частота обновлений (UR:	20 обновлений в секунду
Количество экземпляров (N)	2 (прежде было 5)
Размер кэша (S)	50 000 строк
Задержка репликации (RL)	100 мс
Обновления	72 000 в час
Частота коллизий	5,8 в час
Процентный показатель	0,008%

Размер кэша является единственным фактором, значение которого обратно пропорционально частоте конфликтов. По мере уменьшения размера кэша частота конфликтов увеличивается. В нашем примере уменьшение размера кэша с 50 000 до 10 000 строк (при сохранении всего, что было в первом примере, в неизменном виде) дает частоту конфликтов 72 в час, что значительно выше, чем при 50 000 строках кэша:

Частота обновлений (UR)	20 обновлений в секунду
Количество экземпляров (N)	2 (прежде было 5)
Размер кэша (S)	10 000 строк (прежде было 50 000)
Задержка репликации (RL)	100 мс
Обновления	72 000 в час
Частота коллизий	72,0 в час
Процентный показатель	0,1%

В обычных условиях большинство систем не имеют постоянной частоты обновлений в течение столь длительного периода времени. Поэтому при использовании данного расчета полезно выяснить значение максимальной частоты обновлений при пиковом использовании системы и вычислить минимальную, нормальную и пиковую частоту коллизий.

Облачные и локальные реализации

Когда речь заходит о средах развертывания, архитектура на основе пространства предлагает ряд уникальных вариантов. Вся топология, включая блоки обработки, виртуализированное связующее ПО, средства переноса данных, средства чтения и записи данных, а также базу данных, может быть развернута в облачных средах по модели «on-prem»[1]. Но архитектура этого стиля может также развертываться одновременно в двух средах, предлагая уникальные функциональные возможности, отсутствующие в архитектурах других стилей.

Весьма эффективная особенность архитектуры на основе пространства — возможность развертывания приложений с помощью блоков обработки и виртуализированного связующего ПО в управляемых облачных средах; при этом сохраняются локально (on-prem) физические базы данных и соответствующие данные (рис. 15.11). Эта топология поддерживает высокоэффективную синхронизацию данных в облаке благодаря асинхронному переносу данных и модели конечной согласованности, присущей архитектурам данного стиля. Обработка транзакций может происходить в динамических и адаптируемых облачных средах, а управление физическими данными, отчеты и аналитика данных будут при этом храниться в безопасных локальных средах.

[1] Модель распространения по подписке, но программное обеспечение устанавливается на серверах заказчика.

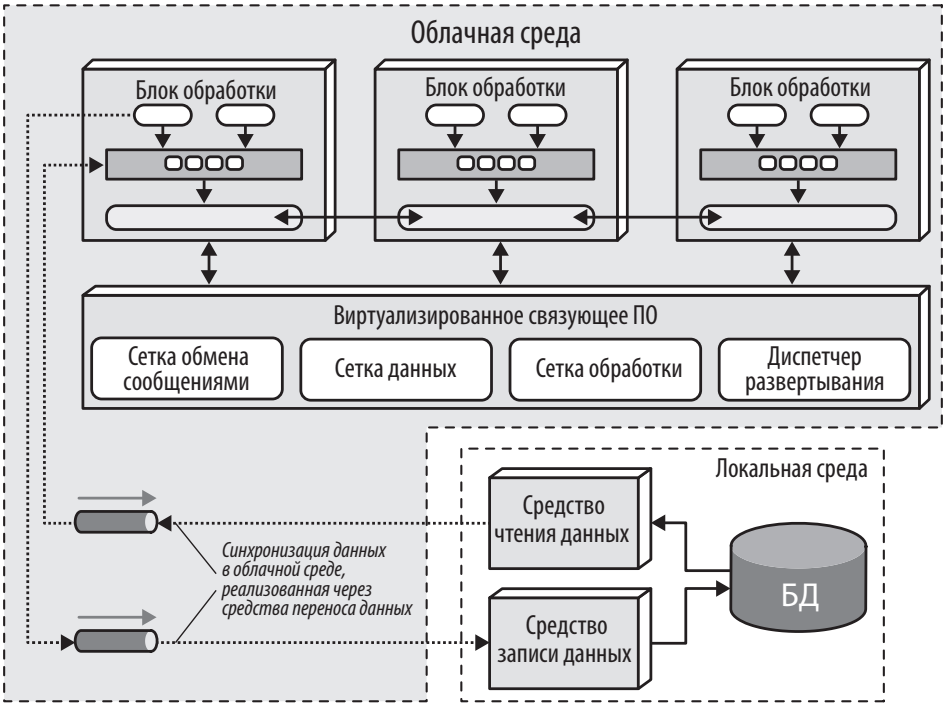


Рис. 15.11. Гибридная топология, построенная на основе облачных и местных сред

Реплицированное и распределенное кэширование

Архитектура на основе пространства базируется на кэшировании всех транзакций в приложении. Отсутствие необходимости прямого чтения и записи в базу данных позволяет такой архитектуре поддерживать высокую масштабируемость, адаптируемость и производительность. Архитектура на основе пространства полагается в основном на реплицированное кэширование, хотя в ней также может использоваться и распределенное.

При реплицированном кэшировании (рис. 15.12) каждый блок обработки содержит свою собственную сетку данных в памяти, которая синхронизируется между всеми блоками обработки, используя кэш с одним и тем же именем. Когда происходит обновление кэша в любом из блоков обработки, информация в других блоках обработки обновляется в автоматическом режиме.

Реплицированное кэширование обладает не только завидной скоростью, но и высокой отказоустойчивостью. При отсутствии центрального сервера, хранящего

кэш, кэширование не имеет единой точки отказа. Но в зависимости от особенностей реализации используемого программного средства здесь могут быть исключения. Некоторые средства кэширования требуют внешнего контроллера для отслеживания репликации данных среди блоков обработки и управления этим процессом, но большинство производителей уже отходят от этой модели.

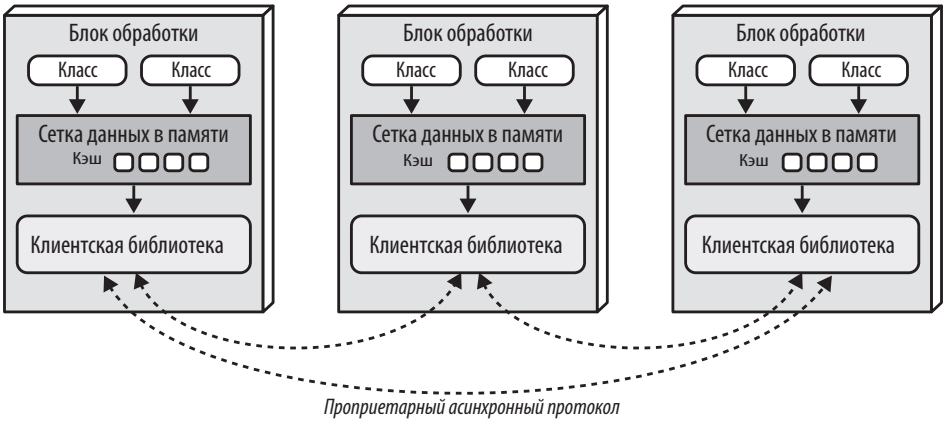


Рис. 15.12. Реплицированное кэширование, распространяющееся на блоки обработки

Реплицированное кэширование является для архитектуры на основе пространства стандартной моделью, но в некоторых случаях воспользоваться им просто невозможно. Такое бывает при больших объемах данных (большом размере кэша) и высокой частоте обновления данных кэша. Внутренняя кэш-память, превышающая 100 Мбайт, может воспрепятствовать высокой адаптируемости и масштабируемости из-за объема памяти, используемого каждым блоком обработки. Эти блоки обычно развертываются внутри виртуальной машины (а в некоторых случаях сами представляют собой такую машину). У каждой виртуальной машины имеется вполне определенный объем памяти, доступный для использования внутренним кэшем, что ограничивает количество экземпляров блоков обработки, которые могут быть запущены для работы, требующей высокой пропускной способности. Кроме того, как уже говорилось раньше в разделе «Коллизии данных» на с. 269, если частота обновления данных кэша слишком высока, сетка данных может оказаться не в состоянии поддерживать такую высокую частоту с гарантированной согласованностью данных во всех экземплярах блока обработки. При возникновении подобных ситуаций можно воспользоваться распределенным кэшированием.

Для распределенного кэширования (рис. 15.13) требуется внешний сервер или сервис, предназначенный для хранения централизованного кэша. В этой мо-

дели блоки обработки не хранят данные во внутренней памяти, а используют собственный протокол для доступа к данным, хранящимся на центральном кэш-сервере. Распределенное кэширование поддерживает высокий уровень согласованности данных, поскольку все они хранятся в одном месте и не требуют репликации. Но эта модель обладает меньшей производительностью, чем реплицированное кэширование, поскольку к данным кэша необходимо обращаться удаленно, что увеличивает общую задержку системы. Отказоустойчивость также является слабой стороной распределенного кэширования. Если кэш-сервер, содержащий данные, выходит из строя, доступ ко всем данным для чтения и обновления закрывается, что делает блоки неработоспособными. Отказоустойчивость может быть повышена за счет зеркального отображения распределенного кэша, но тут могут возникнуть проблемы согласованности, когда основной сервер кэширования неожиданно выйдет из строя и данные не попадут на сервер зеркального кэша.

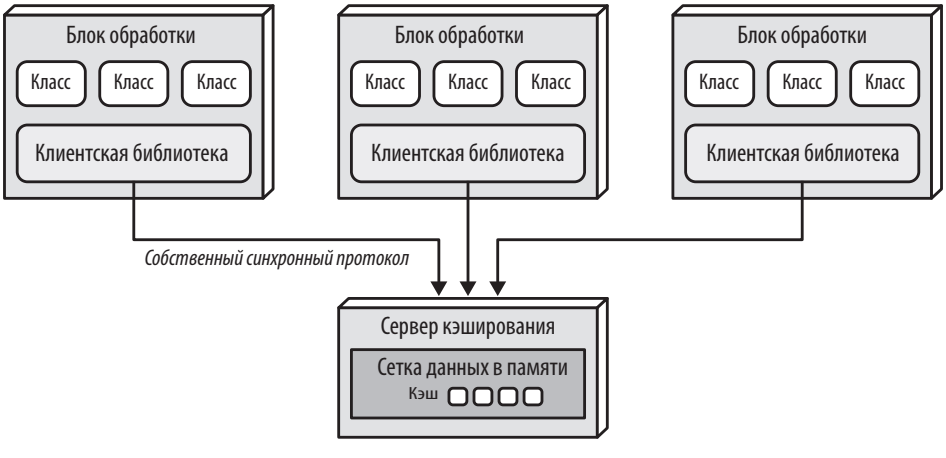


Рис. 15.13. Распределенное кэширование, распространяющееся на блоки обработки

Когда размер кэша сравнительно невелик (менее 100 Мбайт) и частота его обновления достаточно мала, что позволяет репликации программного средства кэширования не отставать от обновлений кэша, выбор между использованием реплицированного и распределенного кэша сводится к выбору между согласованностью данных и производительностью в совокупности с отказоустойчивостью. Распределенный кэш по сравнению с реплицированным неизменно отличается лучшей согласованностью данных, поскольку кэш данных находится в одном месте (а не разбросан по нескольким блокам обработки). Но производительность и отказоустойчивость всегда будут выше при использовании реплицированного кэша. Часто решение по выбору кэша сводится к типу данных,

кэшируемых в блоках обработки. Потребность в данных с высокой степенью согласованности (таких, как запасы доступных товаров) обычно склоняет чашу весов в пользу распределенного кэша, тогда как для быстрого поиска данных, не подвергаемых частым изменениям (например, справочной информации, составленной из пар имя-значение, кодов товаров и их описания), обычно требуется реплицированный кэш. Некоторые критерии выбора между распределенным и реплицированным кэшем перечислены в табл. 15.1.

Таблица 15.1. Сравнение распределенного и реплицированного кэширования

Критерии принятия решения	Реплицированный кэш	Распределенный кэш
Оптимизация	Производительность	Согласованность
Размер кэша	Небольшой (<100 Мбайт)	Большой (>500 Мбайт)
Тип данных	Относительно статичный	Высокодинамичный
Частота обновлений	Относительно низкая	Высокая
Отказоустойчивость	Высокая	Низкая

При выборе типа модели кэширования, применяемой с архитектурой на основе пространства, следует помнить, что в большинстве случаев в любом заданном контексте приложения применимы *обе* модели. Иными словами, ни реплицированное, ни распределенное кэширование не решает всех проблем. Вместо попытки поиска компромиссов с применением единой согласованной модели кэширования для всего приложения нужно воспользоваться каждой из сильных сторон обеих моделей. Например, для высокой согласованности данных блока обработки, поддерживающего текущие запасы товара, следует выбрать модель распределенного кэширования, а для блока обработки, поддерживающего профиль клиента, нужно выбрать модель реплицированного кэша, позволяющую обеспечить высокую производительность и отказоустойчивость.

Особенности near-cache

Near-cache (иногда называют *кэшем клиента*) — это тип гибридной модели кэширования, обеспечивающей связь между сеткой данных в памяти и распределенным кэшем. В этой модели (рис. 15.14) распределенный кэш называется *полным бэк-кэшем* (*full backing cache*), а каждая сетка данных в памяти, содержащаяся в каждом блоке обработки, называется *фронт-кэшем* (*front cache*). Во фронт-кэше всегда содержится небольшое подмножество полноценного бэк-

кэша, и в нем используется *политика вытеснения (eviction policy)*, позволяющая удалять старые элементы для добавления новых. Фронт-кэш может выступать в роли так называемого кэша последних использованных данных (most recently used, MRU), содержащего наиболее актуальные элементы данных, или же кэша наиболее часто используемых данных (most frequently used, MFU), содержащего самые востребованные элементы данных. Как вариант, во фронт-кэше может использоваться политика *произвольного вытеснения* (random replacement, RR), когда при дефиците пространства для добавления нового элемента удаление других элементов данных происходит в случайном порядке. RR является удачной политикой вытеснения при отсутствии четкого анализа данных, позволяющего придерживаться либо метода самых актуальных данных, либо метода самых востребованных данных.

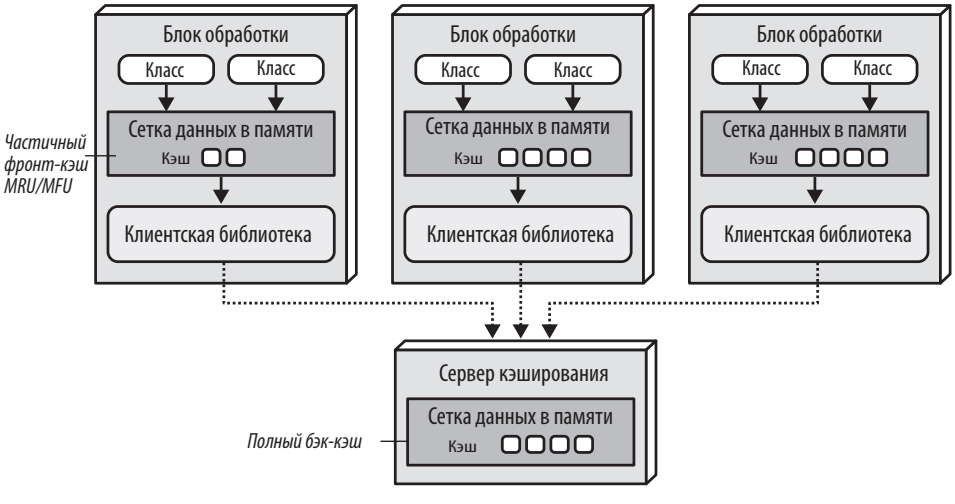


Рис. 15.14. Топология near-cashe

Экземпляры фронт-кэшей всегда синхронизированы с полным бэк-кэшем, но фронт-кэши, содержащиеся в каждом блоке обработки, не синхронизируются с кэшами других блоков обработки, использующими те же данные. Следовательно, несколько блоков обработки, использующих один и тот же контекст данных (например, профиль клиента), будут, скорее всего, иметь в своих фронт-кэшах разные значения данных. Возникает несоответствие в производительности и отзывчивости среди блоков обработки, поскольку каждый блок обработки содержит в своем фронт-кэше свой собственный вариант данных. Поэтому мы не рекомендуем использовать модель near-cashe в приложениях с архитектурой на основе пространства.

Примеры реализации

Архитектура на основе пространства хорошо подходит для приложений, испытывающих резкие скачки количества пользователей или запросов, а также для приложений, пропускная способность которых превышает 10 000 одновременно обращающихся к ним пользователей. К примерам использования архитектуры на основе пространства можно отнести такие приложения, как системы продажи концертных билетов и онлайн-аукционы. В обоих случаях требуются высокая производительность, масштабируемость и высокий уровень адаптируемости.

Система продажи концертных билетов

У системы продажи концертных билетов есть уникальная особенность: относительно небольшое количество одновременно обращающихся к ней пользователей до той поры, пока не будет объявлен концерт популярных исполнителей. Как только билеты на концерт поступят в продажу, объем одновременно обращающихся пользователей обычно взлетает с нескольких сотен до нескольких тысяч (возможно, в зависимости от концерта, и до десятков тысяч), и все они пытаются приобрести билет на концерт (с надеждой на хорошие места!). Билеты обычно распродаются за считаные минуты, для чего требуются архитектурные свойства, присущие архитектуре на основе пространства.

С такой системой связано множество серьезных вызовов. Прежде всего, независимо от предпочитаемых мест, доступно только определенное количество билетов. При большом количестве одновременных запросов наличие мест должно постоянно обновляться и предоставляться как можно быстрее. Если есть механизм бронирования, наличие мест также должно обновляться как можно быстрее. Постоянный синхронный доступ к центральной базе данных для такого типа системы, вероятно, не сработает, поскольку для типичной базы данных будет очень сложно обрабатывать десятки тысяч одновременных запросов через стандартные транзакции при таком уровне масштабирования и частоты обновления.

Высокие требования к адаптируемости, предъявляемые к подобному типу приложений, дают архитектуре на основе пространства довольно высокие шансы на то, что она сможет достойно справиться с системой продажи концертных билетов. Мгновенное увеличение числа одновременно обращающихся пользователей, желающих приобрести билеты на концерт, будет тут же замечено *диспетчером развертывания*, а тот, в свою очередь, запустит большое количество блоков обработки, способных справиться с возросшим объемом запросов. Лучше, чтобы диспетчер развертывания был настроен на запуск необходимого количества

блоков обработки незадолго *до* поступления билетов в продажу и чтобы новые экземпляры находились в режиме ожидания прямо перед всплеском пользовательской нагрузки.

Система онлайн-аукциона

Системы онлайн-аукционов (торговли лотами в режиме аукциона) обладают теми же характеристиками, что и рассмотренные ранее системы онлайн-продажи концертных билетов: обе требуют высокого уровня производительности и адаптируемости и обе имеют непредсказуемые всплески пользовательской нагрузки и объема запросов. В начале аукциона невозможно определить, сколько людей к нему присоединится и сколько из них одновременно выставят заявки на каждую запрашиваемую цену.

Архитектура на основе пространства хорошо приспособлена для решения таких задач, поскольку по мере увеличения нагрузки может запускаться сразу несколько блоков обработки, а по мере спада активности на аукционе неиспользуемые блоки обработки могут отключаться. Для каждого аукциона могут быть выделены отдельные блоки обработки, что обеспечит согласованность данных о торгах. Кроме того, асинхронной характер переноса данных позволяет отправлять сведения о торгах в обработку иного вида (такую, как ведение истории ставок, аналитика ставок и аудит) без какой-либо существенной задержки, что увеличивает общую производительность процесса назначения ставок.

Оценки архитектурных свойств

Одна звезда в оценочной таблице свойств (показанной на рис. 15.15) означает, что данное архитектурное свойство не имеет высокой поддержки в архитектуре, а пять звезд — что архитектурное свойство является одной из самых сильных сторон архитектурного стиля. Определение каждого свойства, указанного в оценочной таблице, можно найти в главе 4.

Следует отметить, что архитектура на основе пространства имеет максимальные показатели адаптируемости, масштабируемости и производительности (все пять звезд). Это наиболее яркие признаки и основные достоинства архитектур данного стиля. Высокие значения всех трех архитектурных свойств достигаются за счет применения кэширования данных в памяти и избавления от ограничивающих факторов, свойственных использованию базы данных. В результате архитектуры данного стиля позволяют системам вести обработку миллионов одновременно обращающихся пользователей.

Архитектурное свойство	Оценка в звездах
Тип разбиения	Предметный и технический
Количество квантов	От одного до нескольких
Развертываемость	☆☆☆
Адаптируемость	☆☆☆☆☆
Эволюционность	☆☆☆
Отказоустойчивость	☆☆☆
Модульность	☆☆☆
Общие затраты	☆☆
Производительность	☆☆☆☆☆
Надежность	☆☆☆☆
Масштабируемость	☆☆☆☆☆
Простота	☆
Тестируемость	☆

Рис. 15.15. Оценки свойств архитектуры на основе пространства

Высокие показатели адаптируемости, масштабируемости и производительности являются несомненными преимуществами архитектур данного стиля, но они достигаются компромиссами, за счет нанесения ущерба другим показателям, особенно общей простоте и тестируемости. Архитектура на основе пространства относится к весьма сложным архитектурным стилям из-за использования кэширования и сложностей в достижении конечной согласованности с основным хранилищем данных, являющимся конечной системой записи информации. Нужно позаботиться о том, чтобы данные не были потеряны в случае сбоя в любой из многочисленных подвижных частей этой архитектуры (см. раздел «Предотвращение потери данных» главы 14 на с. 244).

Из-за сложности моделирования высокого уровня масштабируемости и адаптируемости, свойственных архитектурам данного стиля, тестирование получает всего одну звезду. Проведение тестов с участием сотен тысяч одновременно работающих пользователей при пиковой нагрузке — весьма непростая и затратная

задача, в результате чего основная часть массового тестирования происходит в среде эксплуатации с реальной экстремальной нагрузкой. При этом создается существенный риск для нормальной работы в процессе эксплуатации.

Общие затраты являются еще одним фактором, заставляющим задуматься при выборе этого архитектурного стиля. Архитектура на основе пространства обходится относительно дорого, в основном из-за лицензионных сборов за программные средства кэширования и весьма интенсивного потребления ресурсов в облачных и локальных системах для поддержания высоких уровней масштабируемости и адаптируемости.

Однозначно определить тип разбиения в архитектуре на основе пространства довольно трудно, и в результате мы остановились на том, что разбиение ведется как по предметному, так и по техническому признаку. Разбиение по предметному признаку объясняется не только соответствием определенному типу предметной области (высокоэластичные и масштабируемые системы), но также гибкостью блоков обработки. Эти блоки могут действовать как предметные сервисы точно так же, как и те сервисы, которые определены в архитектуре на основе сервисов или в архитектуре микросервисов. В то же время архитектура на основе пространства имеет разбиение и по техническому принципу, так как в ней с помощью средств переноса данных задачи, связанные с обработкой транзакций с использованием кэширования, отделены от фактического хранения информации в базе данных. Блоки обработки, средства переноса данных, средства чтения и записи данных, а также база данных в совокупности образуют с точки зрения обработки запросов техническую многослойность, очень похожую на устройство монолитной многоуровневой архитектуры.

Количество квантов в архитектуре на основе пространства может варьироваться в зависимости от пользовательского интерфейса и способа обмена данными между блоками обработки. Поскольку блоки обработки не имеют синхронного взаимодействия с базой данных, сама эта база не входит в подсчет квантов. В результате кванты в архитектуре на основе пространства обычно разграничиваются посредством связи между различными пользовательскими интерфейсами и блоками обработки. Все блоки обработки, которые синхронно взаимодействуют друг с другом (или синхронно через сетку обработки для оркестровки), будут частью одного архитектурного кванта.

ГЛАВА 16

Оркестрированная сервис-ориентированная архитектура

Осмысление архитектурных стилей, как и направлений в искусстве, должно вестись в контексте эпохи их развития, и архитектура, о которой пойдет речь в данной главе, лучше других подтверждает это правило. Внешние факторы, часто влияющие на архитектурные решения, вместе с логичной на первый взгляд, но в итоге провальной организационной философией привели к тому, что эта архитектура стала неактуальной. И тем не менее это наглядный пример того, как конкретная организационная идея может иметь логический смысл, но при этом мешать наиболее важным частям процесса разработки.

История и философия

Данный стиль сервис-ориентированных архитектур появился, когда в конце 1990-х годов компании становились предприятиями: сливаясь с другими небольшими компаниями, они росли с невероятной скоростью, и для поддержки своего роста требовали все более сложных информационных технологий. Но вычислительные ресурсы были весьма скудными, дорогостоящими и коммерческими. Именно тогда распределенные вычисления стали доступны и востребованы, а многим компаниям понадобилась переменная масштабируемость и другие полезные свойства.

Многие внешние факторы заставляли архитекторов того времени переходить к распределенным архитектурам с существенными ограничениями. До того как операционные системы с открытым исходным кодом стали считаться достаточно надежными для серьезной работы, использовались дорогие операционные системы с лицензией для каждой машины. Коммерческие серверы

баз данных также поставлялись с запутанными «византийскими» схемами лицензирования, что заставляло поставщиков серверов приложений (предлагавших подключения к базам данных) воевать с поставщиками баз данных. Таким образом, от архитекторов требовалось максимально возможное повторное использование ресурсов. Фактически *многократное использование* во всех формах стало доминирующей философией в этой архитектуре, побочные эффекты которой мы рассмотрим далее в разделе «Многократное использование... и связывание» на с. 287.

Данный архитектурный стиль также показывает, насколько далеко архитекторы могут зайти с идеей технического разбиения, у которой благие цели, но пагубные последствия.

Топология

Топология этого типа сервис-ориентированных архитектур показана на рис. 16.1.

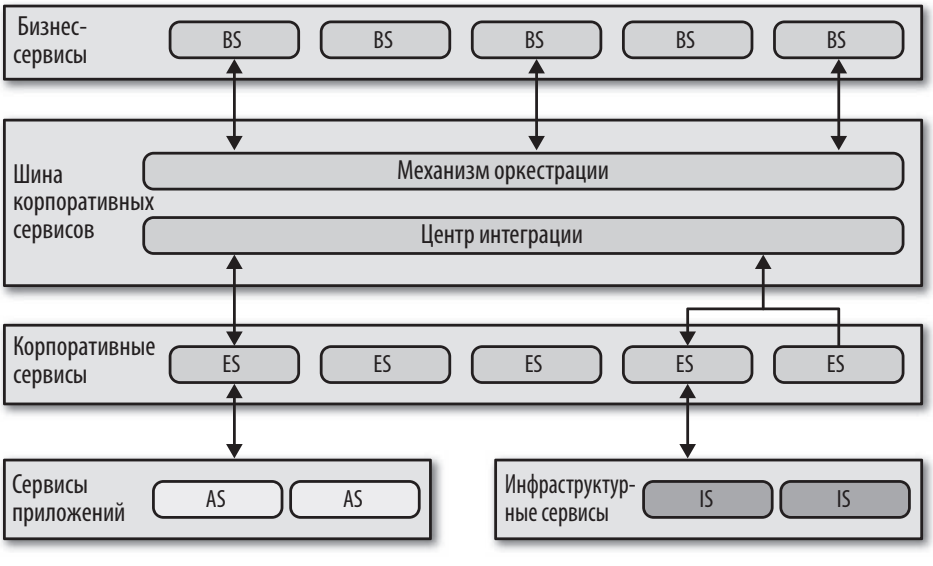


Рис. 16.1. Топология оркестрированной сервис-ориентированной архитектуры

Не все образцы архитектур этого стиля выглядят так, как на рис. 16.1, но все они следуют одной и той же идее систематизации (таксономии) сервисов в архитектуре, где каждый уровень выполняет определенную задачу.

Архитектура, ориентированная на сервисы, является распределенной, но точное обозначение границ на рис. 16.1 не показано, поскольку эти границы варьируются в зависимости от той или иной организации.

Таксономия

Основная идея данной архитектуры состоит в многократном использовании ресурсов на уровне предприятия. Руководство многих крупных компаний раздражало слишком частое переписывание ПО, и тогда была разработана стратегия постепенного решения данной проблемы. Каждый уровень таксономии служил для достижения этой цели.

Бизнес-сервисы

Бизнес-сервисы помещены на вершине данной архитектуры и предоставляют точку входа. К примеру, такие сервисы, как `ExecuteTrade` (совершить сделку) или `PlaceOrder` (разместить ордер), относятся к поведению предметной области. В те времена любили использовать такую лакмусовую бумажку — может ли архитектор утвердительно ответить для каждого из сервисов на вопрос: «Решаем ли мы задачи бизнеса?».

Определения этих сервисов не содержали кода: в них был только ввод, вывод и иногда информация о схеме. Обычно они определялись бизнес-пользователями, отсюда и название *бизнес-сервисы*.

Корпоративные сервисы

Корпоративные сервисы содержат узкоспециализированные совместно используемые реализации. Как правило, перед командой разработчиков стоит задача выстроить атомарное поведение вокруг конкретных бизнес-областей: `CreateCustomer` (создать клиента), `CalculateQuote` (рассчитать котировку) и т. д. Эти сервисы представляют собой строительные блоки, из которых составляются общие бизнес-сервисы, связанные вместе с помощью механизма оркестрации.

Цель этого разделения ответственности в архитектуре — добиться многократного использования кода. Если разработчики смогут создать узкоспециализированные корпоративные сервисы на правильном уровне детализации, бизнесу не придется снова и снова переписывать рабочие бизнес-процессы. Постепенно бизнес будет создавать коллекцию повторно используемых корпоративных сервисов.

К сожалению, постоянно меняющаяся реальность сводит на нет все планы. Бизнес-компоненты не похожи на строительные материалы, которые служат десятилетиями. Рынки, технологические изменения, инженерные практики и множество других факторов не позволяют навязать стабильность миру программного обеспечения.

Сервисы приложений

Не всем сервисам данной архитектуры требуется такой же уровень гранулярности и возможности многократного использования, как корпоративные сервисы. *Сервисы приложений* одноразовые и имеют свою собственную однократную реализацию. К примеру, какому-то приложению может понадобиться геолокация, но организация не хочет тратить время или силы на то, чтобы сделать этот сервис доступным для общего многократного использования. Такие задачи решаются сервисом приложений, принадлежащим обычно одной команде разработчиков.

Инфраструктурные сервисы

Инфраструктурные сервисы обеспечивают решения эксплуатационных задач: мониторинга, ведения журнала, проведения аутентификации и авторизации. Как правило, эти сервисы представляют собой конкретные реализации, принадлежащие команде общей инфраструктуры, чья работа тесно связана с эксплуатацией.

Механизм оркестрации

Механизм оркестрации является центральной частью данной распределенной архитектуры, сшивая вместе реализации бизнес-сервисов и выполняя такие функции, как координация транзакций и преобразование сообщений. Рассматриваемая архитектура обычно привязана к одной или нескольким реляционным базам данных, и у нее нет баз данных для каждого сервиса, как в микросервисных архитектурах. Поэтому транзакционное поведение декларативно обрабатывается в механизме оркестрации, а не в базе данных.

Механизм оркестрации определяет взаимоотношения между бизнес-сервисами и корпоративными сервисами, их совместную работу и границы транзакций. Он также действует в качестве центра интеграции, позволяя архитекторам согласовывать собственный код с пакетами и унаследованными программными системами.

Поскольку этот механизм составляет основу архитектуры, закон Конвея (см. врезку «Закон Конвея» в главе 8 на с. 138) предопределяет, что команда архитекторов, ответственная за него, станет политической силой внутри организации и, в конечном итоге, узким бюрократическим местом.

Каким бы привлекательным ни казался этот подход, на практике он чаще всего приводит к крайне неприятным последствиям. Передача поведения транзакции в инструмент оркестрации казалась вполне разумной идеей, но найти правильный уровень гранулярности транзакций становилось все труднее и труднее. Создать несколько сервисов, охваченных распределенной транзакцией, конечно, можно, но это усложнит архитектуру, поскольку разработчикам нужно будет выяснить, где проходят границы транзакции между сервисами.

Поток сообщений

Все запросы проходят через механизм оркестрации — то место в архитектуре, где находится логика. Таким образом, как показано на рис. 16.2, поток сообщений проходит через этот механизм даже для внутренних вызовов.

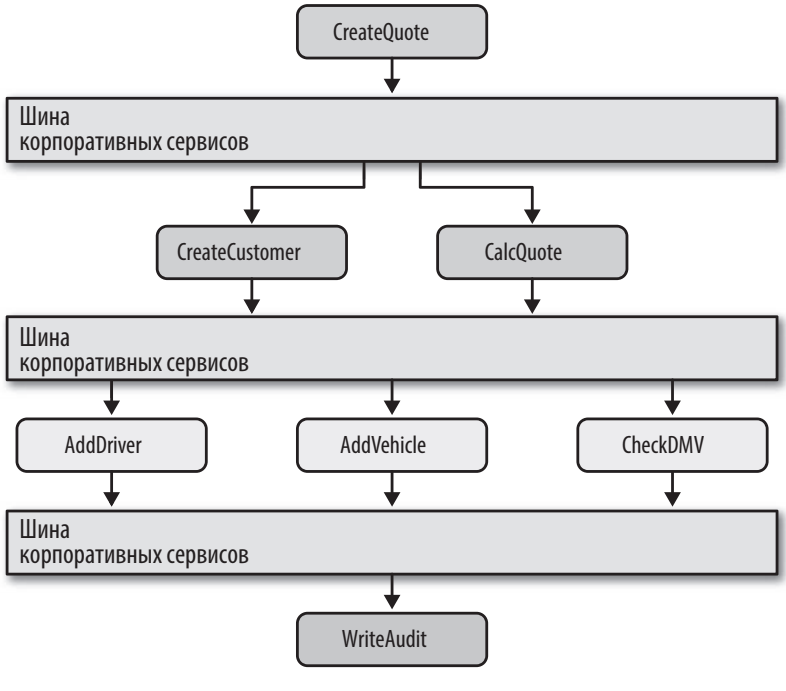


Рис. 16.2. Поток сообщений в архитектуре, ориентированной на использование сервисов

На рис. 16.2 сервис бизнес-уровня `CreateQuote` вызывает сервисную шину, которая определяет рабочий процесс, состоящий из вызовов `CreateCustomer` и `CalculateQuote`, каждый из которых также использует вызовы сервисов приложений. Сервисная шина действует в этой архитектуре в качестве посредника для всех вызовов, выступая одновременно в роли центра интеграции и механизма оркестрации.

Многократное использование… и связывание

Основная цель данной архитектуры — добиться многократного использования на уровне сервисов, то есть получить возможность постепенного выстраивания бизнес-поведения, которое можно было бы многократно и с нарастающей интенсивностью использовать в дальнейшем. Архитекторам, применяющим данную архитектуру, было поручено как можно активнее выискивать возможности многократного использования кода. Рассмотрим, к примеру, ситуацию, показанную на рис. 16.3.

Рис. 16.3. Поиски возможностей многократного использования кода в сервис-ориентированной архитектуре

Глядя на рис. 16.3, архитектор понимает, что каждое из этих подразделений страховой компании использует понятие «Клиент». Следовательно, правильная

стратегия для сервис-ориентированной архитектуры не обойдется без объединения всего, что относится к клиентам, в многократно используемый канонический сервис работы с клиентами под названием `Customer` и разрешения исходным сервисам ссылаться на него, как показано на рис. 16.4.

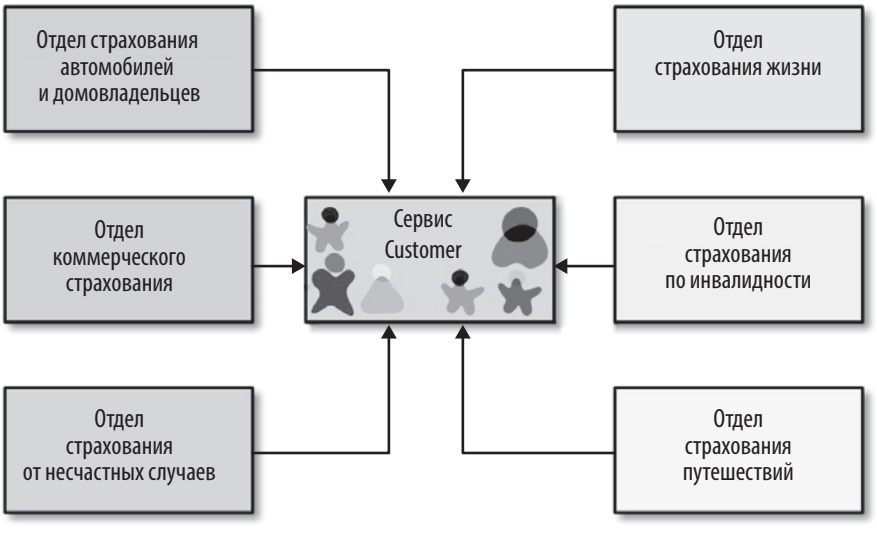


Рис. 16.4. Создание канонических представлений в сервис-ориентированной архитектуре

На рис. 16.4 показано, что архитектор выделил все, что связано с клиентами, в единый сервис `Customer`, добившись вполне очевидной цели его многократного использования.

Но архитекторы долго не могли осознать всех негативных сторон этого дизайна. Прежде всего, когда создается система, где основное внимание уделяется многократности использования, существует риск получить огромное количество связанных компонентов. Например, в системе на рис. 16.4 изменения, вносимые в сервис `Customer`, будут распространяться и на все остальные сервисы, что повлечет за собой высокую степень риска навредить всей системе. Из-за этого архитекторы испытывали большие трудности при внесении инкрементных изменений в сервис-ориентированную архитектуру: каждое изменение потенциально вызывало цепную реакцию. А это, в свою очередь, приводило к необходимости проведения скоординированных развертываний, комплексного тестирования и других факторов, снижающих эффективность разработки.

Для выявления еще одного негативного побочного эффекта рассмотрим случай автострахования и страхования от инвалидности в системе, показанной

на рис. 16.4. В единый сервис `Customer` должны быть включены все сведения о клиентах, известные организации. Автострахование требует сведений о водительских правах, являющихся принадлежностью человека, а не транспортного средства. Следовательно, сервис `Customer` должен включать сведения о водительских правах, которые отдел страхования по инвалидности совершенно не интересуют. И тем не менее команде, занимающейся проблемами страхования по инвалидности, приходится сталкиваться с лишними сложностями при обработке данных клиента.

Возможно, самым неприятным открытием, связанным с этой архитектурой, стало осознание непрактичности архитектуры, настолько сильно сосредоточенной на техническом разбиении. Хотя это казалось логичным для целей разбиения и многократного использования кода, на практике все превратилось в сплошной кошмар. Такие действия предметной области, как оформление заказа по каталогу (`CatalogCheckout`), были распределены по всей этой архитектуре настолько мелкими частями, что практически превратились в пыль. Разработчики часто сталкиваются с такими задачами, как «добавление новой адресной строки в `CatalogCheckout`». В сервис-ориентированной архитектуре в это могут быть вовлечены десятки сервисов на нескольких разных уровнях, а кроме того, это решение повлечет за собой внесение изменений в единую схему базы данных. Если же текущие корпоративные сервисы не были определены с корректной транзакционной гранулярностью, разработчикам придется либо менять дизайн, либо создавать новый, почти такой же сервис, чтобы изменить транзакционное поведение. Не слишком ли высокая плата за многократное использование?

Оценки архитектурных свойств

Многие современные критерии, которые мы используем для оценки, не относились к приоритетным во времена популярности рассматриваемой архитектуры. На самом деле тогда только началось развитие Agile-разработки, которая еще не применялась в крупных компаниях.

Одна звезда в оценочной таблице свойств (рис. 16.5) означает, что данное архитектурное свойство не имеет высокой поддержки в архитектуре, а пять звезд — что архитектурное свойство является одной из самых сильных сторон архитектурного стиля. Определение каждого свойства, указанного в оценочной таблице, можно найти в главе 4.

Архитектура, ориентированная на сервисы, является, пожалуй, самым ярким примером когда-либо созданных архитектур общего назначения с сугубо

Архитектурное свойство	Оценка в звездах
Тип разбиения	Технический
Количество квантов	1
Развертываемость	★
Адаптируемость	★★★
Эволюционность	★
Отказоустойчивость	★★★
Модульность	★★★
Общие затраты	★
Производительность	★★
Надежность	★★
Масштабируемость	★★★★
Простота	★
Тестируемость	★

Рис. 16.5. Оценки свойств сервис-ориентированной архитектуры

техническим разбиением. Фактически именно негативная реакция на недостатки этой структуры привела к появлению таких более современных архитектур, как микросервисы. В данной архитектуре, хотя она и считается распределенной, имеется всего один квант, и на это есть две причины. Во-первых, в ней обычно используется одна база данных или несколько баз данных с созданием в архитектуре точек сопряжения для решения множества различных задач. Во-вторых, что более важно, механизм оркестрации действует как огромная точка сопряжения: ни одна из имеющихся частей архитектуры не может иметь архитектурные свойства, отличающиеся от свойств посредника, управляющего всем поведением. Получается, что в этой архитектуре можно найти недостатки, присущие как монолитной, так и распределенной архитектуре.

Уровень достижения современных инженерных целей, таких как развертываемость и тестируемость, оценивается в этой архитектуре крайне низко, и это не

только из-за из плохой поддержки, но и из-за того, что в то время им не придавалось важного или даже просто желаемого значения.

В этой архитектуре поддерживались, например, высокая адаптируемость и масштабируемость, несмотря на трудности в реализации. Это стало возможным благодаря неимоверным усилиям производителей, которые применили репликацию сеансов между серверами приложений и ряд других методов. Но из-за распределенного характера данной архитектуры производительность никогда не была ее сильной стороной и оценивалась крайне низко, поскольку каждый бизнес-запрос распределялся по большей части архитектуры.

По причине всех изложенных здесь факторов простота и общие затраты совершенно не отвечают предпочтениям большинства архитекторов. И все же эта архитектура стала важной исторической вехой, поскольку показала, насколько сложными могут быть распределенные транзакции в реальной жизни, а также практические ограничения технического разбиения.

ГЛАВА 17

Архитектура микросервисов

Микросервисы — это самый популярный архитектурный стиль, который в последние годы получил широкое распространение. В этой главе будут представлены важные свойства, отличающие эту архитектуру в топологическом и философском плане.

История

Большинство архитектурных стилей получили свои названия уже после того, как были замечены определенные закономерности в архитектуре. Не существует тайного сообщества, которое решает, каким будет следующее популярное направление, скорее многие решения принимаются по мере изменения экосистемы разработки программных средств. Самые эффективные решения задач, возникшие в ходе этих изменений, и полученные результаты становятся архитектурными стилями.

В этом плане микросервисы отличаются: свое название они получили довольно рано и приобрели известность благодаря появившейся в марте 2014 года знаменитой публикации «Microservices»[1] в блоге Мартина Фаулера (Martin Fowler) и Джеймса Льюиса (James Lewis). Авторы выявили множество общих свойств этого относительно нового архитектурного стиля и описали их. Их публикация помогла разобраться в данной архитектуре и понять заложенную в нее философию.

В основу микросервисов были положены идеи предметно-ориентированного проектирования (domain-driven design, DDD) — логического процесса проек-

[1] https://martinfowler.com/articles/microservices.html

тирования программных проектов. В частности, ощутимо ускорила появление микросервисов одна из концепций DDD — *ограниченный контекст (bounded context)*. Эта концепция представляет собой стиль разделения, или развязывания (decoupling). Когда разработчик определяет предметную область, она включает в себя множество сущностей и особенностей поведения, проявляющихся в таких артефактах, как программный код и схемы баз данных. К примеру, у приложения может быть предметная область под названием `CatalogCheckout`, включающая такие понятия, как позиции каталога, клиенты и платежи. В традиционной монолитной архитектуре разработчики совместно применяли бы многие из этих понятий для создания многократно используемых классов и связанных баз данных. Внутри ограниченного контекста такие части, как программный код и схемы базы данных, связаны между собой для выполнения работы, но за его пределами они не связаны ни с базой данных, ни с определением класса из другого ограниченного контекста. Это позволяет каждому контексту определять только то, что ему нужно, без учета других составляющих.

При всех плюсах повторного использования кода не следует забывать Первый закон архитектуры программного обеспечения, касающийся компромиссов. Негативным последствием многократного использования кода является связанность. Когда архитектор проектирует систему, в которой отдается предпочтение многократному использованию кода, для достижения этой многократности ему приходится мириться со связанностью, либо в виде наследования, либо в виде композиции.

Но если целью архитектора является высокая степень развязывания, он отдает предпочтение дублированию кода, а не многократности его использования. Основной целью создания микросервисов является высокая степень развязывания, то есть физическое воплощение логического понятия ограниченного контекста.

Топология

Топология микросервисов представлена на рис. 17.1.

Как показано на рис. 17.1, из-за своей узкоспециализированной природы размеры сервисов в микросервисах намного меньше, чем в других распределенных архитектурах, таких как, например, оркестрированная сервис-ориентированная архитектура. Архитекторы исходят из того, что каждый сервис включает все необходимое для независимой работы: базы данных и другие зависимые компоненты. Свойства, отличающие архитектуру микросервисов, рассматриваются в следующих разделах.

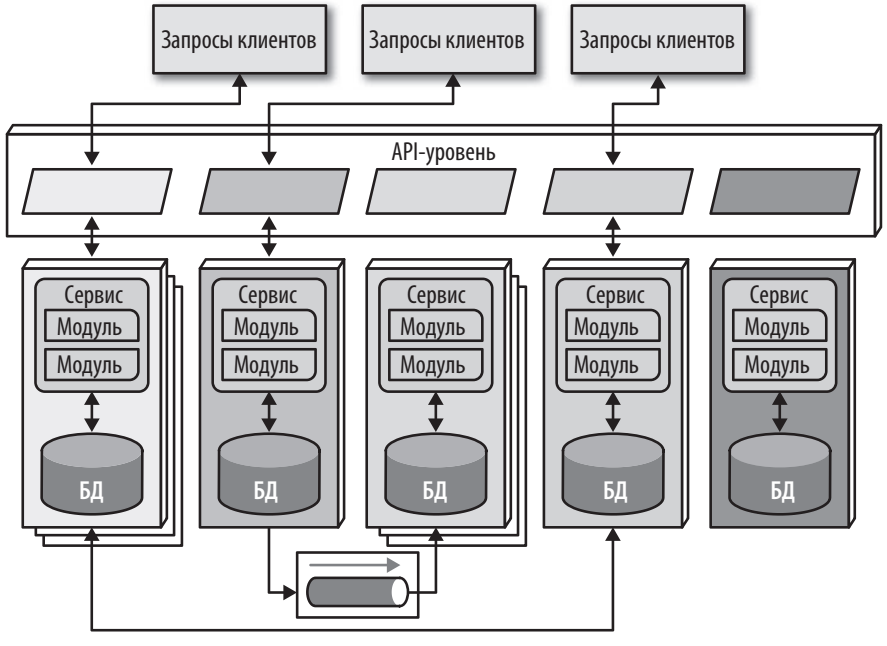


Рис. 17.1. Топология архитектурного стиля микросервисов

Распределенность

Микросервисы формируют *распределенную архитектуру*: каждый сервис запускается в своем собственном процессе, в качестве которого сначала был физический компьютер, вскоре эволюционировавший до виртуальных машин и контейнеров. С помощью такой развязки сервисов решились многие распространенные архитектурные проблемы многопользовательской инфраструктуры поддержки приложений. Например, когда несколькими запущенными приложениями управляет один сервер, вместе с другими преимуществами появляется возможность повторного использования пропускной способности сети, памяти, дискового пространства. Но если все поддерживаемые приложения продолжат разрастаться, то со временем ресурсы общей инфраструктуры будут исчерпаны. Еще одной проблемой может стать некорректная изолированность совместно используемых приложений.

Выделение каждого сервиса в отдельный процесс позволяет решать все проблемы, возникающие при совместном использовании оборудования. До появления свободно распространяемых операционных систем с открытым кодом с автоматическим выделением ресурсов было непрактично выделять каждой

предметной области свою собственную инфраструктуру. Но теперь, благодаря облачным хранилищам и контейнерной технологии, команды разработчиков могут пользоваться преимуществами полной развязки как на уровне предметной области, так и на эксплуатационном уровне.

Распределенность микросервисов часто негативно отражается на общей производительности. Сетевые вызовы гораздо продолжительнее вызовов методов; кроме того, в каждом эндпоинте приходится дополнительно тратить время на проверку безопасности, что требует от архитекторов особо тщательного подхода к гранулярности при проектировании системы.

Поскольку микросервисы относятся к распределенной архитектуре, опытные архитекторы не рекомендуют использовать транзакции, пересекающие границы сервисов, что делает определение гранулярности сервисов ключом к успеху данной архитектуры.

Ограниченный контекст

В основе философии микросервисов лежит понятие *ограниченного контекста*: в каждом сервисе моделируется предметная область или рабочий процесс. Таким образом, каждый сервис включает в себя все необходимое для работы в составе приложения, в том числе классы, другие подчиненные компоненты и схемы базы данных. Эта философия определяет многие решения, принимаемые архитекторами в рамках данной архитектуры. К примеру, в монолитной системе разработчики практикуют совместное использование общих классов, таких как `Address`, между разделенными частями приложения. А в микросервисах стараются избегать связанности, поэтому архитектор, выстраивающий этот архитектурный стиль, предпочитает дублирование кода.

В микросервисах концепция предметного разбиения доведена до абсолюта. Каждый сервис предназначен для представления предметной области или ее части; во многих отношениях микросервисы являются физическим воплощением логической концепции предметно-ориентированного проектирования.

Гранулярность

В микросервисах архитекторы стремятся добиться приемлемого уровня детализации сервисов и часто допускают ошибку, делая их слишком мелкими, что вынуждает возвращаться к выстраиванию связей между ними для нормальной работы.

> Понятие «микросервис» — это всего лишь *название*, а не *описание*.
>
> *Мартин Фаулер*

Иными словами, тем, кто придумал это понятие, просто нужно было *как-то* назвать этот новый стиль, и они выбрали термин «микросервисы», чтобы противопоставить его доминирующему на тот момент сервис-ориентированному архитектурному стилю, который мог бы получить название «макросервисы». Но многие разработчики понятие «микросервисы» воспринимают не как описание, а как указание создавать слишком мелкие сервисы.

Границы сервиса в микросервисах предназначены для охвата предметной области или рабочего процесса. В ряде приложений для некоторых частей системы эти естественные границы могут быть шире других из-за бизнес-процессов с более тесной связанностью. Найти приемлемые границы можно с помощью следующих рекомендаций:

Назначение

Самые очевидные границы определяются предметной областью. В идеале каждый микросервис должен обладать абсолютной функциональной связностью (cohesion), обеспечивая только одну важную для всего приложения функциональность.

Транзакции

Ограниченные контексты являются рабочими бизнес-процессами, и зачастую правильная граница сервиса зависит от тех сущностей, которым требуется взаимодействие в рамках транзакции. Поскольку транзакции в распределенных архитектурах связаны с определенными проблемами, архитекторы, способные спроектировать свои системы без их использования, создают более качественный дизайн.

Хореография

Если архитектор формирует набор сервисов, которые обеспечивают предметным областям абсолютную изолированность, но при этом требуется их постоянное взаимодействие друг с другом для корректной работы, то для снижения коммуникационных накладных расходов стоит объединить эти сервисы в один более крупный сервис.

Единственным способом обеспечения приемлемого дизайна сервисов является итеративное приближение к идеалу. Архитекторам довольно редко удается с первого раза определить хороший уровень детализации, зависимость данных

и стили обмена данными. Но после ряда итераций архитектор с большой вероятностью сможет улучшить свой дизайн.

Изолированность данных

Еще одним требованием к микросервисам, определяемым концепцией ограниченного контекста, является изолированность данных. Многие архитектурные стили обращаются к единой базе данных для хранения информации. Однако в микросервисах стараются избегать любых проявлений связанности, в том числе общих схем и баз данных, использующихся в качестве точек интеграции.

Изолированность данных является еще одним фактором, который архитектор учитывает при выборе гранулярности. Здесь следует избегать попадания в западню сущности (рассмотренную в соответствующем подразделе в главе 8 на с. 146), а не просто моделировать свои сервисы как отдельные сущности в базе данных.

Архитекторы обычно используют реляционные базы данных для унификации значений в системе, создавая единый источник истины (single source of truth, SSOT). Но это невозможно при распределении данных по всей архитектуре. Поэтому архитекторы должны выбрать другой вариант решения этой задачи: либо считать одну предметную область источником истины и обращаться к ней для получения значений, либо воспользоваться репликацией базы данных или кэшированием для распределения информации.

Хотя такой уровень изоляции данных существенно усложняет работу, он также раскрывает и новые возможности. Теперь, когда команды разработчиков не обязаны придерживаться единой базы данных, для каждого сервиса может быть выбран наиболее подходящий инструмент в зависимости от затрат, типа хранилища или множества других факторов. В системе с высокой степенью разделенности команды пользуются возможностью выбирать наиболее подходящую базу данных (или другую зависимость), не влияя на работу других команд, которым не разрешено вникать в подробности реализации.

API-уровень

Большинство изображений микросервисов содержит API-уровень, находящийся между потребителями системы (либо пользовательскими интерфейсами, либо вызовами, поступающими от других систем), но его наличие необяза-

тельно. Он довольно популярен, потому что на этом уровне могут выполняться важные задачи либо посредством его косвенного использования в качестве прокси-сервера, либо через связь с такими эксплуатационными средствами, как сервис именования (см. раздел «Многократное использование в эксплуатации» на с. 298).

API-уровень может пригодиться для решения множества задач, но его не следует использовать в качестве медиатора или оркестратора, если архитектор намерен придерживаться основной философии микросервисной архитектуры: вся необходимая архитектурная логика должна находиться в ограниченном контексте, а размещение оркестровки или иной логики в медиаторе нарушает это правило. Это также показывает разницу между техническим и предметным разбиением в архитектуре: обычно медиаторы используются в архитектурах, разбиваемых по техническому принципу, а в микросервисах всегда только предметное разбиение.

Многократное использование в эксплуатации

Если в микросервисах применяется дублирование, а не связанность кода, то что делать с теми частями архитектуры, которые действительно выигрывают от связанности, — мониторингом, журналированием и автоматическими выключателями (circuit breakers)? Одной из основных идей традиционной сервис-ориентированной архитектуры было многократное использование максимально возможного количества функциональных средств, как предметной области, так и эксплуатационных. В микросервисах архитекторы стараются отделить эти два направления друг от друга.

Разработав несколько микросервисов, команда понимает, что у каждого из них есть общие элементы, но в таком повторении есть свои плюсы. Например, если организация позволяет каждой команде разработчиков сервисов вести мониторинг самостоятельно, то как контролировать работу таких команд? Как будет решена проблема обновлений инструмента мониторинга? Будет ли управление обновлениями обязанностью каждой команды и сколько это будет занимать времени?

Решить эту задачу предлагается за счет применения паттерна *sidecar*, показанного на рис. 17.2.

На рис. 17.2 общие эксплуатационные задачи показаны внутри каждого сервиса в виде отдельного компонента, которым могут владеть как отдельные команды разработчиков, так и команда общей инфраструктуры. Sidecar-компонент управляется со всеми эксплуатационными задачами, что удобно для совместной рабо-

ты команд. Таким образом, как только приходит время обновить инструменты мониторинга, команда общей инфраструктуры обновляет sidecar-компоненты, и новые функциональные возможности становятся доступны каждому микросервису.

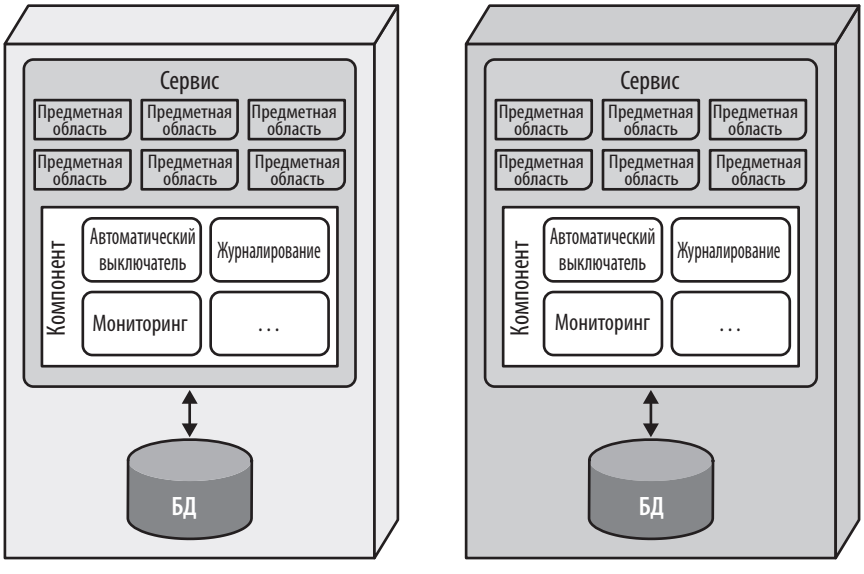


Рис. 17.2. Паттерн sidecar в микросервисах

Поскольку команды знают, что каждый сервис включает в себя общий sidecar-компонент, они могут создать *сервисную сетку (service mesh)*, позволяющую обеспечить единый контроль над архитектурой для таких задач, как журналирование и мониторинг. Общие sidecar-компоненты соединяются для формирования единого эксплуатационного интерфейса для всех микросервисов (рис. 17.3).

Каждый sidecar-компонент, показанный на рис. 17.3, подключен к панели обслуживания, формирующей согласованный интерфейс для каждого сервиса.

Сама сервисная сетка образует консоль, предоставляющую разработчикам единый доступ к сервисам (рис. 17.4).

Как показано на рис. 17.4, каждый сервис образует узел в общей сетке. Сервисная сетка формирует консоль, позволяющую командам глобально управлять такими функциями, как мониторинг, журналирование, и другими сквозными эксплуатационными задачами.

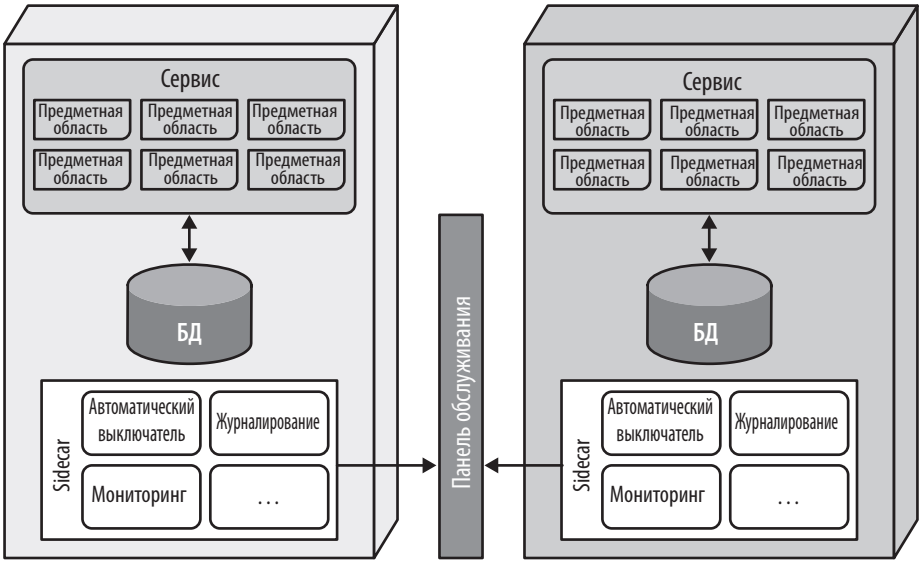


Рис. 17.3. Панель обслуживания, соединяющая sidecar-компоненты в сервисную сетку

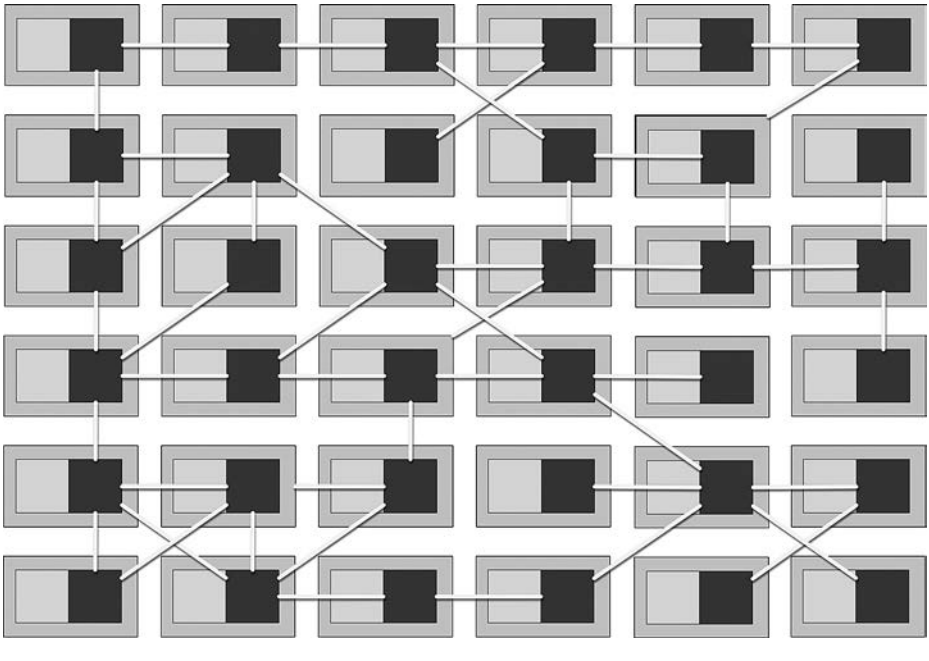


Рис. 17.4. Сервисная сетка дает единое представление эксплуатационных аспектов микросервисов

Для повышения гибкости микросервисов архитекторы используют инструменты *обнаружения сервисов* (*service discovery*). Вместо того чтобы сразу вызвать конкретный сервис, запрос проходит через инструмент обнаружения сервисов, который может отследить количество и частоту запросов, а также запустить новые экземпляры сервисов, чтобы справиться с задачами масштабирования или достичь нужной гибкости. Архитекторы часто включают этот инструмент в сервисную сетку, делая его частью каждого микросервиса. API-уровень часто используется для размещения инструмента обнаружения сервисов, предоставляя единое место для пользовательских интерфейсов или других систем вызова с целью гибкого и согласованного поиска и добавления сервисов.

Клиентские стороны приложения (фронтенды)

Предпочтение в микросервисах отдается развязыванию, распространяющемуся в идеале на пользовательские интерфейсы (UI), а также на задачи серверной части (бэкенда). Фактически в первоначальном варианте микросервисов, в соответствии с принципами DDD, пользовательский интерфейс был частью ограниченного контекста. Но практические аспекты разбиения, требуемые для веб-приложений, а таже другие внешние ограничения создавали трудности в достижении этой цели. Поэтому в микросервисах обычно используются два стиля пользовательских интерфейсов, первый из которых показан на рис. 17.5.

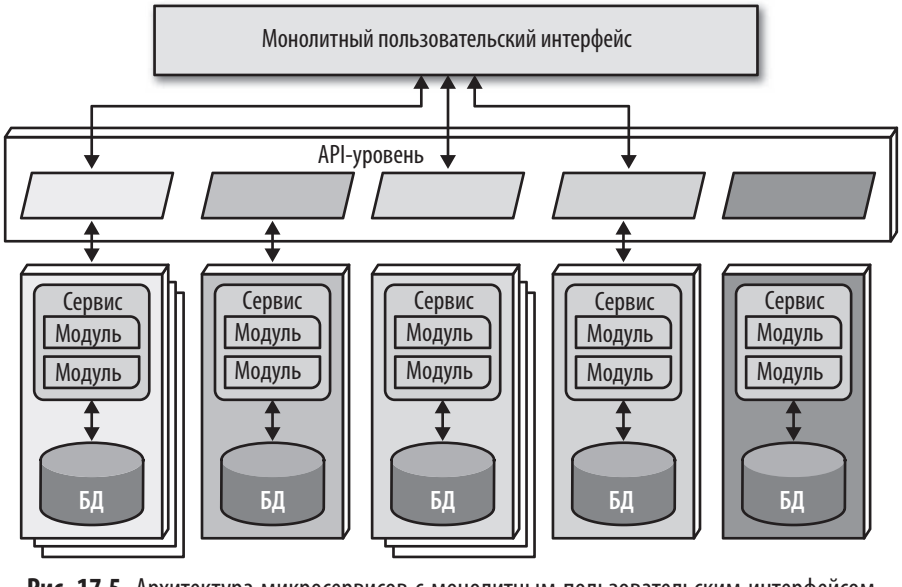


Рис. 17.5. Архитектура микросервисов с монолитным пользовательским интерфейсом

Показанный на рис. 17.5 монолитный фронтенд представлен единым UI, из которого все вызовы для обслуживания пользовательских запросов проходят через API-уровень. Фронтенд может представлять собой полноценный настольный компьютер, мобильное или веб-приложение. К примеру, во многих современных веб-приложениях для выстраивания единого UI используется веб-фреймворк на языке JavaScript.

Во втором стиле пользовательских интерфейсов используются *микрофронтенды* (*microfrontends*), показанные на рис. 17.6.

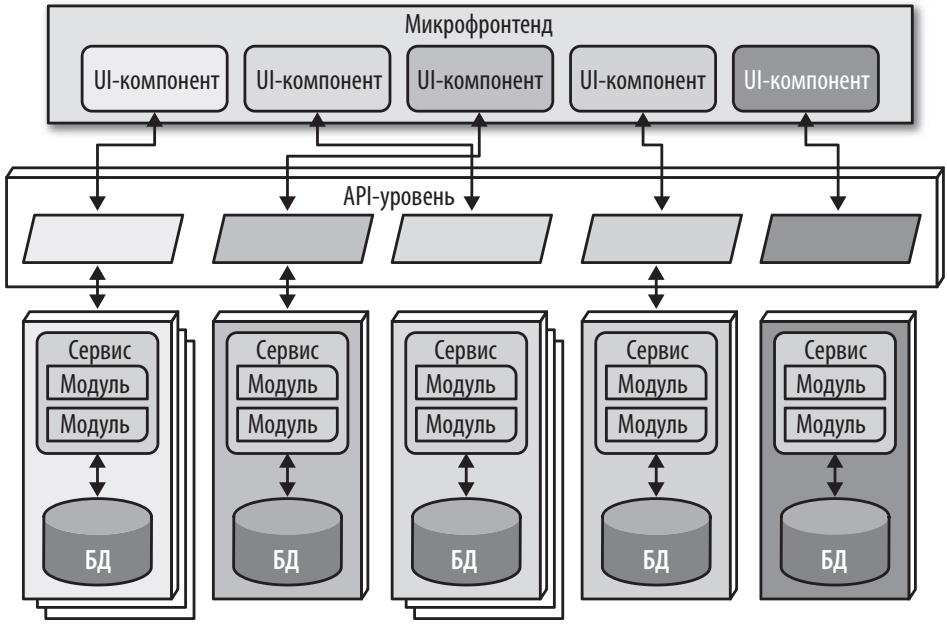


Рис. 17.6. Паттерн микрофронтенда в микросервисах

На рис. 17.6 показано, как компоненты на уровне UI используются в качестве бэкенд-сервисов для создания синхронного уровня детализации и изоляции в пользовательском интерфейсе. Каждый сервис сам для себя эмитирует пользовательский интерфейс, а фронтенд согласовывает его с другими эмитируемыми UI-компонентами. Используя такой паттерн, команды разработчиков могут изолировать границы сервисов от пользовательского интерфейса до бэкенд-сервисов, объединяя всю предметную область для одной команды.

Разработчики могут реализовать паттерн микрофронтенда различными способами: либо с использованием React — веб-фреймворка, основанного на компо-

нентах, либо с помощью одного из многих фреймворков с открытым исходным кодом, поддерживающих этот паттерн.

Обмен данными

В микросервисах архитекторы и разработчики сталкиваются с проблемой выбора подходящей гранулярности, которая непосредственно влияет как на обмен данными, так и на изоляцию данных. Подбор правильного стиля обмена данными помогает командам поддерживать разделенность сервисов, сохраняя при этом согласованность их работы.

По сути, архитекторы должны выбирать между *синхронным* и *асинхронным* способом обмена данными. Синхронный обмен требует, чтобы вызывающий ждал ответа от вызываемого. Архитектуры микросервисов обычно используют *разнородную интероперабельность на основе протокола* (*protocol-aware heterogeneous interoperability*). Давайте разобьем это понятие на составные части:

На основе протокола (protocol-aware)

Поскольку обычно в микросервисах, чтобы избежать операционной связанности, нет никакого единого центра интеграции, каждый сервис должен знать, как можно вызывать другие сервисы. Для организации таких вызовов архитекторы обычно придерживаются определенного стандарта: конкретный тип REST-обмена, очереди сообщений и т. д. Следовательно, сервисам должно быть известно (или они должны уметь обнаруживать), каким именно протоколом нужно пользоваться для вызова других сервисов.

Разнородная (heterogeneous)

Поскольку микросервисы относятся к распределенной архитектуре, каждый сервис может быть написан в рамках того или иного технологического стека. *Разнородность* предполагает, что микросервисы полностью поддерживают многоязычные среды, где разные сервисы используют разные платформы.

Интероперабельност (interoperability)

Это означает, что сервисы вызывают друг друга. Хотя архитекторы стараются не допускать в микросервисах транзакционных вызовов методов, сервисы обычно вызывают другие сервисы по сети для совместной работы и обмена информацией.

Для асинхронного обмена данными архитекторы часто используют события и сообщения, поэтому для внутренних целей применяется архитектура, управляемая

событиями, о которой говорилось в главе 14, а брокер и медиатор в микросервисах работают как *хореография* и *оркестровка*.

ВЫНУЖДЕННАЯ РАЗНОРОДНОСТЬ

Известный архитектор, родоначальник стиля микросервисов, был главным архитектором стартапа по управлению персональной информацией в мобильных устройствах. Поскольку дело касалось решения задач в весьма динамичной предметной области, он хотел убедиться, что ни одна из команд разработчиков случайным образом не создаст никаких точек связанности друг с другом, ограничивающих независимость их действий. Оказалось, что команды у этого архитектора обладали широким набором технических навыков, поэтому каждая должна была использовать свой собственный стек технологий. Если одна команда работала на Java, а другая применяла .NET-технология, то абсолютно исключалось случайное совместное использование каких-либо классов.

Этот подход стал полной противоположностью большинству стратегий корпоративного управления, настаивающих на использовании единого технологического стека. В мире микросервисов цель состоит не в создании максимально сложной экосистемы, а в выборе правильной технологии масштабирования для решения узкого круга задач. В мощной реляционной базе данных нуждается далеко не каждый сервис, и принуждение небольших команд к ее использованию вредит их работе, замедляя весь процесс. Эта концепция еще больше усиливает разделенность микросервисов.

Хореография и оркестровка

Хореография применяет тот же стиль обмена данными, что и брокер в архитектуре, управляемой событиями. Иначе говоря, благодаря соблюдению философии ограниченного контекста, в ней нет центрального координатора. Таким образом, для архитекторов будет вполне приемлемой реализация несвязанных событий между сервисами.

Изоморфизм предметной области и архитектуры — одно из ключевых свойств, на которое архитекторам следует обратить внимание, оценивая, подходит ли стиль архитектуры для решения конкретной задачи. Данный термин описывает соотношение формы архитектуры и конкретного архитектурного стиля. Например, на рис. 8.7 архитектура проекта Silicon Sandwiches разбита по техническому

принципу и структурно поддерживает настраиваемость, а стиль микроядерной архитектуры предлагает точно такую же общую структуру. Следовательно, задачи, требующие высокого уровня настраиваемости, легче реализовать в микроядре.

Аналогичным образом, поскольку цель архитектора в микросервисной архитектуре — разделенность, то микросервисы похожи на брокера из архитектуры, управляемой событиями, что вполне допускает симбиоз этих двух паттернов.

В хореографии каждый сервис вызывает другие сервисы по мере необходимости, без центрального медиатора. Рассмотрим, к примеру, сценарий, показанный на рис. 17.7.

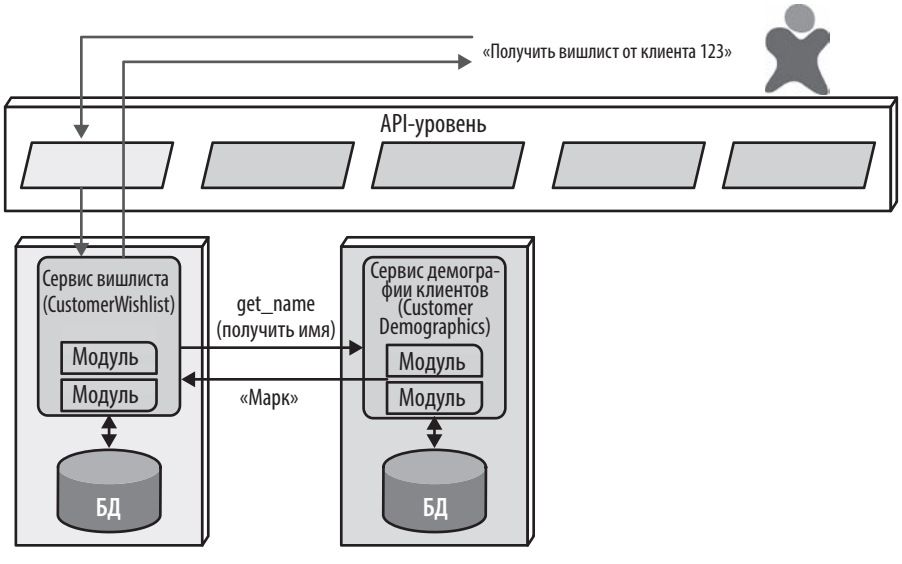


Рис. 17.7. Применение в микросервисах хореографии для управления координацией

На рис. 17.7 показано, что пользователь запросил информацию из вишлиста клиента. Поскольку сервис `CustomerWishList` не содержит всей необходимой информации, он обращается к сервису `CustomerDemographics` для получения недостающих данных и возвращает результат пользователю.

Так как архитектура микросервисов не содержит глобального медиатора, как в других сервис-ориентированных архитектурах, то при необходимости координации работы нескольких сервисов архитектор может создать свой собственный локализованный медиатор (рис. 17.8).

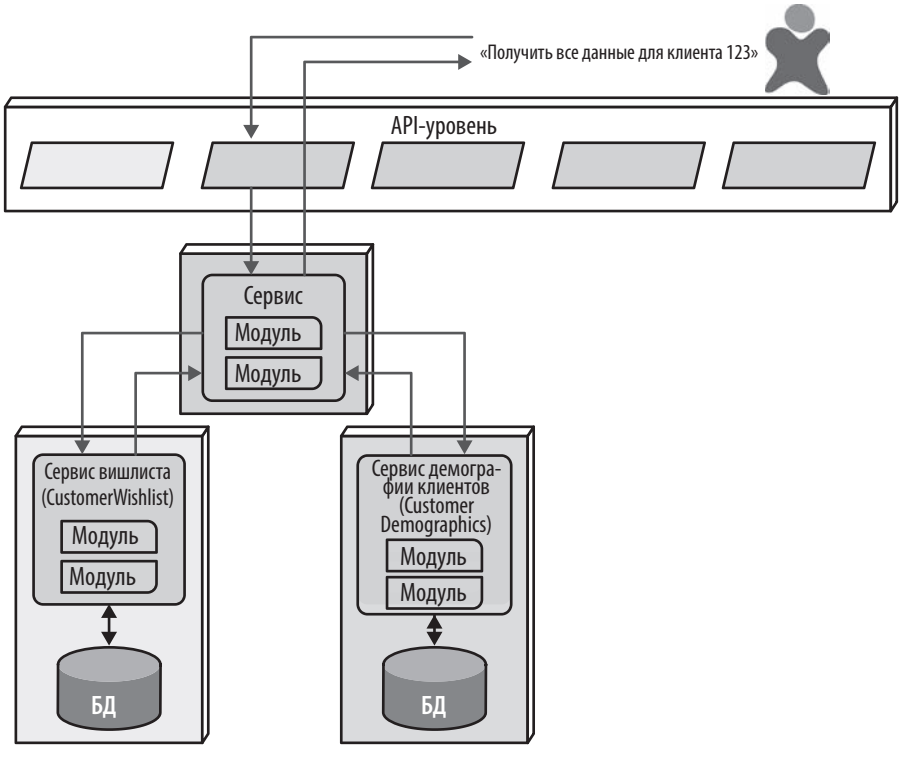


Рис. 17.8. Использование оркестровки в микросервисах

На рис. 17.8 показано, что разработчики создали сервис, единственной задачей которого является координация вызова для получения информации о конкретном клиенте. Пользователь вызывает медиатор `ReportCustomerInformation`, который запускает другие необходимые сервисы.

Согласно Первому закону архитектуры программного обеспечения, ни одно из этих решений не является идеальным, у каждого есть свои компромиссы. В хореографии архитектор придерживается философии архитектуры с высокой степенью разделенности, получая от этого стиля максимальную выгоду. Но решение стандартных задач, таких как обработка ошибок и координация, в средах с хореографией существенно усложняется.

Рассмотрим пример с более сложным рабочим процессом, показанным на рис. 17.9.

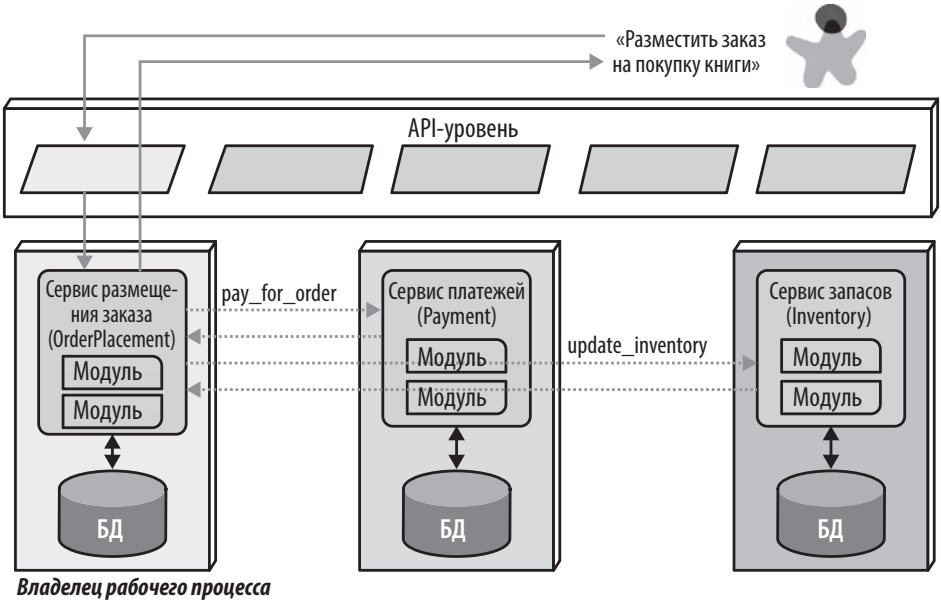


Рис. 17.9. Использование хореографии для сложного бизнес-процесса

На рис. 17.9 показано, что первый вызванный сервис должен согласовывать работу широкого круга других сервисов, не только выполняя свои предметные обязанности, но и действуя в качестве медиатора. Эта схема называется паттерном *фронт-контроллера* (*front controller pattern*), в котором условный хореографический сервис становится более сложным медиатором для решения той или иной задачи. Недостатком этого паттерна является усложнение сервиса.

Как вариант, для сложных бизнес-процессов архитектор может выбрать оркестровку, показанную на рис. 17.10.

На рис. 17.10 показано, что архитектор создает медиатор, чтобы справиться со сложностями и координацией в бизнес-процессе. Несмотря на возникновение связанности, такое решение позволяет архитектору сосредоточить координацию в одном сервисе, сократив ее влияние на все остальные. Во многих случаях рабочие процессы предметной области неизбежно связаны друг с другом, и цель архитектора — найти наилучший способ представления этой связанности, чтобы решать и архитектурные задачи, и задачи предметной области.

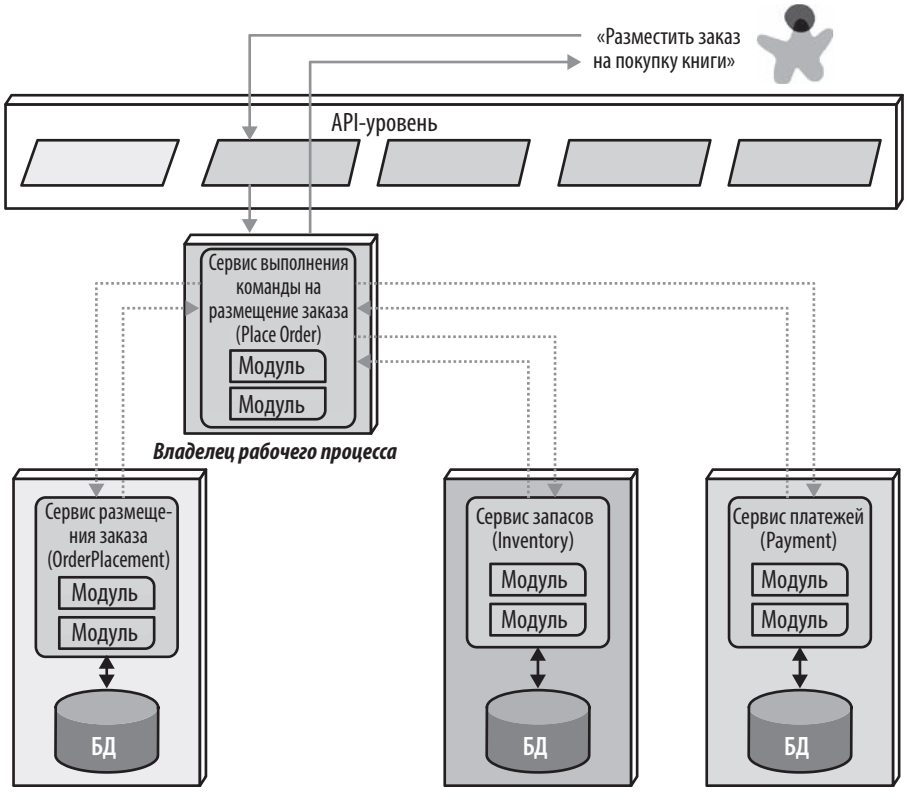


Рис. 17.10. Использование оркестровки для сложного бизнес-процесса

Транзакции и саги

Архитекторы стремятся к полной разделенности в микросервисах, но затем довольно часто сталкиваются с проблемой обеспечения координации транзакций между сервисами. Разделенность в архитектуре должна относиться также и к базам данных, поэтому атомарность, которая легко обеспечивается в монолитных приложениях, становится проблемой в распределенных.

Выстраивание транзакций, пересекающих границы сервисов, нарушает основной принцип разделенности в архитектуре микросервисов (а также создает наихудший вид динамической коннасценции, а именно коннасценцию значений). Лучшим советом для архитекторов, желающих совершать транзакции между сервисами, будет следующий: *не делайте этого!* Лучше скорректируйте гранулярность компонентов. Нередко архитекторы, выстраивающие архитектуры микросервисов, со временем обнаруживают необходимость связать их вместе

с помощью транзакций — это происходит, если компоненты в их дизайне слишком мелко гранулированы. Границы транзакций — один из лучших индикаторов гранулярности сервисов.

Не используйте транзакции в микросервисах, лучше скорректируйте уровень гранулярности!

И тем не менее без исключений не обойтись. Например, может возникнуть ситуация, когда двум отдельным сервисам с разными архитектурными свойствами и отчетливыми границами все-таки понадобится координация через транзакции. Для таких случаев существуют паттерны оркестровки транзакций, но с серьезными компромиссами.

В микросервисах популярным паттерном транзакций является паттерн *саги (saga pattern)*, показанный на рис. 17.11.

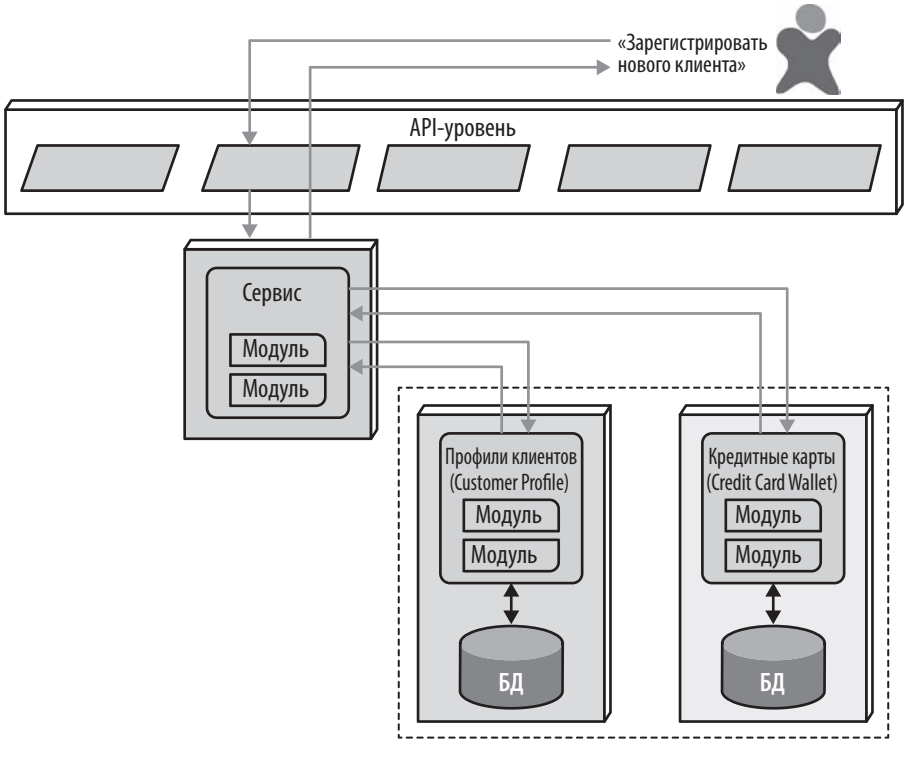


Рис. 17.11. Паттерн саги в архитектуре микросервисов

Показанный на рис. 17.11 сервис работает в качестве медиатора нескольких сервисных вызовов и координирует транзакцию. Медиатор вызывает каждую часть транзакции, фиксирует успех или сбой и координирует результаты. Если все идет по плану, все значения в сервисах и в содержащихся в них базах данных обновляются синхронно.

При возникновении ошибки медиатор должен гарантировать, что при сбое хотя бы одной части ни одна из других частей транзакции не будет считаться успешно проведенной. Рассмотрим ситуацию, показанную на рис. 17.12.

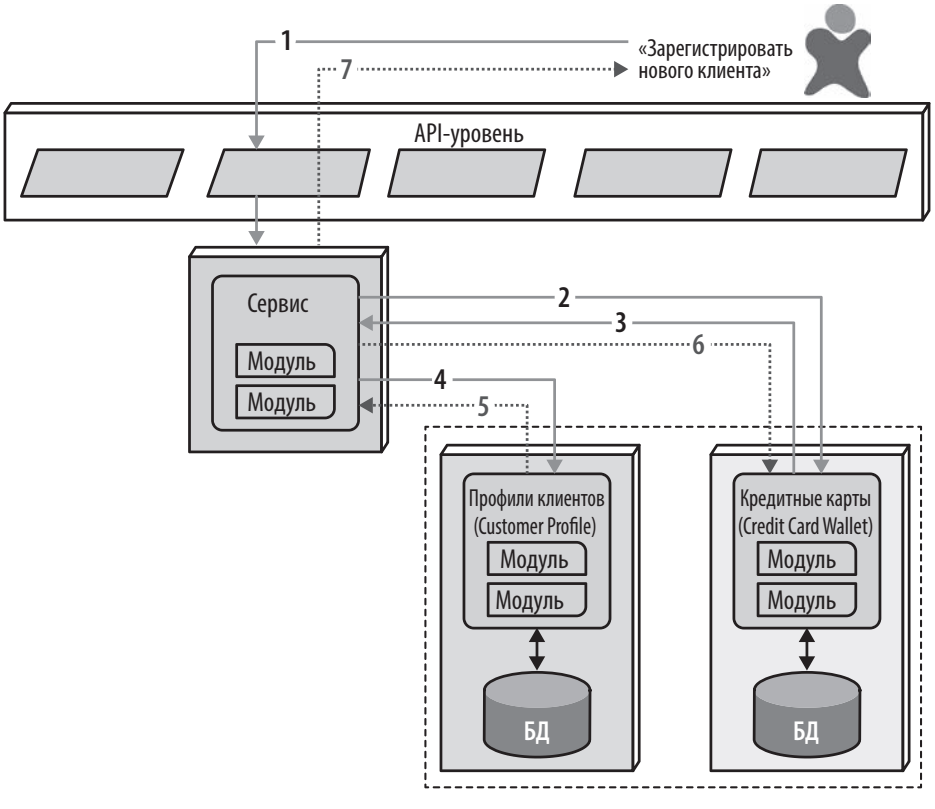


Рис. 17.12. Паттерн саги, отменяющий транзакции при возникновении ошибки

Если первая часть показанной на рис. 17.12 транзакции завершается успешно, а вторая дает сбой, медиатор должен отправить запрос ко всем частям транзакции, завершившимся успешно, с предписанием отмены предыдущего запроса. Такой стиль транзакционной координации называется *фреймворком отменяемых транзакций (compensating transaction framework)*. В этом паттерне каждый

запрос от медиатора переходит в состояние ожидания, пока медиатор не сообщит о положительном результате. Однако такой дизайн сильно усложняется, когда появляется необходимость обрабатывать асинхронные запросы, особенно при появлении новых запросов, зависящих от ожидающего состояния транзакции. Кроме того, создается большой трафик для координации на сетевом уровне.

Еще одна реализация фреймворка отменяемых транзакций заставляет разработчиков выстраивать *прохождения* и *откаты (do* и *undo)* для каждой потенциально транзакционной операции. При этом усилия на координацию в ходе транзакций снижаются, но операции *отката* порой оказываются существенно сложнее операций *прохождения*, что практически удваивает объем работы по проектированию, реализации и отладке.

Хотя у архитекторов есть возможность выстраивания транзакций между сервисами, это противоречит самой идее микросервисов. Разумеется, есть и исключения, поэтому лучшим советом архитекторам будет умеренное использование паттерна саги.

Порой без небольшого числа транзакций между сервисами просто не обойтись. Но если они становятся доминантой архитектуры, значит, с ней что-то не в порядке!

Оценки архитектурных свойств

Стиль архитектуры микросервисов в нашей стандартной шкале оценок, показанной на рис. 17.13, демонстрирует ряд крайних значений. Одна звезда означает, что данное архитектурное свойство не имеет высокой поддержки в архитектуре, а пять звезд — что архитектурное свойство является одной из самых сильных сторон архитектурного стиля. Определение каждого свойства, указанного в оценочной таблице, можно найти в главе 4.

В оценках, показанных на рис. 17.13, примечательна высокая поддержка современных технологий разработки ПО, таких как автоматическое развертывание, тестируемость и т. д. Микросервисы не могли бы существовать без DevOps-революции и упорного стремления к автоматизации эксплуатационных задач.

Поскольку микросервисы относятся к распределенным архитектурам, им свойственны многие недостатки, присущие архитектурам, которые состоят из частей, объединяемых в процессе выполнения приложения. От избытка межсервисного обмена данными страдают отказоустойчивость и надежность. Но эти оценки указывают только лишь на тенденции в архитектуре; разработчики избавляются от множества подобных проблем за счет избыточности и масштабирования с приме-

нением инструментов обнаружения сервисов. В обычных условиях независимые одноцелевые сервисы, как правило, обеспечивают высокую отказоустойчивость, отсюда и высокая оценка этого свойства в архитектуре микросервисов.

Архитектурное свойство	Оценка в звездах
Тип разбиения	Предметный
Количество квантов	От одного и более
Развертываемость	★★★★
Адаптируемость	★★★★★
Эволюционность	★★★★★
Отказоустойчивость	★★★★
Модульность	★★★★★
Общие затраты	★
Производительность	★★
Надежность	★★★★
Масштабируемость	★★★★★
Простота	★
Тестируемость	★★★★

Рис. 17.13. Оценки микросервисов

Отличительными чертами рассматриваемой архитектуры являются масштабируемость, адаптируемость и эволюционность. Этот стиль уже использовался в целом ряде реализованных на практике высокомасштабируемых систем. Аналогично сказанному выше, поскольку архитектура в значительной степени полагается на автоматизацию и интеллектуальную интеграцию с операциями, разработчики также могут встроить в архитектуру поддержку высокой адаптируемости. Данный стиль способствует достижению высокой степени разделения на инкрементальном уровне, поэтому он поддерживает современную бизнес-практику эволюционных изменений даже на уровне архитектуры. Современный бизнес развивается стремительно, и разработка программных средств старается

не отставать от него. Создание архитектуры с предельно малыми и сильно разделенными модулями развертывания позволяет архитекторам получить структуру, способную поддерживать более высокую скорость изменений.

В микросервисах часто приходится сталкиваться с проблемой производительности: для корректной работы распределенные архитектуры должны выполнять массу сетевых вызовов с высокими накладными расходами производительности. Кроме того, они должны проводить в каждом эндпоинте проверку безопасности с подтверждением идентичности и прав доступа. Для повышения производительности в мире микросервисов существует множество паттернов, включая интеллектуальное кэширование и репликацию данных для предотвращения излишнего количества сетевых вызовов. Производительность выступает в микросервисах еще одним аргументом в пользу более частого применения хореографии, а не оркестровки, поскольку меньший объем взаимодействий обеспечивает более быстрый обмен данными и создает меньшее количество узких мест.

Микросервисы бесспорно относятся к предметно-ориентированной архитектуре, где все границы сервисов должны соответствовать предметным областям. По сравнению с любыми другими современными архитектурами микросервисы обладают наиболее выраженными квантами, являясь эталоном того, что оценивают квантовые показатели. Философия полной разделенности может привнести много проблем, но способна дать и огромные преимущества, если все сделано правильно. Как и в любой другой архитектуре, архитекторы должны усвоить правила, чтобы разумно их нарушать.

Дополнительные источники информации

Цель этой главы заключалась в том, чтобы рассмотреть ряд наиболее важных аспектов данного архитектурного стиля, но существует множество превосходных источников информации для его дальнейшего углубленного изучения. Дополнительные и более подробные сведения о микросервисах можно получить из следующих книг:

- «Building Microservices», Sam Newman (O'Reilly)[1]
- «Microservices vs. Service-Oriented Architecture», Mark Richards (O'Reilly)[2]
- «Microservices AntiPatterns and Pitfalls», Mark Richards (O'Reilly)[3]

[1] Ньюмен С. «Создание микросервисов». СПб., издательство «Питер».

[2] https://learning.oreilly.com/library/view/microservices-vs-service-oriented/9781491975657

[3] https://learning.oreilly.com/library/view/microservices-antipatterns-and/9781492042716

ГЛАВА 18

Выбор подходящего архитектурного стиля

Все зависит от конкретных обстоятельств! Имея в своем распоряжении все доступные варианты (с учетом почти ежедневного появления новых), мы бы хотели подсказать вам, каким именно следует воспользоваться, но это невозможно. Слишком большая зависимость в организации от множества факторов и от того, какое ПО в ней создается. Выбор архитектурного стиля является результатом анализа и размышлений о компромиссах, допускаемых в отношении архитектурных свойств, специфики предметной области, стратегических целей и множества других вещей.

Но независимо от контекста существует ряд общих советов по выбору подходящего архитектурного стиля.

Веяния моды в архитектуре

Со временем под воздействием ряда факторов меняются и предпочтения, отдаваемые тем или иным стилям:

Взгляд из прошлого

Как правило, новые архитектурные стили возникают в результате наблюдений и анализа возникающих ошибок. Знания, полученные во время работы с системами в прошлом, влияют на будущие решения. Архитекторы должны полагаться на свой накопленный опыт, то есть на то, что и позволило им стать архитекторами. Часто новые архитектурные проекты являются работой над ошибками предыдущих архитектурных стилей. Например, архитекторы серьезно задумались о последствиях многократного использования кода по-

сле создания архитектур, в которых оно было реализовано, и осознали все негативные компромиссы.

Изменения в экосистеме

Экосистеме разработки программных средств свойственны постоянные изменения, поскольку со временем меняется практически все. Изменения в нашей экосистеме настолько хаотичны, что предсказать их абсолютно невозможно. К примеру, пару лет назад никто ничего не слышал о проекте *Kubernetes*, а сейчас по всему миру проводятся конференции с тысячами разработчиков. А еще через пару лет на смену Kubernetes может прийти какой-нибудь другой, пока еще не созданный инструментарий.

Новые возможности

При появлении новых возможностей в архитектуре может произойти не столько замена одного инструмента другим, сколько переход к совершенно новой парадигме. Например, вряд ли с появлением таких контейнеров, как Docker, многие архитекторы или разработчики ожидали существенных сдвигов в мире разработки программных продуктов. Хотя это был эволюционный шаг, но влияние, оказанное им на архитекторов, инструменты, приемы разработки и множество других факторов, возымело поразительный эффект. Изменения экосистемы приводят к появлению новых наборов инструментов и дополнительных возможностей. Архитекторам нужно постоянно вести мониторинг обновлений не только инструментов, но и парадигм. Что-то может выглядеть как доработка того, что уже есть, но при этом содержать такие нюансы или изменения, которые меняют правила игры. Новые возможности необязательно должны потрясать весь мир разработки ПО: полученные функции могут быть теми самими скромными изменениями, которые точно соответствуют целям архитектора.

Ускорение

Дело не только в постоянных изменениях экосистемы, но и в постоянно нарастающей стремительности этого процесса. Новые инструменты позволяют создавать новые приемы разработки, ведущие к появлению совершенно иных дизайнов и возможностей. Архитекторы постоянно пребывают в изменчивой среде, поскольку перемены носят всеобъемлющий и непрерывный характер.

Изменения в предметной области

Предметная область, для которой разработчики создают программный продукт, постоянно изменяется: либо по причине непрекращающегося развития бизнеса, либо из-за слияния с другими бизнесами.

Технологические изменения

Поскольку технологии не стоят на месте, организации стараются идти в ногу хотя бы с некоторыми новинками, особенно с теми, что приносят очевидную прибыль.

Внешние факторы

Организационные изменения инициируются и множеством внешних факторов, косвенно связанных с разработкой программных продуктов. Например, архитекторы и разработчики могут быть абсолютно довольны конкретным инструментом, но стоимость лицензирования становится непомерно высокой, вынуждая переходить к другим вариантам.

Независимо от того, насколько организация следует веяниям современной архитектурной моды, архитектор должен улавливать текущие тенденции в своей сфере деятельности, чтобы принимать разумные решения о том, когда им нужно следовать, а когда лучше воздержаться.

Критерии принятия решения

При выборе архитектурного стиля следует учитывать все факторы, влияющие на структуру разработки ПО для той или иной предметной области. По сути, архитектор выстраивает проект по двум направлениям: по конкретной предметной области и по всем другим структурным элементам, необходимым для успешной работы системы.

Архитектор должен принимать проектное решение, исходя из анализа следующих факторов:

Предметная область

Архитекторы должны разбираться во многих важных аспектах предметной области, особенно в тех, что оказывают влияние на эксплуатационные свойства архитектуры. Необязательно быть экспертом в предметной области, но необходимо обладать пониманием основных особенностей той области, для которой разрабатывается дизайн.

Архитектурные свойства, влияющие на структуру

Следует выявить архитектурные свойства, важные для решения задач предметной области, с учетом влияния других внешних факторов.

Архитектура данных

Архитекторы и администраторы баз данных должны сотрудничать по вопросам применения БД, построения ее схемы и решению других задач, относя-

щихся к работе с данными. В этой книге архитектуре данных не уделяется много внимания, поскольку цель книги иная. Но архитекторам необходимо разбираться в том влиянии, которое оказывает схема данных на дизайн, особенно если новая система должна вступать во взаимодействие с более старой и/или уже используемой архитектурой данных.

Организационные факторы

На дизайн могут оказывать влияние многие внешние факторы. Например, идеальному дизайну может помешать стоимость облачных услуг конкретного поставщика. Или, возможно, компания планирует проводить слияния и поглощения, и тогда следует задуматься о перспективах использования открытых решений и интеграционных архитектур.

Процессы разработки, команды разработчиков и особенности эксплуатации

На дизайн влияют также многие специфические особенности проекта: процесс разработки ПО, взаимодействие (или его отсутствие) с эксплуатационниками или процесс QA (контроля качества программного продукта). Например, если организация еще не полностью освоила Agile-методы, применение архитектурных стилей, которые подразумевают использование этих методов, будет сильно затруднено.

Предметно-архитектурный изоморфизм

Некоторые предметные области соответствуют топологии архитектуры. Например, архитектурный стиль микроядра идеально подходит к системе, требующей индивидуальных настроек: архитектор может спроектировать настройки в виде дополнительных модулей (плагинов). Еще одним примером может послужить анализ генома, требующий большого количества дискретных операций, и архитектура на основе пространства, предлагающая большое количество отдельных процессоров.

Другим предметным областям могут совершенно не подходить некоторые стили архитектуры. Например, системы, требующие широкого масштабирования, совершенно не сочетаются с монолитным дизайном, поскольку весьма непросто обеспечить поддержку большого числа одновременно обращающихся пользователей в кодовой базе с сильной связанностью. Сильно разделенная распределенная архитектура плохо согласуется с предметной областью, включающей огромное количество семантических связей. Например, приложение для страховой компании, состоящее из многостраничных форм, каждая из которых основана на контексте предыдущих страниц, было бы сложно смоделировать в микросервисах. Эта проблема сильной связанности поставила бы архитекторов, выбирающих разделенную архитектуру, перед серьезными проблемами при проектировании, поэтому правильным выбором будет менее разделенная структура, например архитектура на основе сервисов.

Принимая во внимание все вышеизложенное, архитектор должен ответить на следующие вопросы:

Монолит или распределенная архитектура?

Применяя рассмотренную ранее концепцию квантов, архитектор должен определить: будет ли для проектирования достаточно одного набора архитектурных свойств или же разные части системы нуждаются в разных архитектурных свойствах? Единый набор подразумевает, что может подойти и монолит (хотя к использованию распределенной архитектуры архитектора могут подтолкнуть другие факторы), а различные архитектурные свойства приводят к выбору распределенной архитектуры.

Где должны находиться данные?

Если архитектура является монолитной, архитекторы обычно предполагают использовать одну или несколько реляционных баз данных. Для распределенной архитектуры следует решить, какими сервисами должны сохраняться данные. Кроме того, надо подумать о потоке данных по архитектуре для формирования рабочих процессов. При проектировании архитектуры необходимо учитывать и структуру, и поведение, а также не бояться итераций для получения наилучшей комбинации.

Каков будет стиль обмена данными между сервисами: синхронным или асинхронным?

После того как архитектор определил порядок разбиения данных, ему нужно решить, каким будет обмен данными между сервисами: синхронным или асинхронным? В большинстве случаев наиболее удобен синхронный обмен данными, но он может привести к снижению масштабируемости и надежности и другим негативным последствиям. Асинхронная связь способна обеспечить уникальные преимущества в плане производительности и масштабирования, но может вызвать множество проблем, связанных с синхронизацией данных, взаимоблокировкой, состоянием гонки, отладкой и т. д.

Поскольку синхронный обмен данными создает меньше проблем при проектировании, реализации и отладке, по умолчанию лучше выбирать именно этот режим, а к асинхронному обмену обращаться по мере необходимости.

По умолчанию используйте синхронный режим, а на асинхронный переходите по мере необходимости.

Результатом описанного выше процесса проектирования будет топология архитектуры, включающая в себя выбранный архитектурный стиль (и гибридные формы архитектуры), записи архитектурных решений в отношении тех частей проекта, которые потребовали наибольших усилий, а также функции пригодности для защиты важных принципов и эксплуатационных свойств архитектуры.

Конкретный пример монолита: Silicon Sandwiches

В архитектурном ката Silicon Sandwiches при исследовании свойств архитектуры мы определили, что для реализации этой системы будет достаточно одного кванта. Кроме того, это несложное приложение со скромным бюджетом, поэтому здесь нас привлекает простота монолитного решения.

Но все же для Silicon Sandwiches были созданы два разных дизайна компонентов: один с предметным, а другой с техническим разбиением. Учитывая простоту решения, мы создадим дизайн для каждого из вариантов и рассмотрим полученные компромиссы.

Модульный монолит

Модульный монолит предусматривает создание предметно-ориентированных компонентов с единой базой данных, развернутых в виде единого кванта. Модульный дизайн монолита для Silicon Sandwiches показан на рис. 18.1.

На рис. 18.1 показан монолит с единственной реляционной базой данных, реализованный с единым пользовательским интерфейсом на веб-основе (с тщательно продуманным дизайном для мобильных устройств) с целью уменьшения общих затрат. Все уже определенные архитектором предметные области представлены в виде компонентов. При достаточном количестве времени и ресурсов архитектор должен предусмотреть такое же разбиение таблиц и других составляющих базы данных по компонентам предметных областей, что упростит переход от этой архитектуры к распределенной, если в будущем возникнет такая необходимость.

Поскольку для индивидуальных настроек сам архитектурный стиль в силу своих особенностей не предназначен, следует ввести их в дизайн предметной области. Поэтому архитектор предусматривает эндпоинт `Override`, куда разработчики смогут загружать индивидуальные настройки. Для получения настраиваемых свойств каждый из компонентов предметной области ссылается на компонент переопределения `Override`, тем самым будет создана превосходная функция пригодности.

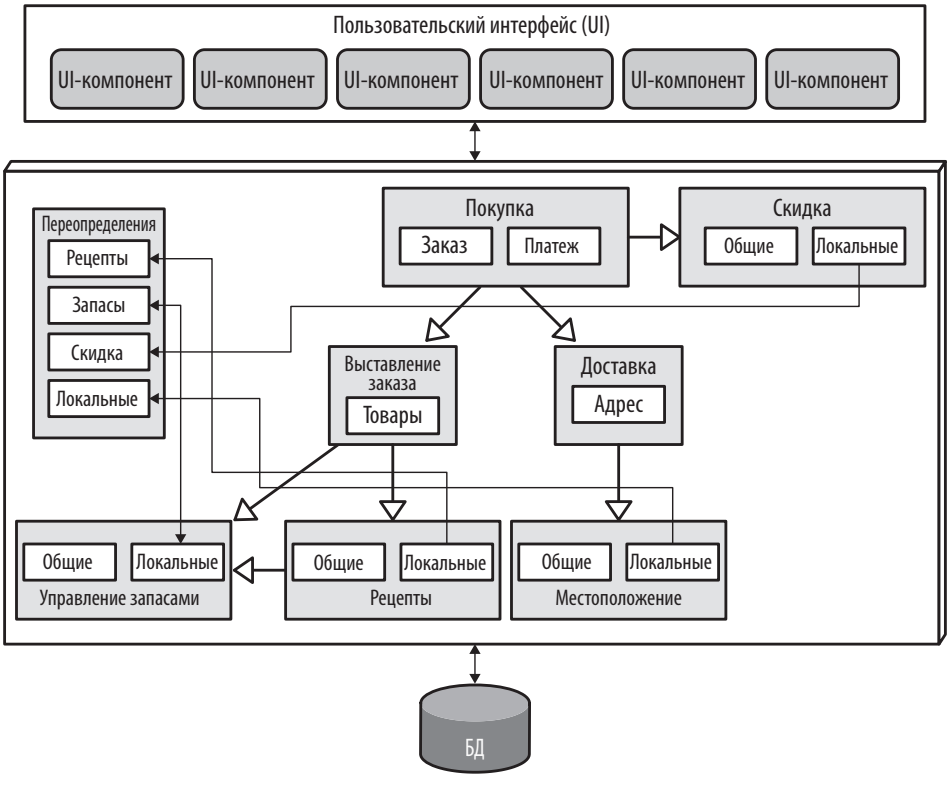


Рис. 18.1. Модульная монолитная реализация Silicon Sandwiches

Микроядро

Одним из архитектурных свойств, намеченных архитектором в Silicon Sandwiches, была настраиваемость. Проанализировав предметно-архитектурный изоморфизм, архитектор может выбирать реализацию с применением микроядра (рис. 18.2).

Основная система (ядро), показанная на рис. 18.2, состоит из компонентов предметной области и одной реляционной базы данных. Как и в предыдущем дизайне, хорошая синхронизация между предметными областями и структурой данных позволит в будущем мигрировать от ядра к распределенной архитектуре. Каждая настройка содержится в плагине; общие настройки — в одном наборе плагинов (с соответствующей базой данных), а также имеется ряд локальных плагинов, каждый с собственными данными. Поскольку плагины не должны быть связаны, каждый из них поддерживает только свои данные.

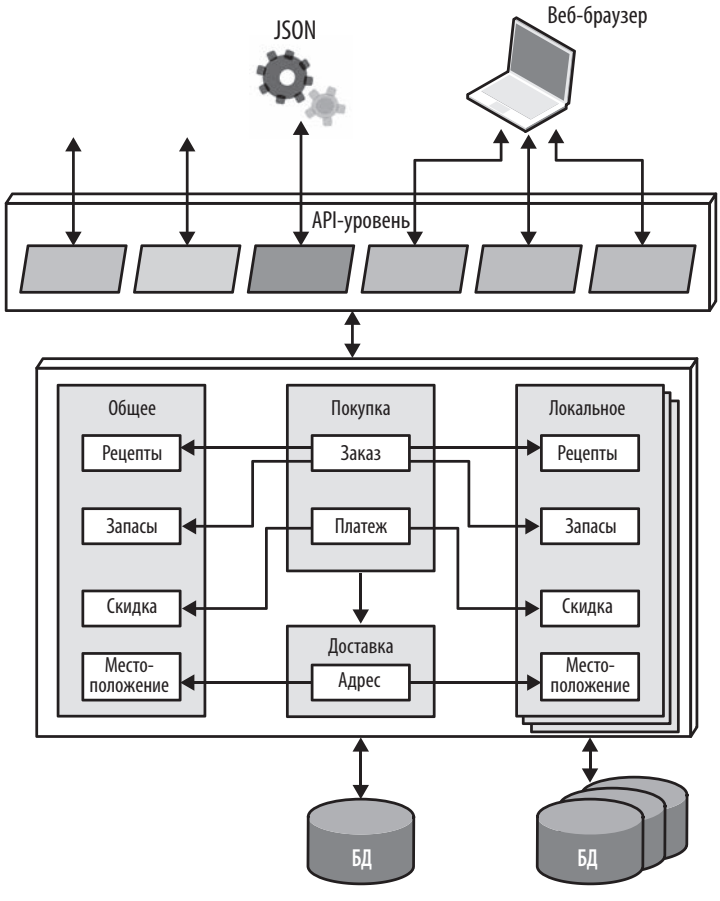


Рис. 18.2. Реализация Silicon Sandwiches с применением микроядра

Другим уникальным элементом дизайна здесь является паттерн Backends for Frontends (BFF)[1], что превращает API-уровень в тонкий адаптер микроядра. Он поставляет общую информацию от внутренних средств поддержки (бэкенда), а BFF-адаптеры переводят ее в подходящий формат для устройства пользовательского интерфейса (фронтенда). Например, BFF для iOS возьмет общий выход с бэкенда и настроит его под ожидания нативного приложения iOS: формата данных, разбиения на страницы, времени отклика и других факторов. Каждый BFF-адаптер позволяет создавать богатый пользовательский интерфейс и открывает возможность расширения для поддержки в будущем других устройств. Это одно из преимуществ стиля микроядра.

[1] https://samnewman.io/patterns/architectural/bff/

Обмен данными внутри любой архитектуры Silicon Sandwich может быть синхронным: архитектура не предъявляет экстремальных требований к производительности или адаптируемости, и ни одна из операций не будет иметь слишком продолжительного характера.

Конкретный пример распределенной архитектуры: Going, Going, Gone

Ката Going, Going, Gone (GGG) ставит перед нами более интересные задачи. Основываясь на анализе компонентов, рассмотренном в разделе «Конкретный пример: Going, Going, Gone: Выявление компонентов» на с. 149 главы 8, можно прийти к выводу, что разным частям архитектуры нужны разные архитектурные свойства. Например, к таким архитектурным свойствам, как доступность и масштабируемость, будут предъявляться абсолютно разные требования для ролей организатора торгов (auctioneer) и участника торгов (bidder).

Требования к GGG также явно указывают на вполне определенные высокие уровни масштабируемости, адаптируемости, производительности, а также на множество других непростых эксплуатационных свойств архитектуры. Архитектору нужно выбрать такую схему, которая позволит получить внутри архитектуры высокую степень настраиваемости на довольно тонком уровне. Из всех кандидатур из области распределенных архитектур наиболее соответствует требуемым свойствам либо низкоуровневая архитектура, управляемая событиями, либо микросервисы. Микросервисы лучше поддерживают различающиеся эксплуатационные архитектурные свойства. А вот используемые в чистом виде архитектуры, управляемые событиями, обычно не выделяют какие-либо части для реализации этих эксплуатационных архитектурных свойств; вместо этого они основываются на стиле обмена данными с оркестровкой, а не с хореографией.

Достичь в микросервисах заявленных показателей производительности будет непросто, но архитекторы зачастую способны за счет умелого проектирования справиться с любым слабым звеном архитектуры. Например, микросервисы по своей природе обеспечивают высокую степень масштабируемости, но при этом архитекторам приходится решать конкретные проблемы с производительностью, вызванные слишком большим объемом оркестровки, слишком агрессивным разделением данных и т. д.

Реализация ката GGG с использованием микросервисов представлена на рис. 18.3.

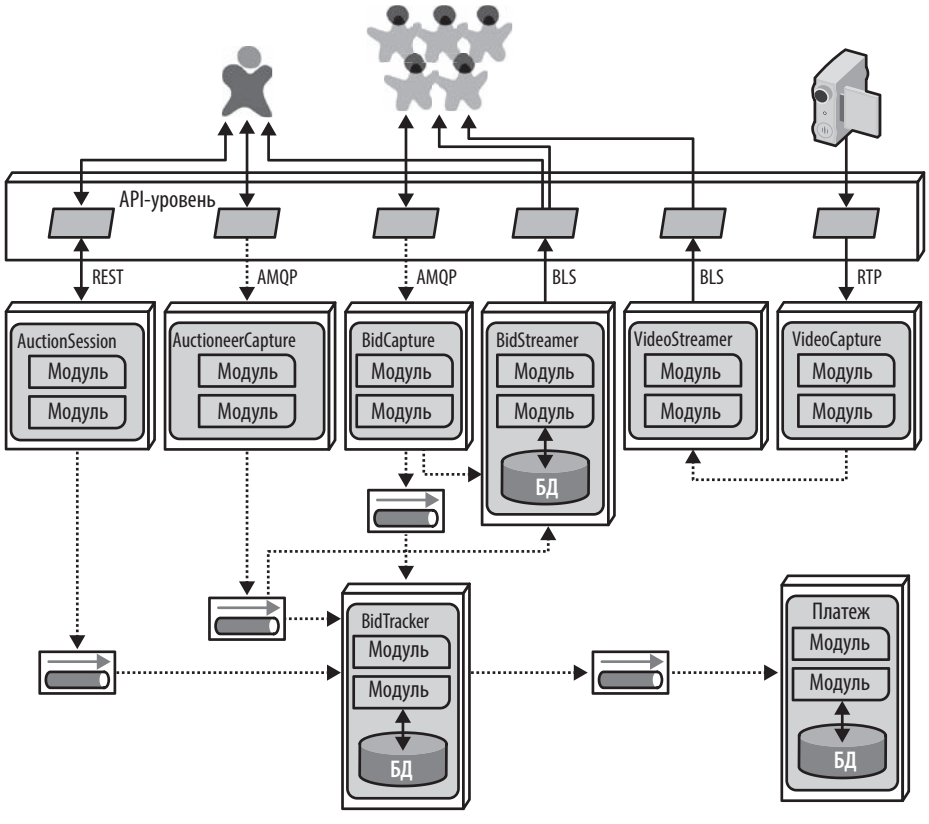


Рис. 18.3. Микросервисная реализация Going, Going, Gone

Как показано на рис. 18.3, все выявленные компоненты стали в архитектуре сервисами, при этом гранулярность компонентов и гранулярность сервисов должны соответствовать друг другу. У GGG имеется три различных пользовательских интерфейса:

Участник торгов (Bidder)

Многочисленные участники торгов онлайн-аукциона.

Организатор торгов, или аукционист (Auctioneer)

Он один на аукцион.

Создатель потока (Streamer)

Сервис, отвечающий за создание видеопотока и потока ставок для участников торгов. Следует заметить, что это поток, предназначенный только

для просмотра, позволяющий выполнять оптимизацию, недоступную при необходимости установки обновлений.

В этом дизайне GGG-архитектуры намечаются следующие сервисы:

`BidCapture` (захватчик ставок)

Захватывает вводимые онлайн-ставки участника торгов и отправляет их в асинхронном режиме отслеживателю ставок `BidTracker`. Этому сервису не требуется сохранение данных, поскольку он работает как средство передачи онлайн-ставок.

`BidStreamer` (создатель потока ставок)

Создает обратный поток ставок для онлайн-участников в высокопроизводительном, предназначенном только для просмотра потоке.

`BidTracker` (отслеживатель ставок)

Отслеживает ставки как от захватчика ставок аукциониста `AuctioneerCapture`, так и от захватчика ставок `BidCapture`. Этот компонент объединяет два разных информационных потока, размещая ставки как можно ближе к реальному времени. Следует обратить внимание, что оба входящих подключения к этому сервису являются асинхронными, что позволяет разработчикам использовать очереди сообщений в качестве буферов для обработки сильно отличающихся скоростей потока сообщений.

`AuctioneerCapture` (захватчик ставок аукциониста)

Захватывает ставки, делающиеся через организатора. Результат анализа кванта в разделе «Конкретный пример: Going, Going, Gone: Выявление компонентов» главы 8 на с. 149 подсказал архитектору решение о разделении единого захватчика на `BidCapture` и `AuctioneerCapture`, поскольку им требуются абсолютно разные архитектурные свойства.

`AuctionSession` (сеанс аукциона)

Управляет рабочим процессом отдельно взятых аукционов.

`Payment` (платеж)

Сторонний поставщик платежа, обрабатывающий платежную информацию после того, как `AuctionSession` завершил аукцион.

`VideoCapture` (захватчик видеопотока)

Захватывает видеопоток живого аукциона.

`VideoStreamer` (создатель видеопотока)

Создает аукционный видеопоток для онлайн-участников аукциона.

Архитектор весьма тщательно подошел к определению как синхронного, так и асинхронного стиля обмена данными в этой архитектуре. Его выбор в пользу асинхронного обмена данными обусловлен в первую очередь учетом разных эксплуатационных архитектурных свойств разных сервисов. Например, если сервис `Payment` способен обрабатывать новый платеж не чаще чем раз в 500 мс и при этом одновременно завершается большое число аукционов, синхронный обмен данными между сервисами приведет к истечениям сроков ожидания (тайм-аутам) и другим неприятным последствиям, снижающим надежность системы. Используя очереди сообщений, архитектор может повысить надежность в критической части архитектуры, продемонстрировавшей свою хрупкость.

В результате финального анализа дизайн был сведен к пяти квантам, обозначенным на рис. 18.4.

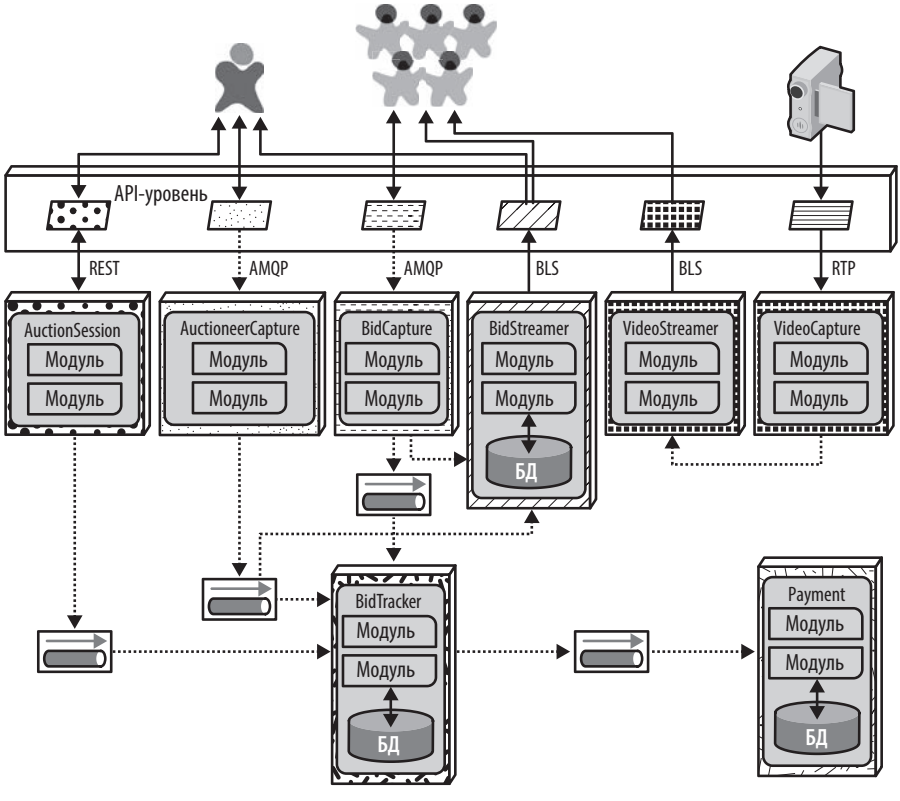


Рис. 18.4. Границы квантов для GGG

Дизайн, показанный на рис. 18.4, включает кванты для платежей (`Payment`), организатора торгов (`Auctioneer`), участника торгов (`Bidder`), потоков участников торгов (`BidderStreams`) и отслеживателя ставок (`BidTracker`), примерно соответствующие создаваемым сервисам. При наличии нескольких экземпляров они обозначаются на схеме стеками контейнеров. Использование квантового анализа на стадии проектирования компонентов позволило архитектору упростить определение границ сервисов, структуры данных и режима обмена данными.

Отметим, что все это не стоит считать неким «правильным» дизайном для GGG, и разумеется, это не единственный возможный вариант. Мы даже не настаиваем, что это лучший дизайн из всех возможных, но при этом полагаем, что у него не самый плохой набор компромиссов. Выбор микросервисов, а затем разумное использование событий и сообщений позволяет получить максимальный выигрыш от применения универсальной архитектурной модели, создавая одновременно основу для ее будущего развития и расширения.

ЧАСТЬ III

Технические приемы и гибкие навыки

Чтобы думать как архитектор, руководить командами разработчиков и убеждать стейкхолдеров, успешный архитектор ПО должен не только разбираться в технических аспектах, но и обладать основными гибкими навыками[1] менеджера. Эта часть книги посвящена ключевым приемам и гибким навыкам, необходимым для профессионального становления архитектора.

[1] Для английского выражения «soft skills» нет устоявшегося русского эквивалента. Чаще всего употребляют «гибкие навыки», иногда «софт скиллс» или «мягкие компетенции». Под этим подразумевается умение работать в команде, успешно коммуницировать, эмпатия и т. п. — *Примеч. ред.*

ГЛАВА 19

Архитектурные решения

Одна из основных обязанностей архитектора — принимать архитектурные решения. Эти решения обычно касаются структуры приложения или системы, но могут также затрагивать выбор технологических средств, влияющих на свойства архитектуры. Независимо от контекста, архитектурное решение можно назвать удачным, если оно помогает командам разработчиков сделать правильный технический выбор. Принятие архитектурных решений включает в себя сбор достаточного объема информации для обоснования решения, документирование решения и умение убедить всех стейкхолдеров в его правильности.

Антипаттерны архитектурных решений

Принятие архитектурных решений — это целое искусство. Неудивительно, что архитектор может попасть в ситуацию, описываемую одним из так называемых архитектурных антипаттернов. Известный программист Эндрю Кениг (Andrew Koenig) определяет антипаттерн как нечто, что кажется хорошей идеей, когда вы начинаете, но в итоге приносит одни неприятности. Антипаттерном также называют повторяющийся процесс, выдающий отрицательные результаты. Три основных архитектурных антипаттерна, которые могут возникнуть (и часто возникают) в процессе принятия архитектурных решений: антипаттерн *Covering Your Assets (излишняя перестраховка)*, антипаттерн *Groundhog Day (день сурка)* и антипаттерн *Email-Driven Architecture (архитектура, управляемая имейлами)*. Обычно эти три антипаттерна стоят на пути архитектора друг за другом: если избежать антипаттерна Covering Your Assets, можно попасть в антипаттерн Groundhog Day, избегание которого, в свою очередь, ведет в антипаттерн Email-Driven Architecture. Принятие удачных и выверенных архитектурных решений требует от архитектора преодоления всех трех антипаттернов.

Антипаттерн Covering Your Assets

Первым антипаттерном, в который архитектор может попасть при принятии архитектурных решений, является антипаттерн Covering Your Assets (излишняя перестраховка). Он возникает, когда архитектор избегает или откладывает принятие архитектурного решения из-за страха сделать неправильный выбор.

Существует два способа избежать этого антипаттерна. Во-первых, для принятия важного архитектурного решения можно дождаться *последнего ответственного момента*. Этот момент наступает, когда накоплено достаточно информации для обоснования и проверки вашего решения; но не стоит ждать слишком долго, задерживая команды разработчиков или попадая в антипаттерн *аналитического паралича* (*Analysis Paralysis*). Во-вторых, избежать этого антипаттерна можно при постоянном сотрудничестве с командами разработчиков, чтобы убедиться, что решение будет реализовано так, как и предполагалось. Это крайне важно, поскольку архитектор не может знать все особенности конкретной технологии и все связанные с ней вопросы. Тесно сотрудничая с командами разработчиков, архитектор в случае возникновения проблем сможет быстро скорректировать архитектурное решение.

В качестве иллюстрации предположим, что архитектор решает кэшировать все справочные данные о товарах (описание, вес, размеры) во всех экземплярах сервисов, нуждающихся в этой информации, и воспользоваться для этого реплицируемым кэшем, предназначенным только для чтения, где владельцем первой копии будет сервис каталога. Реплицируемый кэш предполагает, что при любых изменениях информации о товаре (или при добавлении нового товара) сервис каталога обновит свой кэш, который затем будет реплицирован на все другие сервисы, требующие эти данные, с помощью реплицированного (в памяти) кэша продукта. Неплохим обоснованием данного решения послужит уменьшение связанности между сервисами и эффективный обмен данными без потребности в межсервисных вызовах. Но команды разработчиков, реализующие это архитектурное решение, выясняют, что из-за требований к масштабируемости ряда сервисов объем необходимой памяти превысит объем доступной памяти. Благодаря тесному сотрудничеству с командами разработчиков архитектор получит своевременное уведомление о грядущей проблеме и скорректирует архитектурное решение, чтобы справиться с подобными ситуациями.

Антипаттерн Groundhog Day

Избежав антипаттерна Covering Your Assets и приступив к принятию решений, архитектор может попасть в следующий антипаттерн — Groundhog Day (день сурка). Он возникает, когда люди не понимают причин принятия решения

и продолжают обсуждать его снова и снова. Антипаттерн получил свое название благодаря фильму с Биллом Мюрреем «День сурка», в котором каждое утро неизменно наступало 2 февраля.

Этот антипаттерн складывается из-за того, что архитектор не может убедительно обосновать свое решение (или весь проект). Ведь важно предоставлять не только технические, но и бизнес-аргументы. К примеру, архитектор решает разбить монолитное приложение на отдельные сервисы и разделить функциональность приложения таким образом, чтобы каждая часть приложения потребляла меньше ресурсов виртуальной машины и могла сопровождаться и развертываться отдельно от других частей. Это хороший вариант технического обоснования, но не хватает бизнес-аргументов: иными словами, почему бизнес должен оплачивать этот рефакторинг? Подходящим бизнес-обоснованием данного решения может стать более оперативная реализация новых бизнес-функций, что сократит время выхода на рынок. Еще одним аргументом будет снижение затрат, связанных с разработкой и выпуском новых функций.

Наличие ценных бизнес-аргументов в обосновании крайне важно для любого архитектурного решения. Они также могут послужить неплохой лакмусовой бумажкой для определения целесообразности принятия того или иного решения. Если конкретное архитектурное решение бесполезно для бизнеса, возможно, это далеко не лучшее решение и его следует пересмотреть.

Четыре наиболее распространенных бизнес-обоснования — это стоимость, время вывода на рынок, удовлетворенность пользователей и стратегическое позиционирование. Сосредоточившись на этих основных бизнес-обоснованиях, надо не забывать и то, что важно для всех стейкхолдеров. Обоснование того или иного решения только экономией средств может быть не самым правильным подходом, если бизнес-стейкхолдеров больше волнует время вывода конечного продукта на рынок, а не затраты.

Антипаттерн Email-Driven Architecture

После того как архитектор принял решения и полностью их обосновал, возникает опасность попадания в третий архитектурный антипаттерн: *Email-Driven Architecture* (архитектура, управляемая имейлами). Он возникает, когда люди не понимают сути принятого решения, забывают о нем или даже не знают о том, что оно было принято, и поэтому не в состоянии его реализовать. Этот антипаттерн касается эффективного доведения ваших архитектурных решений до исполнителей. Электронная почта, конечно, хорошее средство коммуникации, но на ее основе не выстроить нужную систему хранения документации.

Существует множество способов эффективной коммуникации, позволяющих избежать антипаттерна Email-Driven Architecture. Первое правило: не включать архитектурное решение в тело электронного письма. В противном случае расплодится множество систем записи этого решения. Зачастую важные подробности (включая обоснование) не попадают в электронную почту, воскрешая раз за разом антипаттерн дня сурка. И вообще, если это архитектурное решение будет когда-либо изменено или заменено другим, то каким образом люди получат обновленное решение? Лучше в теле письма упомянуть только сам смысл решения и предоставить ссылку на единую систему записи для получения актуального архитектурного решения и соответствующих подробностей (неважно, что это будет: ссылка на вики-страницу или же на документ в файловой системе).

Второе правило: уведомлять только тех людей, которые действительно заинтересованы в архитектурном решении. Конструктивно составленное электронное письмо может иметь следующий вид:

> «Привет, Сандра, я принял важное решение по обмену данными между сервисами, непосредственно касающееся твоей работы. Пожалуйста, ознакомься с ним по следующей ссылке...»

Обратите внимание на фразу в первом предложении: «важное решение по обмену данными между сервисами». В ней упомянут только смысл решения, но не оно само. Вторая часть первого предложения еще важнее: «непосредственно касающееся твоей работы». Если архитектурное решение не имеет прямого отношения к человеку, то зачем его беспокоить? Данное рассуждение — отличная лакмусовая бумажка для определения круга заинтересованных лиц (включая разработчиков), кого следует непосредственно уведомить о принятии архитектурного решения. Во втором предложении предоставляется ссылка на то место, где находится архитектурное решение, то есть на единственное место, где его можно найти, из чего вытекает существование единой системы записи решений.

Архитектурно значимые решения

Многие архитекторы считают, что если архитектурное решение основано на какой-то конкретной технологии, то это уже не архитектурное, а техническое решение. Но это верно далеко не всегда. Если архитектор принимает решение воспользоваться конкретной технологией по причине того, что она непосредственно поддерживает определенное архитектурное свойство (например, производительность или масштабируемость), то тогда это уже архитектурное решение.

Майкл Нейгард (Michael Nygard)[1], широко известный архитектор ПО и автор книги «Release It!» (Pragmatic Bookshelf)[2], предложил ответ на вопрос, за какие решения должен отвечать архитектор (то есть что собой представляют архитектурные решения), введя такое понятие, как *архитектурно значимые решения*. Согласно его определению, к архитектурно значимым относятся решения, влияющие на структуру, нефункциональные свойства, зависимости, интерфейсы или технологические методы разработки.

Структура касается решений, влияющих на используемые паттерны или стили архитектуры. В качестве примера можно привести решение по обмену данными между набором микросервисов. Это окажет влияние на ограниченный контекст микросервиса, а следовательно, и на структуру приложения.

Нефункциональные свойства — это свойства архитектуры (выражаемые словами с окончанием *-ость*), играющие важную роль для разрабатываемых или сопровождаемых приложений и систем. Если выбор технологии влияет на производительность, а та, в свою очередь, является важным аспектом приложения, значит, этот выбор превращается в архитектурное решение.

Зависимости определяют связанность компонентов и/или сервисов внутри системы, а она, в свою очередь, влияет на общую масштабируемость, модульность, гибкость, тестируемость, надежность и т. д.

Интерфейсы касаются способов доступа к сервисам и компонентам и их оркестровки, обычно посредством шлюза, концентратора, сервисной шины или прокси-сервера API. В интерфейсах зачастую используются определения контрактов, включая стратегию управления версиями контрактов и снятия устаревших версий. Интерфейсы влияют на пользователей и, следовательно, являются архитектурно значимыми.

И наконец, *технологические методы разработки* относятся к платформам, средам и даже процессам, которые хоть и технические по своей природе, но могут влиять на ряд архитектурных аспектов.

Запись архитектурных решений

Одним из наиболее действенных способов документирования архитектурных решений является ведение *записей архитектурных решений* (Architecture

[1] https://www.michaelnygard.com/

[2] Нейгард М. «Release it! Проектирование и дизайн ПО для тех, кому не все равно». СПб., издательство «Питер».

Decision Records (ADR[1])). Эти записи были впервые предложены в публикации блога[2] Майкла Нейгарда, а впоследствии отмечены как рекомендованные к использованию в ThoughtWorks Technology Radar[3]. ADR представляет собой короткий текстовый файл (обычно не более одной-двух страниц) с описанием специфики архитектурного решения. Хотя ADR могут быть записаны с помощью обычного текста, чаще они делаются в каком-либо формате текстового документа, например AsciiDoc[4] или Markdown[5]. Как вариант, ADR может вестись с помощью шаблона вики-страницы.

Для управления ADR доступны также инструментальные средства. Нат Прайс (Nat Pryce), соавтор книги «Growing Object-Oriented Software Guided by Tests» (Addison-Wesley), создал для ведения ADR инструмент с открытым кодом под названием ADR-tools[6]. Этот инструмент предоставляет интерфейс командной строки для управления ADR, включая местоположения, схемы нумерации и логику замен. Миха Копс (Micha Kops), разработчик ПО из Германии, опубликовал заметку[7] в блоге об использовании ADR-tools, где приведен ряд весьма полезных примеров применения этого инструмента для управления записью архитектурных решений.

Базовая структура

Структура ADR состоит из пяти главных разделов: *Название* (*Title*), *Статус* (*Status*), *Контекст* (*Context*), *Решение* (*Decision*) и *Последствия* (*Consequences*). Мы обычно добавляем еще два раздела: *Соблюдение требований*, или *Комплаенс* (*Compliance*), и *Примечания* (*Notes*). Эта базовая структура (рис. 19.1) может быть расширена за счет любого другого раздела, представляющегося необходимым, при условии сохранения последовательности и лаконичности документа. Например, если нужен анализ всех других альтернативных решений, можно добавить раздел *Альтернативы* (*Alternatives*).

---

1 https://adr.github.io/

2 https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions

3 https://www.thoughtworks.com/radar/techniques/lightweight-architecture-decision-records

4 http://asciidoc.org/

5 https://www.markdownguide.org/

6 https://github.com/npryce/adr-tools

7 https://www.hascode.com/2018/05/managing-architecture-decision-records-with-adr-tools/

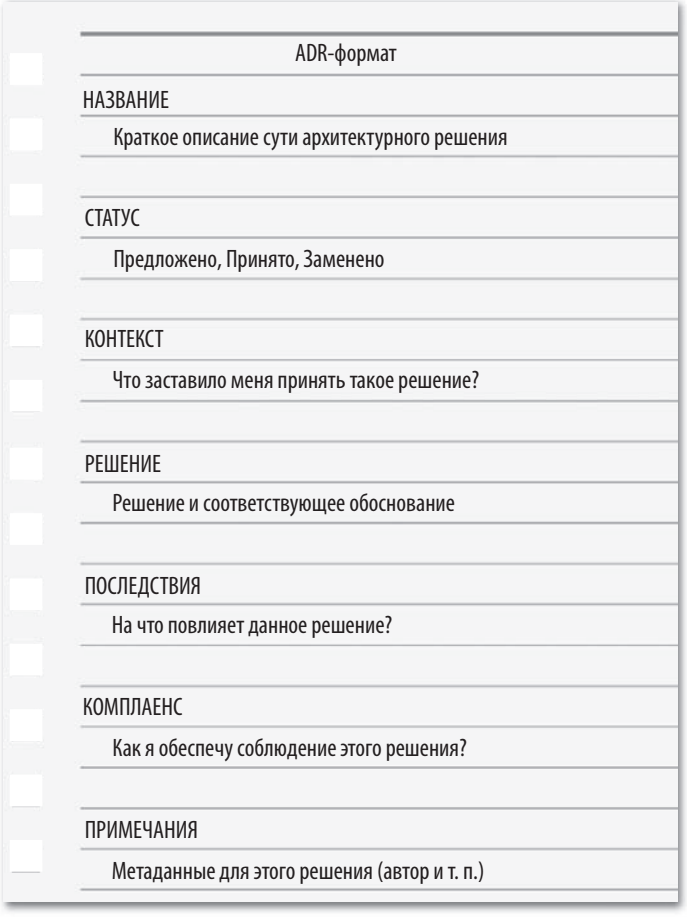


Рис. 19.1. Базовая структура ADR

Название

Название ADR обычно имеет последовательную нумерацию и содержит краткую фразу, дающую описание архитектурных решений. Например, решение об использовании асинхронного обмена сообщениями между сервисом заказов и сервисом платежей может выглядеть следующим образом: «42. Использование асинхронного обмена сообщениями между сервисами Order и Payment». Название должно быть достаточно информативным для устранения любой неясности относительно сути и контекста решения и в то же время кратким и понятным.

Статус

В статусе ADR могут быть пометки *Предложено*, *Принято* или *Заменено*. Статус *Предложено* означает, что решение должно быть утверждено либо тем, кто принимает решения на более высоком уровне, либо каким-то органом архитектурного управления (например, наблюдательным советом). Статус *Принято* означает, что решение утверждено и готово к реализации. Статус *Заменено* означает, что решение изменилось и его запись была заменена другой ADR-записью. Этот статус всегда предполагает, что предыдущим статусом ADR был *Принято*; иначе говоря, ADR со статусом *Предложено* никогда не будет заменена другой ADR-записью, и решение будет совершенствоваться до тех пор, пока его запись не получит статус *Принято*.

Статус *Заменено* является действенным способом сохранения истории принимаемых решений, причин их принятия, а также сути нового решения и причинах изменения прежнего решения. Обычно когда ADR меняется, она помечается указанием на то решение, которое ее заменило. А запись нового решения, в свою очередь, содержит пометку об ADR-записи предыдущего. Предположим, к примеру, что ADR 42 («Использование асинхронного обмена сообщениями между сервисами Order и Payment») ранее было утверждено, но в силу последующего изменения реализации и местонахождения сервиса Payment для обмена данными между двумя сервисами должна теперь использоваться технология REST (ADR 68). Статус должен приобрести следующий вид:

> *ADR 42. Использование асинхронного обмена сообщениями между сервисами Order и Payment*
>
> Статус: Заменено решением 68
>
> *ADR 68. Использование между сервисами Order и Payment технологии REST*
>
> Статус: Принято, заменяет решение 42
>
> Ссылка и прослеживание истории между ADR-записями 42 и 68 позволяют избежать ожидаемого вопроса касательно ADR 68: «А как насчет использования обмена сообщениями?».

Другим важным аспектом раздела «Статус» является то, что он подталкивает архитектора к выяснению у своего начальника или ведущего архитектора критериев, по которым можно самостоятельно утверждать архитектурные решения или же отправлять на дополнительное согласование архитектору более высокого уровня, наблюдательному совету или какому-то другому органу управления архитектурой.

ADR И ЗАПРОС КОММЕНТАРИЕВ (REQUEST FOR COMMENTS, RFC)

Когда архитектору необходимо отправить черновую ADR-запись для получения комментариев (что вполне оправданно, если он хочет знать мнение всех стейкхолдеров), мы рекомендуем создать новый статус под названием *Запрос комментариев* (Request for Comments, RFC) и указать дедлайн для высказывания мнений. Это позволит избежать попадания в антипаттерн аналитического паралича, когда решение обсуждается вечно, но никогда не принимается. Как только наступит дедлайн, архитектор может проанализировать все комментарии в отношении проекта ADR, внести в решение любые необходимые поправки, принять окончательный вариант решения и установить статус «Предложено» (в том случае, если архитектор сам не может его утвердить, иначе он установит статус «Принято»). Пример статуса RFC может выглядеть следующим образом:

СТАТУС

Запрос комментариев, дедлайн 9.02.2010

Три критерия, с которых можно начать, — это стоимость, влияние, оказываемое на все команды, и безопасность. Стоимость может складываться из расходов на приобретение программных средств или на лицензионные сборы, дополнительных затрат на технические средства, а также из общего уровня затрат на усилия по реализации архитектурного решения. Уровень затрат на усилия можно оценить, перемножив расчетное количество часов на реализацию архитектурного решения со стандартной, принятой в компании, ставкой *эквивалента полной занятости* (Full-Time Equivalency, FTE). Сведениями о ставке FTE обычно располагает владелец проекта или его менеджер. Если стоимость архитектурного решения превышает определенную сумму, его записи следует присвоить статус «Предложено» и получить утверждение от кого-либо еще. Если архитектурное решение оказывает влияние на работу других команд или систем либо имеет любые последствия для безопасности, оно не может быть утверждено самим архитектором и должно получить одобрение от управляющего органа более высокого уровня или от ведущего архитектора.

После того как критерии и соответствующие ограничения установлены и согласованы (например, «расходы, превышающие 5000 евро, должны быть одобрены наблюдательным советом»), они должны быть задокументированы, чтобы все архитекторы, создающие ADR-записи, знали, когда они могут и когда не могут утверждать свои собственные архитектурные решения.

Контекст

В ADR-разделе «Контекст» указываются обстоятельства, в силу которых принималось решение. Иными словами, «что именно заставило меня принять такое решение?». Этот раздел позволяет архитектору обрисовать конкретную ситуацию или проблему и кратко изложить возможные альтернативы. Если архитектору нужно подробно задокументировать анализ каждого альтернативного варианта, то в ADR можно добавить дополнительный раздел «Альтернативы» и не вносить эту аналитику в раздел «Контекст».

Раздел «Контекст» также предоставляет способ документирования архитектуры. При описании контекста архитектор заодно дает описание архитектуры. Это весьма действенный способ документирования конкретной области архитектуры понятным и лаконичным образом. Если продолжить пример из предыдущего раздела, то контекст может иметь следующий вид: «Сервис заказов должен передать информацию сервису платежей для оплаты размещаемого в данный момент заказа. Это можно сделать, воспользовавшись технологией REST или асинхронным обменом сообщений». Обратите внимание, что в этом кратком изложении указан не только сценарий, но и альтернатива.

Решение

ADR-раздел «Решение» содержит собственно архитектурное решение, а также его полное обоснование. Майкл Нейгард ввел в практику отличный способ заявить об архитектурном решении не в пассивной, а в активной, утвердительной форме. Например, решение воспользоваться асинхронным обменом сообщениями между сервисами будет читаться следующим образом: «*Мы будем использовать* асинхронный обмен сообщениями между сервисами». Это позволяет изложить решение гораздо более убедительно, чем: «*Я полагаю*, что асинхронный обмен сообщениями будет наиболее подходящим выбором». Обратите внимание, что во втором варианте не разъясняется суть решения, а также было ли оно принято к реализации или нет; изложено только мнение архитектора.

Возможно, один из наиболее важных аспектов раздела «Решение» в ADR состоит в том, что он позволяет архитектору больше уделить внимание ответу на вопрос «почему», нежели на вопрос «как». Понимание причины принятия решения гораздо важнее, чем понимание, как именно что-то работает. Взглянув на контекстные диаграммы, большинство архитекторов и разработчиков смогут разобраться в том, что и как работает, но не в том, почему было принято такое решение. Знание причин принятия решения и его обоснования помогает глубже понять контекст задачи и избежать возможных ошибок при рефакторинге под другое решение, способное стать источником проблем.

Чтобы проиллюстрировать сказанное, рассмотрим реальное архитектурное решение, принятое несколько лет назад: использовать среду удаленного вызова процедур от компании Google (Google's Remote Procedure Call, gRPC)[1] в качестве средства обмена данными между двумя сервисами. Не разобравшись в причинах принятия такого решения, другой архитектор через несколько лет решил отказаться от него и воспользоваться обменом сообщениями, чтобы усилить разделенность сервисов. Но рефакторинг совершенно неожиданно вызвал резкое повышение периода ожидания, что, в свою очередь, приводило к тайм-аутам в вышележащих уровнях системы. Если бы второй архитектор знал, что технология gRPC была выбрана именно для существенного снижения задержки (за что было заплачено ценой тесной связанности сервисов), то ему бы не пришло в голову менять решение.

Последствия

ADR-раздел «Последствия» играет не менее важную роль. В нем документируется общее воздействие архитектурного решения. Каждое принятое архитектурное решение имеет то или иное воздействие, как положительное, так и отрицательное. Анализ этих воздействий заставляет архитектора задуматься: а не перевесит ли соответствующий отрицательный эффект всю пользу от принятого решения?

Еще одним положительным моментом присутствия данного раздела является документирование анализа компромиссов, связанных с архитектурным решением. Эти компромиссы могут основываться на стоимостных соображениях или на анализе всех «за» и «против» по сравнению с другими свойствами архитектуры. Рассмотрим, к примеру, решение по использованию асинхронного обмена сообщениями (по принципу «выстрелил и забыл») для публикации отзыва на веб-сайте. Обоснованием этого решения служит существенное улучшение отзывчивости на запросы по публикации отзывов с 3100 мс до 25 мс за счет того, что пользователям не придется ждать фактической публикации отзыва (ожидание коснется только отправки сообщения в очередь). Обоснование вполне убедительное, но кто-то может возразить, сказав, что сама идея не выдерживает критики из-за сложностей с обработкой ошибок, присущих асинхронным запросам («А что, если кто-то опубликует отзыв с неприемлемыми словами?»). Специалисту, критикующему данное решение, неизвестно, что вопрос уже обсуждался с бизнес-стейкхолдерами, а также с другими архитекторами, и было принято решение пойти на компромисс, поскольку важнее добиться повышения отзывчивости и справиться с усложненной обработкой ошибок, чем заставлять

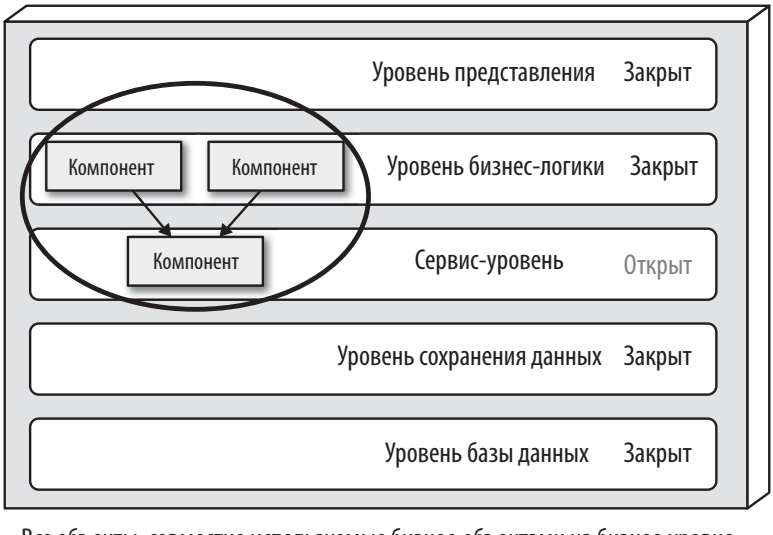
[1] https://www.grpc.io/

пользователя ждать ответа об успешной публикации отзыва при синхронном обмене данными. При использовании ADR этот анализ компромиссов может быть включен в раздел «Последствия», предоставляя полную картину обстоятельств (и компромиссов) принятия архитектурного решения и позволяя избежать подобных ситуаций.

Комплаенс

Раздел «Комплаенс» в ADR не является стандартным, но мы настоятельно рекомендуем его добавлять. Этот раздел заставляет архитектора задуматься, как оценивать и управлять архитектурным решением для проверки соблюдения его требований. Необходимо решить, должно ли соответствие требованиям контролироваться вручную или же этот контроль можно автоматизировать с помощью функции пригодности. Во втором случае архитектор может далее указать в разделе, как должна создаваться эта функция и будут ли другие изменения кодовой базы, необходимые для оценки комплаенса данного архитектурного решения.

Рассмотрим, к примеру, следующее, показанное на рис. 19.2, архитектурное решение в традиционной многоуровневой архитектуре: «Все объекты, совместно



Все объекты, совместно используемые бизнес-объектами на бизнес-уровне, будут находиться на уровне общих сервисов, они будут изолированы и будут содержать общую функциональность

Рис. 19.2. Пример архитектурного решения

используемые бизнес-объектами на бизнес-уровне, будут находиться на уровне общих сервисов, они будут изолированы и будут содержать общую функциональность».

Комплаенс этого архитектурного решения может оцениваться и определяться автоматически, путем использования либо ArchUnit[1] в Java, либо NetArchTest[2] в C#. Например, при использовании ArchUnit в Java функция пригодности для автоматизированного тестирования может иметь следующий вид:

```
@Test
public void shared_services_should_reside_in_services_layer() {
    classes().that().areAnnotatedWith(SharedService.class)
        .should().resideInAPackage("..services..")
        .because("All shared services classes used by business " +
                 "objects in the business layer should reside in the services " +
                 "layer to isolate and contain shared logic")
        .check(myClasses);
}
```

Заметьте, что эта автоматическая функция пригодности потребует написания новых сценариев для создания новой Java-аннотации (`@SharedService`) и последующего добавления этой аннотации ко всем совместно используемым классам. В этом разделе также указывается, каким должен быть тест, где его можно будет найти, а также как и когда он будет выполнен.

Примечания

Мы также настойчиво рекомендуем добавлять в ADR еще один нестандартный раздел — «Примечания». В него включаются различные метаданные, касающиеся ADR-записи, например:

- Исходный автор.
- Дата утверждения.
- Кем утверждено.
- Дата замены.
- Дата последней корректировки.
- Кем скорректировано.
- Последняя корректировка.

[1] https://www.archunit.org/

[2] https://github.com/BenMorris/NetArchTest

Даже если ADR-записи сохраняются в системе управления версиями (например, в Git), дополнительные метаданные будут нелишними и дополнят те, что поддерживаются репозиторием, поэтому мы рекомендуем добавлять этот раздел независимо от того, как и где хранятся ADR.

Сохранение ADR

Раз архитектор создает ADR, то она должна где-то храниться. Независимо от того, где именно сохраняются ADR-записи, каждому архитектурному решению должен принадлежать свой собственный файл или вики-страница. Кому-то из архитекторов нравится хранить ADR в Git-репозитории вместе с исходным кодом. Это позволяет управлять версиями ADR и отслеживать ее историю. Но мы хотим предостеречь крупные организации от подобной практики по нескольким причинам. Во-первых, не у всех, кому нужно просматривать архитектурное решение, может быть доступ к Git-репозиторию. Во-вторых, это не самое удачное место для хранения тех ADR, чей контекст выходит за рамки Git-репозитория приложения (имеются в виду интеграционные архитектурные решения, корпоративные архитектурные решения или же решения, распространяющиеся на каждое приложение). Поэтому мы рекомендуем хранить ADR либо в вики-системе (используя вики-шаблон), либо в совместно используемом каталоге, находящемся на общем файловом сервере, с доступом из вики-среды или из другого программного средства визуализации документов. Пример возможной структуры каталогов (или структуры переходов по вики-страницам) показан на рис. 19.3.

Каталог *application* (приложение) содержит архитектурные решения, относящиеся к определенному контексту приложения. Этот каталог разбит на несколько подкаталогов. Подкаталог *common* (общие) предназначен для архитектурных решений, применимых ко всем приложениям, например: «Все классы, связанные со средой разработки, будут содержать аннотацию (@Framework в Java) или атрибут ([Framework] в C#), идентифицирующий принадлежность класса к коду исходной среды разработки». Подкаталоги в каталоге *application* соответствуют конкретному приложению или системному контексту и содержат архитектурные решения, специфичные для этого приложения или системы (в данном примере это приложения ATP и PSTD). Каталог *integration* (интеграция) содержит те ADR-записи, которые касаются обмена данными между приложениями, системами или сервисами. ADR-записи корпоративных архитектурных решений содержатся в каталоге *enterprise* (предприятие), чье название указывает на то, что это глобальные архитектурные решения, касающиеся всех систем и приложений. Пример ADR-записи корпоративного архитектурного решения может выглядеть так: «Весь доступ к системной базе

данных будет только из владеющей ею системы», что не позволяет совместно использовать БД сразу нескольким системам.

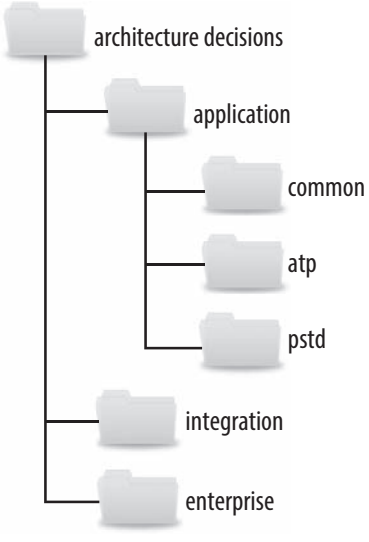


Рис. 19.3. Пример структуры каталогов для хранения ADR

При сохранении ADR в вики-системе (что мы и рекомендуем) применяется такая же структура, что и ранее рассмотренная, где каждая структура каталога представляет собой целевую страницу в системе навигации. Каждая ADR-запись будет представлена в виде отдельной вики-страницы внутри каждой целевой страницы (Application, Integration или Enterprise).

Названия каталогов или целевых страниц, указанные в этом разделе, носят исключительно рекомендательный характер. Каждая компания может выбрать подходящие для нее имена, при условии, что они однозначно воспринимаются всеми командами разработчиков.

ADR как документация

Документирование архитектуры ПО всегда представляло собой весьма непростую тему. Хотя для построения архитектурных диаграмм некоторые стандарты все же появляются (например, стандарт C4 Model[1] от архитектора программного

[1] https://c4model.com/

обеспечения Саймона Брауна (Simon Brown) или стандарт ArchiMate[1] от Open Group), для документирования архитектуры ПО подобных стандартов пока не существует. И тут на первый план выходит ADR.

Записи архитектурных решений могут использоваться в качестве весьма эффективного средства документирования программной архитектуры. Раздел «Контекст» в ADR предоставляет отличную возможность раскрыть конкретную область системы, требующую принятия архитектурного решения. Этот раздел также позволяет дать описание альтернативных вариантов. Наверное, еще более важным является то обстоятельство, что в разделе «Решение» раскрываются причины принятия конкретного решения, что безусловно является лучшей формой документирования архитектуры. Раздел «Последствия» добавляет заключительную часть документации, давая описание дополнительных аспектов конкретного решения, таких, например, как анализ компромиссов при выборе повышения производительности в ущерб масштабируемости.

Использование ADR для введения стандартов

Стандарты вряд ли кому-то нравятся. Зачастую кажется, что их вводят больше для контроля над людьми и над тем, как они что-то делают, чем для чего-то более полезного. Использование ADR для введения стандартов может изменить сложившееся к ним негативное отношение. Например, в разделе ADR «Контекст» дается описание ситуации, вынудившей ввести конкретный стандарт. Раздел ADR «Решение» может использоваться не только для изложения сути стандарта, но и, что более важно, для обоснования необходимости его введения. Это прекрасный способ, позволяющий в первую очередь определить, имеет ли данный стандарт право на существование. Если архитектор не может обосновать необходимость стандарта, то, возможно, этот стандарт не так уж и хорош, чтобы настаивать на его введении. Более того, чем больше разработчиков поймет, зачем вводится стандарт, тем выше шансы, что они ему последуют (и, соответственно, не станут его оспаривать). Раздел ADR «Последствия» — еще одно подходящее место, где архитектор может дать определение пригодности и необходимости стандарта. В этом разделе архитектору следует продумать и задокументировать значение и последствия от принятия создаваемого стандарта. Анализируя последствия, архитектор может в конечном итоге решить, что стандарт все-таки не нужен.

[1] https://www.opengroup.org/archimate-forum/archimate-overview

Пример

В рамках нашего примера «Going, Going, Gone» из главы 7 на с. 128 принималось множество архитектурных решений. Выбор микросервисной архитектуры, управляемой событиями, разбиение пользовательского интерфейса на интерфейс участника и интерфейс организатора торгов, использование протокола передачи данных в реальном времени (Real-time Transport Protocol, RTP) для видеозахвата, применение единого API-уровня и использование обмена сообщениями на основе публикаций и подписок — это только некоторые из десятков архитектурных решений, принятых для данной системы аукционных торгов. Каждое архитектурное решение, каким бы очевидным оно ни было, должно быть задокументировано и обосновано.

На рис. 19.4 показано одно из архитектурных решений, принятых для системы аукционных торгов «Going, Going, Gone», которое задает обмен сообщениями между сервисами захвата ставок (bid capture), создания потока ставок (bid streamer) и отслеживания торгов (bid tracker) на основе публикаций и подписок (publish-and-subscribe, pub/sub).

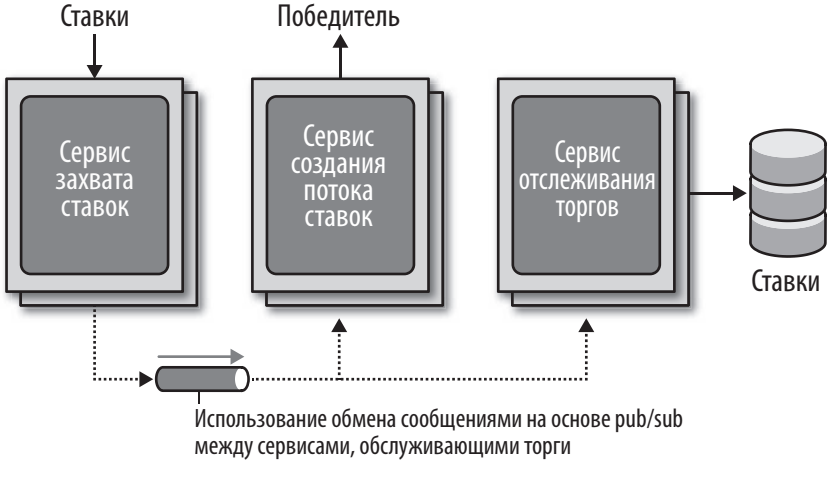


Рис. 19.4. Использование pub/sub между сервисами

ADR этого архитектурного решения может выглядеть приблизительно так (рис. 19.5):

ADR 76. Асинхронный обмен сообщениями на основе pub/sub между сервисами, обслуживающими торги

СТАТУС

Принято

КОНТЕКСТ

Сервис Bid Capture, получив заявку от онлайн-участника торгов или от участника живого аукциона через организатора, должен переслать эту заявку на сервис Bid Streamer и сервис Bidder Tracker. Это можно сделать с помощью асинхронного обмена сообщениями «точка — точка» (p2p), асинхронного обмена сообщениями на основе публикаций и подписок (pub/sub) или с применением REST-технологии через API-уровень онлайн-аукциона.

РЕШЕНИЕ

Между сервисом Bid Capture, сервисом Bid Streamer и сервисом Bidder Tracker будет использоваться асинхронный обмен сообщениями на основе pub/sub.

Сервис Bid Capture не нуждается в какой-либо информации, возвращаемой сервисом Bid Streamer или сервисом Bidder Tracker.

Сервис Bid Streamer должен получать ставки в том же самом порядке, в каком они были приняты сервисом Bid Capture. Использование системы обмена сообщениями и выстраивания очередей автоматически гарантирует для потока нужный порядок следования заявок.

Использование асинхронного обмена сообщениями на основе публикаций и подписок будет способствовать подъему производительности процесса ведения торгов и позволит в дальнейшем расширить подачу информации о торгах.

ПОСЛЕДСТВИЯ

Нам потребуется кластеризация и высокая доступность очередей сообщений.

Внутренние события ставок будут обходить проверки безопасности, выполняемые на API-уровне.

ОБНОВЛЕНИЕ. 14.04.2020 года на заседании наблюдательного совета по архитектуре принято решение, что этот компромисс приемлем, и дополнительные проверки безопасности событий ставок, пересылаемых между этими сервисами, не требуются.

КОМПЛАЕНС

Чтобы убедиться, что между сервисом Bid Capture, сервисом Bid Streamer и сервисом Bidder Tracker применяется обмен сообщениями на основе pub/sub, мы будем периодически вручную делать кодревью и дизайн-ревью.

ПРИМЕЧАНИЯ

Автор: Субашини Наделла

Утверждено: Участниками заседания наблюдательного совета по архитектуре, 14.04.2020

Последнее обновление: 15.04.2020 года, внесено Субашини Наделла

Рис. 19.5. ADR 76. Асинхронный обмен сообщениями на основе pub/sub между сервисами, обслуживающими торги

ГЛАВА 20

Анализ архитектурных рисков

Безупречных архитектур не бывает, и проблемы могут коснуться чего угодно: доступности, масштабируемости или целостности данных. Анализ архитектурных рисков — одно из ключевых направлений в построении архитектуры. Постоянно оценивая риски, архитектор может обнаружить недостатки архитектуры и предпринять нужные действия для их исправления. В этой главе будет представлен ряд ключевых методов и практических приемов, позволяющих классифицировать риски, оценивать и обнаруживать их путем проведения так называемого *риск-штурма* (*risk storming*).

Матрица рисков

Первая задача, возникающая при оценке архитектурного риска, — классифицировать его как низкий, средний или высокий. Эта классификация носит слишком субъективный характер, не позволяющий четко определить, в каких частях архитектуры действительно высокая степень риска, а в каких нет. К счастью, существует матрица рисков, которой архитекторы могут воспользоваться для объективного определения степени риска в конкретной области архитектуры.

В матрице архитектурных рисков (рис. 20.1) используются два измерения: общее влияние риска и вероятность его возникновения. У каждого измерения имеются соответствующие оценки: низкая (1), средняя (2) и высокая (3). Эти числа перемножаются в каждом элементе матрицы, давая объективный числовой показатель степени риска. Числа 1 и 2 (на светло-сером фоне) обозначают низкий риск, числа 3 и 4 (на сером фоне) относятся к среднему риску, а числа 6–9 (на темно-сером фоне) — к высокому риску.

Чтобы понять, как применяется матрица риска, предположим, что есть опасения по поводу доступности центральной базы данных, используемой в приложении.

Рис. 20.1. Матрица определения архитектурных рисков

Сначала оценим влияние: насколько серьезными будут последствия в случае выхода базы данных из строя или потери ее доступности? Здесь архитектор может решить, что риск довольно значительный, оценив его на 3 (средний), 6 (высокий) или 9 (высокий). Но после оценки по второму измерению (вероятности возникновения риска) архитектор понимает, что база данных находится на серверах с высокой степенью доступности в кластерной конфигурации и вероятность, что она станет недоступна, весьма мала. Поэтому пересечение линий серьезного влияния риска и низкой вероятности его возникновения дает общую оценку 3 (средний риск).

При использовании матрицы рисков в первую очередь учитывайте общее влияние, а вероятность возникновения — во вторую.

Оценка рисков

Матрица рисков, рассмотренная в предыдущем разделе, может использоваться для формирования так называемой *оценки рисков (risk assessment)*. Эта оценка представляет собой сводный отчет по общему риску архитектуры с учетом контекстных и значимых оценочных критериев.

Оценка рисков может существенно варьироваться, но в целом она включает в себя риск (полученный из матрицы рисков) по некоторым *оценочным критериям*, связанным с сервисами или предметными областями приложения. Этот

базовый формат отчета по оценке рисков показан на рис. 20.2, где светло-серый фон (с цифрами 1–2) соответствует низкой степени риска, серый (3–4) — средней, а темно-серый (6–9) — высокой. Обычно цветовая кодировка состоит из зеленого (низкая степень риска), желтого (средняя степень риска) и красного (высокая степень риска) цвета, но оттенки серого могут пригодиться для черно-белого отображения и для их нормального восприятия людьми с особенностями зрения.

КРИТЕРИЙ ОЦЕНКИ РИСКОВ	Регистрация клиента	Оформление заказа по каталогу	Выполнение заказа	Отправка заказа	ОБЩИЙ РИСК
Масштабируемость	2	6	1	2	11
Доступность	3	4	2	1	10
Производительность	4	2	3	6	15
Безопасность	6	3	1	1	11
Целостность данных	9	6	1	1	17
ОБЩИЙ РИСК	24	21	8	11	

Рис. 20.2. Пример стандартной оценки рисков

Риск, оцененный по матрице рисков, может накапливаться по критериям риска, а также по сервисной или предметной области. Например, на рис. 20.2 можно заметить, что накопленный риск для целостности данных является наивысшим показателем риска с общим значением 17, а накопленный риск для доступности оценен только в 10 баллов (наименьший показатель). С помощью примера оценки рисков можно также определить относительный риск в каждой предметной области. В данном случае с наибольшим риском сопряжена регистрация клиента, а с наименьшим — выполнение заказа. Затем эти относительные показатели могут отслеживаться, чтобы видеть снижение или повышение степени риска в рамках его конкретной категории или предметной области.

Хотя показанный на рис. 20.2 пример оценки рисков содержит все результаты анализа, в реальности оценку так не презентуют. Для визуализации конкретного

анализа в заданном контексте важно применять фильтрацию. Предположим, к примеру, что архитектор на совещании хочет показать участникам те области системы, которые подвержены наибольшему риску. Вместо представления подробной оценки рисков, показанной на рис. 20.2, можно отфильтровать результаты и показать только области с наибольшим риском (рис. 20.3), улучшив тем самым соотношение сигнал — шум и обрисовав четкую картину состояния системы (хорошее или плохое).

КРИТЕРИЙ ОЦЕНКИ РИСКОВ	Регистрация клиента	Оформление заказа по каталогу	Выполнение заказа	Отправка заказа	ОБЩИЙ РИСК
Масштабируемость		6			6
Доступность					0
Производительность				6	6
Безопасность	6				6
Целостность данных	9	6			15
ОБЩИЙ РИСК	15	12	0	6	

Рис. 20.3. Отфильтрованная оценка рисков, содержащая только наивысшие показатели

Еще одна проблема картины оценки рисков, показанной на рис. 20.2, заключается в том, что такой отчет об оценках дает только статичное представление на конкретное время, на ней не видны какие-либо улучшения или ухудшения ситуации. Иными словами, на рис. 20.2 не показаны направления развития рисков. Отобразить эти направления не так-то просто. Если для индикации направлений воспользоваться стрелками вверх и вниз, то что будут означать эти стрелки? Улучшение или ухудшение? На протяжении многих лет мы задавали людям вопросы, что означает стрелка вверх; примерно 50% опрошенных говорили, что эта стрелка означает ухудшение, а другие 50% — что она означает улучшение. То же самое касается и стрелок вправо-влево. Поэтому при использовании стрелок должно даваться пояснение. Но мы поняли, что и этот вариант не работает. Стоит только прокрутить изображение, выводя пояснение из зоны просмотра, как снова возникает путаница.

Обычно (рис. 20.4) мы используем универсальные символы направления в виде знаков «плюс» (+) и «минус» (–), которые ставятся возле оценки риска. Обратите внимание, что на рис. 20.4 риск снижения производительности при регистрации клиентов оценен как средний (4), а направление показано знаком «минус» (на цветном изображении он будет красного цвета), что свидетельствует об ухудшении ситуации и о движении в сторону повышения риска. А масштабируемость оформления заказа по каталогу имеет высокий показатель риска (6), но со знаком «плюс» (на цветном изображении он будет зеленого цвета), свидетельствующим об улучшении ситуации. Оценки рисков без знаков «плюс» и «минус» показывают, что риск стабилен и ситуация с ним не ухудшается и не улучшается.

КРИТЕРИЙ ОЦЕНКИ РИСКОВ	Регистрация клиента	Оформление заказа по каталогу	Выполнение заказа	Отправка заказа	ОБЩИЙ РИСК
Масштабируемость	2	6 +	1	2	11
Доступность	3	4	2 –	1	10
Производительность	4 –	2 +	3 –	6 +	15
Безопасность	6 –	3	1	1	11
Целостность данных	9 +	6 –	1 –	1	17
ОБЩИЙ РИСК	24	21	8	11	

Рис. 20.4. Индикация направления развития рисков с помощью знаков «плюс» и «минус»

Но иногда даже знаки «плюс» и «минус» кого-то могут сбить с толку. Еще один прием обозначения направления заключается в использовании стрелок с числовой оценкой того риска, к которому направлено движение. Пример на рис. 20.5 не требует пояснения, поскольку направление не вызывает никаких сомнений. Более того, если использовать цветовое оформление (красные стрелки для ухудшения, а зеленые для улучшения ситуации), то это даст еще более четкое представление о направлении движения рисков.

Направление развития рисков можно определять путем постоянных измерений с помощью функций пригодности, ранее рассмотренных в этой книге. Проводя объективный анализ каждого критерия, можно отследить тенденции и получить направление развития.

КРИТЕРИЙ ОЦЕНКИ РИСКОВ	Регистрация клиента	Оформление заказа по каталогу	Выполнение заказа	Отправка заказа	ОБЩИЙ РИСК
Масштабируемость	2	6 ↑4	1	2	11
Доступность	3	4	2 ↓3	1	10
Производительность	4 ↓6	2 ↑1	3 ↓4	6 ↑4	15
Безопасность	6 ↓9	3	1	1	11
Целостность данных	9 ↑6	6 ↓9	1 ↓2	1	17
ОБЩИЙ РИСК	24	21	8	11	

Рис. 20.5. Индикация направления развития рисков с помощью стрелок и чисел

Проведение риск-штурма

Ни один архитектор не в состоянии определить общий риск системы. Причина здесь двоякая. Во-первых, один архитектор может пропустить или не заметить зону риска, а во-вторых, далеко не все архитекторы обладают полноценными знаниями каждой части системы. Вот здесь-то и может прийти на помощь проведение *риск-штурма*.

Риск-штурм — это совместная работа по выявлению архитектурного риска в конкретном направлении. Основные параметры (зоны риска) включают непроверенную технологию, производительность, масштабируемость, доступность (в том числе транзитивных зависимостей), потерю данных, точки отказа и безопасность. В большинстве случаев в риск-штурмах участвуют несколько архитекторов, но разумнее также пригласить главных разработчиков и техлидов. Разработчики предоставят свою оценку архитектурного риска с точки зрения реализации проекта, а также смогут лучше понять весь архитектурный замысел.

Проведение риск-штурма состоит из индивидуальной и совместной части. В первой все участники в индивидуальном порядке (без совместной работы) распределяют степени риска по областям архитектуры, используя матрицу рисков из предыдущего раздела. Эта часть риск-штурма необходима для того, чтобы участники не могли повлиять друг на друга или отвлечь внимание от конкретных областей архитектуры. В совместной части риск-штурма все участники

работают вместе, чтобы прийти к согласию о зонах риска, обсудить степени риска и выработать решения по устранению рисков.

Для проведения обеих частей риск-штурма используется диаграмма архитектуры. Для всесторонних оценок рисков обычно используется комплексная диаграмма, а для риск-штурма в определенных областях приложения — контекстная. Ответственность за актуальность и доступность этих диаграмм для всех участников возлагается на архитектора, проводящего риск-штурм.

Пример архитектуры для иллюстрации процесса риск-штурма показан на рис. 20.6. В этой архитектуре гибкий балансировщик нагрузки под названием Elastic Load Balancer выходит на экземпляры EC2, содержащие веб-серверы (Nginx) и сервисы приложения. Эти сервисы обращаются с вызовами к базе данных MySQL, к кэшу Redis и к базе данных MongoDB, предназначенной для журналирования. Они также обращаются с вызовами к пуш-серверам расширений — Push Expansion Servers. Все серверы расширений, в свою очередь, взаимодействуют с базой данных MySQL, кэшем Redis и со средством журналирования на основе MongoDB.

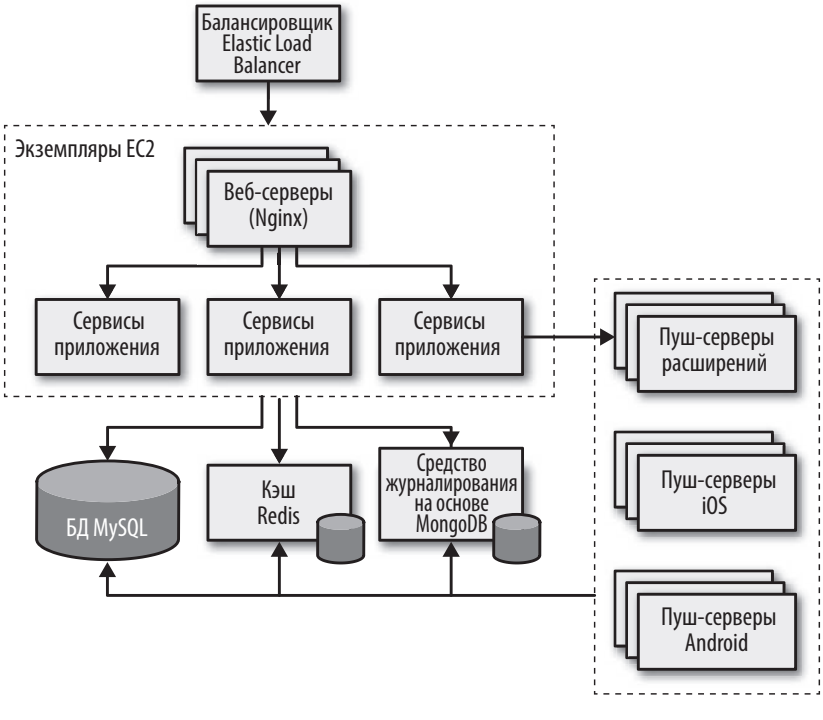


Рис. 20.6. Диаграмма архитектуры для примера риск-штурма

Риск-штурм разбивается на три основных этапа:

1. Выявление.
2. Консенсус.
3. Снижение.

Выявление всегда проводится в индивидуальном порядке без совместной работы, а консенсус и снижение неизменно являются совместной деятельностью всех участников, работающих вместе в одном помещении (в крайнем случае в виртуальном пространстве). Подробности проведения всех трех этапов рассматриваются в следующих разделах.

Выявление

Выявление рисков в риск-штурме проводится индивидуально каждым участником, определяющим зоны риска в архитектуре. Этот процесс состоит из следующих шагов:

1. Архитектор, проводящий риск-штурм, рассылает приглашения всем участникам за один-два дня до начала совместной части работы. Приглашение содержит диаграмму архитектуры (или ссылку на место, где ее можно найти), область проведения риск-штурма (зону риска, анализируемую при проведении данного риск-штурма), дату и место проведения совместной части риск-штурма.
2. Используя матрицу рисков, рассмотренную в первом разделе данной главы, участники в индивидуальном порядке проводят анализ архитектуры и определяют степень риска как низкую (1–2), среднюю (3–4) или высокую (6–9).
3. Участники готовят небольшие стикеры с соответствующими цветами (зеленым, желтым и красным) и записывают число, соответствующее степени риска (взятое из матрицы рисков).

Основная часть работы при проведении риск-штурма связана с анализом одной конкретной области (например, производительности), но иногда из-за проблем с доступностью персонала или со временем в ходе одного риск-штурма может вестись работа сразу по нескольким областям (например, по производительности, масштабируемости и потере данных). В тех случаях, когда анализируется сразу несколько областей, участники записывают название области на стикере рядом с числом, соответствующим степени риска. Предположим, к примеру, что три участника усмотрели риск, связанный с центральной базой данных. Во всех трех случаях степень риска была признана высокой (6), но один участник об-

наружил риск, связанный с доступностью, а два — с производительностью. Эти две области будут обсуждаться отдельно друг от друга.

По возможности старайтесь ограничивать работу при риск-штурме какой-то одной областью. Это позволит участникам сконцентрировать внимание на этой конкретной области и избежать путаницы с выявлением сразу нескольких зон риска в одной и той же части архитектуры.

Консенсус

Этап *консенсуса* в риск-штурме призван выработать общее мнение всех участников в отношении оценки рисков архитектуры. Наиболее эффективно эта работа проводится при наличии большой распечатанной версии диаграммы архитектуры, вывешенной на стене. Вместо нее может использоваться электронная версия диаграммы, показанная на большом экране.

Как только участники прибывают на риск-штурм, они тут же размещают свои стикеры, созданные в результате индивидуального анализа, на диаграмме архитектуры. Если используется электронная версия, архитектор, проводящий риск-штурм, опрашивает каждого участника и размещает оценки рисков на диаграмме в электронном виде (рис. 20.7).

После размещения всех стикеров начинается совместная часть риск-штурма. В ней все подчинено анализу областей единой командой и выработке консенсуса в квалификации рисков. Обратите внимание, что в архитектуре, показанной на рис. 20.7, выявлены сразу несколько зон риска:

1. Два участника в индивидуальном порядке определили степень риска балансировщика Elastic Load Balancer как среднюю (3), а один участник поставил высокую оценку (6).
2. Один участник в индивидуальном порядке определил степень риска пушсерверов расширений как высокую (9).
3. Три участника в индивидуальном порядке определили степень риска базы данных MySQL как среднюю (3).
4. Один участник в индивидуальном порядке определил степень риска кэша Redis как высокую (9).
5. Три участника определили степень риска средства ведения журнала на основе MongoDB как низкую (2).
6. Все остальные части архитектуры не рассмотрены в качестве подверженных каким-либо рискам, поэтому никаких стикеров на них нет.

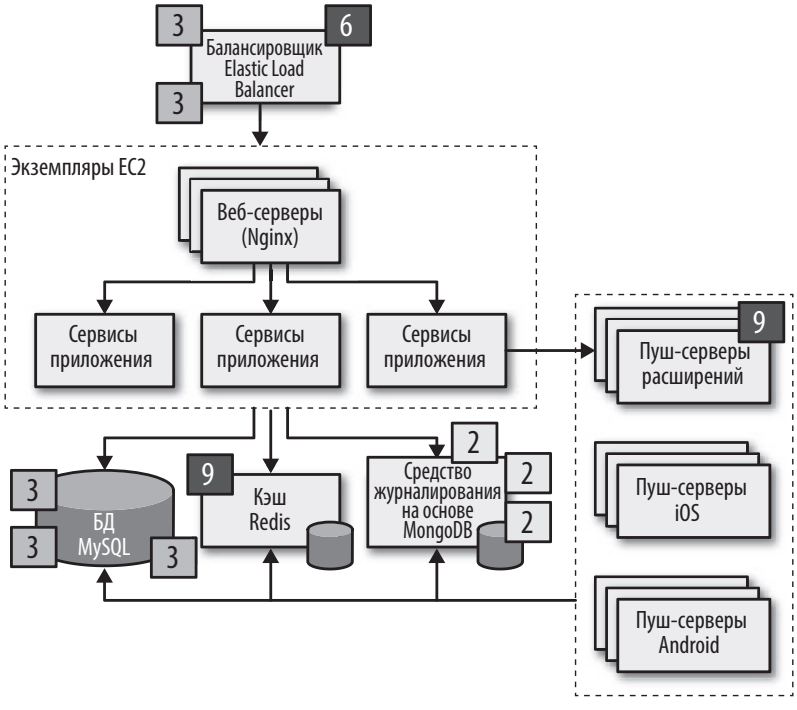


Рис. 20.7. Исходное выявление зон риска

Пункты 3 и 5 в предыдущем списке не нуждаются в дальнейшем рассмотрении, поскольку все участники согласились со степенями и квалификациями рисков. Но следует заметить разницу мнений в пункте 1 данного списка, а также то, что в пунктах 2 и 4 фигурирует только один участник, выявивший риск. Эти пункты следует обсудить в процессе риск-штурма.

В пункте 1 списка показано, что два участника в индивидуальном порядке определили степень риска балансировщика Elastic Load Balancer как среднюю (3), а один участник оценил ее высоко (6). В таком случае эти два участника спрашивают третьего, почему он дал такую высокую оценку. Предположим, что третий участник ответил: если Elastic Load Balancer выйдет из строя, недоступной станет вся система. Это действительно так, и фактически общее влияние этого риска довольно велико, но остальные два участника убеждают третьего, что вероятность такого сбоя весьма мала. После бурного обсуждения третий участник соглашается, снизив степень риска до средней (3). Однако вполне возможно, что первый и второй участники не видели конкретный аспект риска в Elastic Load Balancer, обнаруженный третьим участником, поэтому возникла потребность в сотрудничестве в рамках проведения риск-штурма.

А теперь рассмотрим пункт 2 из списка, где один участник в индивидуальном порядке определил степень риска пуш-серверов расширений как высокую (9), в то время как ни один другой участник вообще не выявил там никакого риска. В таком случае все остальные спрашивают, почему была дана такая высокая оценка. Этот участник говорит, что, исходя из его печального опыта, пуш-серверы расширений, имеющиеся и в данной архитектуре, постоянно сбоят под высокой нагрузкой. Этот пример показывает, насколько высока польза от проведения риск-штурмов, ведь без привлечения этого участника никто бы и не заметил высокой степени риска (разумеется, до того момента, пока продукт не пошел бы в производство!).

Не менее интересная история складывается вокруг пункта 4. Один участник определил для кэша Redis высокую степень риска (9), а другие вообще не увидели какой-либо угрозы. Остальные участники спрашивают о причине такой высокой оценки, а в ответ слышат вопрос: «А что такое кэш Redis?» Оказывается, этот участник вообще не знал, что такое Redis, и поэтому дал высокую оценку риску в данной области.

Всегда ставьте для непроверенных или неизвестных технологий высокую оценку степени риска (9), поскольку для этой области матрица рисков не может использоваться.

Пример с пунктом 4 нашего списка показал, насколько мудрым (и важным) шагом является приглашение разработчиков на риск-штурмы. Это не только позволяет разработчикам глубже изучать архитектуру, но и предоставляет архитектору ценную информацию в отношении общего риска в том случае, если один из участников (разработчик одной из команд в нашем примере) не знаком с той или иной технологией.

Этот процесс продолжается до тех пор, пока все участники не согласятся с выявленными зонами риска. Как только все стикеры будут согласованы, этот этап заканчивается и можно приступать к следующему. Окончательный результат показан на рис. 20.8.

Снижение рисков

После того как все участники придут к согласию насчет квалификации зон риска в архитектуре, начинается заключительное и наиболее важное действие — *снижение рисков (risk mitigation)*. В архитектуре оно обычно достигается за счет изменений или улучшений в определенных ее областях, которые в других обстоятельствах могли бы считаться не требующими изменений.

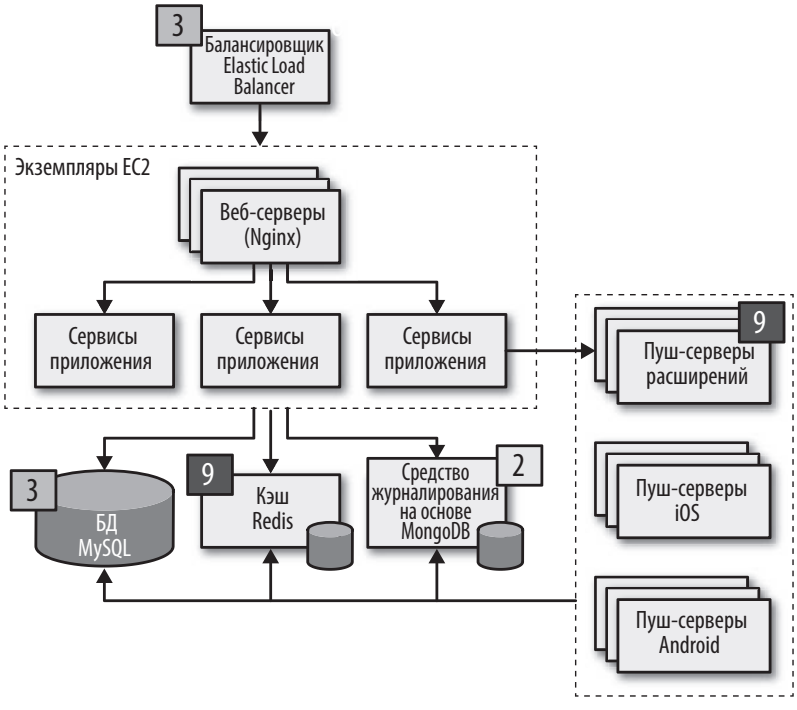


Рис. 20.8. Консенсус по зонам риска

Это коллективное действие заключается в поиске путей сокращения или исключения рисков, выявленных на предыдущем этапе. Бывает и так, что в результате выявления рисков полной замене подлежит вся исходная архитектура, а бывает, что требуется рефакторинг, например, добавить очередь для обратного давления, чтобы избавиться от узкого места, сформировавшегося из-за низкой пропускной способности.

Независимо от тех изменений, которые требуется внести в архитектуру, это действие обычно сопряжено с дополнительными затратами. Поэтому ключевые стейкхолдеры обычно решают, не перевешивают ли эти затраты выявленный риск. Предположим, к примеру, что в результате проведения риск-штурма степень риска в области доступности центральной базы данных была определена как средняя (4). В данном случае участники согласились, что с помощью кластеризации БД в сочетании с разбиением единой базы на отдельные физические БД можно минимизировать эту проблему. Но наряду с существенным уменьшением степени риска реализация данного решения обойдется в 20 000 долларов. После этого архитектор проводит встречу с ключевыми бизнес-стейкхолдерами, чтобы обсудить компромиссы. В ходе переговоров владелец бизнеса решает, что цена

слишком высока и что стоимость не перевешивает риск. Архитектор не сдается и предлагает другой подход: «А что, если обойтись без кластеризации и разбить базу данных на две части? Тогда стоимость упадет до 8000 долларов, а основная часть риска все же снизится». В этом случае акционер соглашается.

Предыдущий сценарий показывает, какое влияние может оказать риск-штурм не только на архитектуру в целом, но и на переговоры архитекторов с бизнес-стейкхолдерами. Риск-штурм в сочетании с оценками, описанными в начале этой главы, является отличным средством выявления и отслеживания рисков, улучшения архитектуры и ведения переговоров между ключевыми стейкхолдерами.

Анализ рисков пользовательских историй в методологии Agile

Риск-штурм может быть использован не только для архитектуры, но и для других аспектов разработки программного обеспечения. Например, в ходе предварительной подготовки пользовательской истории (user story) мы использовали риск-штурм для определения общего риска незавершения выполнения требований заказчика (пользовательского сценария) в рамках заданной Agile-итерации (и, следовательно, для получения общей оценки риска этой итерации). При использовании матрицы рисков степень риска пользовательского сценария может быть выявлена по первому измерению (общее влияние, если история не будет завершена в течение итерации) и второму измерению (вероятности того, что история не будет завершена). Используя ту же самую матрицу архитектурных рисков для сценариев, команды разработчиков могут выявлять сценарии с высокой степенью риска, тщательно их отслеживать и присваивать им высокий приоритет.

Примеры риск-штурма

Чтобы проиллюстрировать эффективность проведения риск-штурмов и показать, как они могут улучшить общую архитектуру системы, рассмотрим пример: колл-центр для поддержки медсестер, консультирующих пациентов по вопросам состояния здоровья. К такой системе будут предъявляться следующие требования:

- В системе будет использоваться сторонний механизм диагностики, который обрабатывает задаваемые вопросы и дает указания медсестрам или пациентам по возникшим медицинским проблемам.

- Пациенты могут либо позвонить в колл-центр, чтобы пообщаться с медсестрой, либо воспользоваться веб-сайтом самообслуживания, напрямую обращаясь к диагностической системе.
- Система должна одновременно обслуживать до 250 медсестер и до сотен тысяч пациентов по всей стране, обращающихся к веб-сайту самообслуживания.
- Медсестры могут получать доступ к медицинским картам пациентов через систему обмена медицинскими картами, но пациенты не имеют доступа к своим собственным медкартам.
- Система должна соответствовать требованиям HIPAA[1] в отношении медкарт. То есть важно, чтобы к медкартам не имел доступа никто, кроме медсестер.
- С помощью системы необходимо принимать соответствующие меры при вспышках массовых заболеваний в сезоны простуды и гриппа.
- Маршрутизация вызовов к медсестрам зависит от профиля той или иной медсестры (например, при потребностях общения на двух языках).
- Сторонний механизм диагностики может обрабатывать до 500 запросов в секунду.

Архитектор системы создал высокоуровневую архитектуру, показанную на рис. 20.9. В ней имеются три отдельных пользовательских веб-интерфейса: один для самообслуживания, один для медсестер, принимающих звонки, и один для административного персонала, чтобы добавлять и поддерживать профили медсестер и параметры конфигурации. Та часть системы, которая относится к колл-центру, состоит из компонента приема вызовов и маршрутизатора вызовов, направляющего вызовы следующей доступной медсестре на основе ее профиля (обратите внимание, как маршрутизатор вызовов осуществляет доступ к центральной базе данных для получения информации о профиле медсестры). Центральным элементом этой архитектуры является API-шлюз системы диагностики, выполняющий проверку безопасности и направляющий запрос в соответствующий бэкенд-сервис.

В системе имеется четыре основных сервиса: сервис управления историями болезней, сервис управления профилями медсестер, интерфейс системы обмена медицинскими картами и внешний сторонний механизм диагностики. Во всех обменах данными, за исключением собственных протоколов обращения к внешним системам и сервисам колл-центра, используется REST-технология.

[1] Акт о передаче и защите данных учреждений здравоохранения (действующий в США). — *Примеч. ред.*

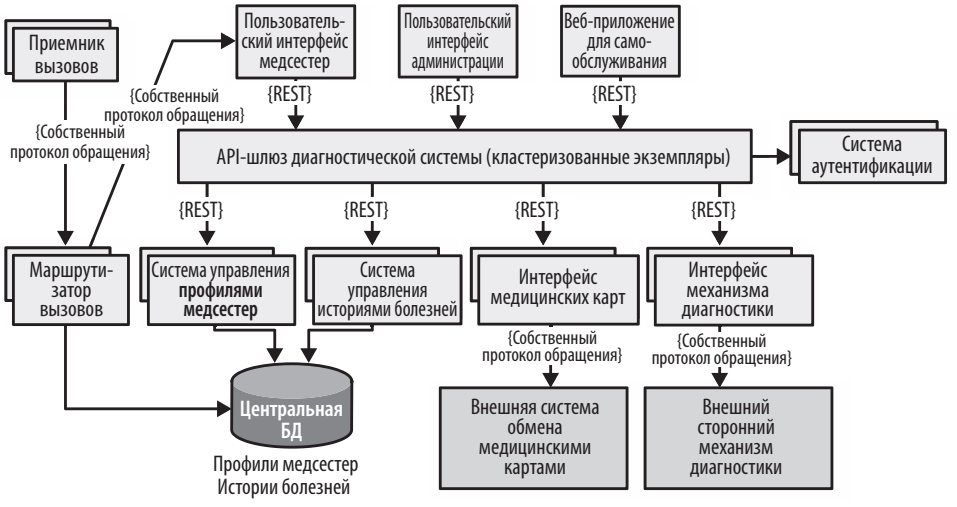


Рис. 20.9. Высокоуровневая архитектура для диагностической системы с участием медсестер

Архитектор пересматривал эту архитектуру много раз и считает, что она готова к реализации. Чтобы проверить себя, изучите требования, предъявляемые к системе, и диаграмму архитектуры, показанную на рис. 20.9, и попытайтесь определить степень риска в данной архитектуре с позиций доступности, адаптируемости и безопасности. После выявления степени риска решите, какие изменения нужно будет внести в архитектуру для снижения этого риска. В следующих разделах содержатся сценарии, которые можно использовать для сравнения с вашими действиями.

Доступность

В ходе первой сессии риск-штурма архитектор решил сосредоточиться на доступности, поскольку доступность системы играет ключевую роль для ее успешного функционирования. После действий по выявлению рисков и совместных действий участников в рамках риск-штурма с помощью матрицы рисков были определены следующие зоны риска (рис. 20.10):

- Использование центральной базы данных определено как фактор высокого риска (6) из-за большого общего влияния (3) и средней вероятности возникновения (2).
- Для доступности механизма диагностики определена высокая степень риска (9) из-за большого общего влияния (3) и неизвестной вероятности возникновения (3).

- Для доступности системы обмена медицинскими картами определена низкая степень риска (2), поскольку для работы системы этот компонент не является обязательным.
- Остальные части системы не выявлены в качестве зон риска в отношении доступности из-за наличия нескольких экземпляров каждого сервиса и кластеризированного API-шлюза.

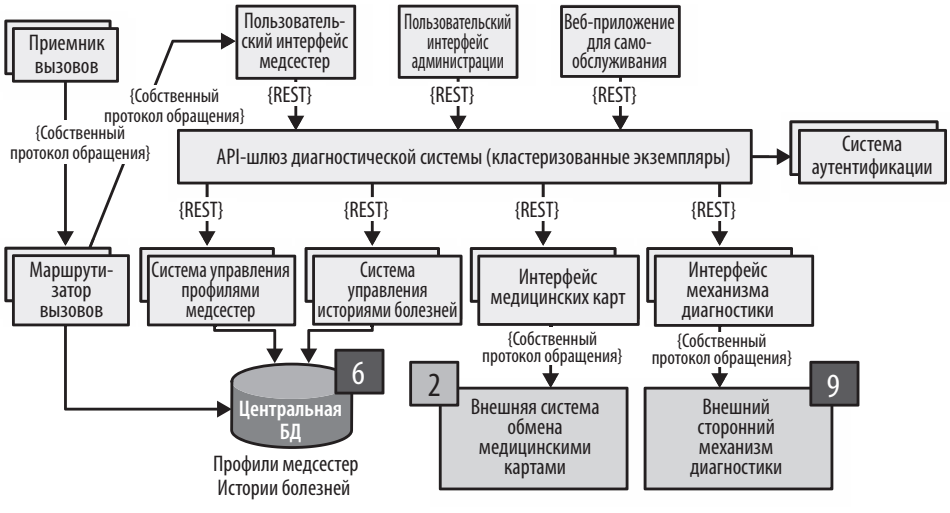


Рис. 20.10. Зоны риска для доступности

В ходе проведения риск-штурма все участники согласились, что в случае выхода из строя базы данных медсестры могут записывать истории болезней вручную, а вот если БД не будет доступна, маршрутизатор вызовов не сможет функционировать. Чтобы устранить риски, связанные с БД, участники решили разбить единую физическую базу данных на две отдельные: одна кластеризованная база данных будет содержать информацию о профилях медсестер, а другая будет предназначена для историй болезней. Это изменение архитектуры не только решило проблему доступности БД, но и помогло обезопасить истории болезней от доступа со стороны администрации. Еще одним вариантом снижения этого риска могло бы стать кэширование информации профиля медсестры в маршрутизаторе вызовов. Но поскольку о реализации маршрутизатора вызовов ничего известно не было и он может быть сторонним продуктом, участники применили подход, связанный с базами данных.

Снизить риск ухудшения доступности внешних систем (механизма диагностики и системы обмена медкартами) гораздо труднее, поскольку отсутствует контроль

над этими системами. Один из способов снижения подобного риска — выяснить, имеются ли для каждой из этих систем опубликованные соглашения об уровне обслуживания (service-level agreement, SLA) или о целях уровня обслуживания (service-level objective, SLO). SLA, как правило, представляет собой договорное соглашение и имеет обязательную юридическую силу, а SLO — нет. Основываясь на том, что ему удалось узнать, архитектор обнаружил, что согласно SLA для механизма диагностики этот механизм будет гарантированно доступен в 99,99% случаев обращений (что соответствует 52,6 минуты простоя в год), а система обмена медкартами будет гарантированно доступна в 99,9% случаев обращений (что соответствует 8,77 часа простоя в год). С учетом относительности риска этой информации было достаточно для того, чтобы посчитать выявленный риск устраненным.

Соответствующие изменения, внесенные в архитектуру после проведения этого риск-штурма, показаны на рис. 20.11. Обратите внимание на то, что теперь используются две базы данных, а также на то, что данные SLA-соглашений отражены на диаграмме архитектуры.

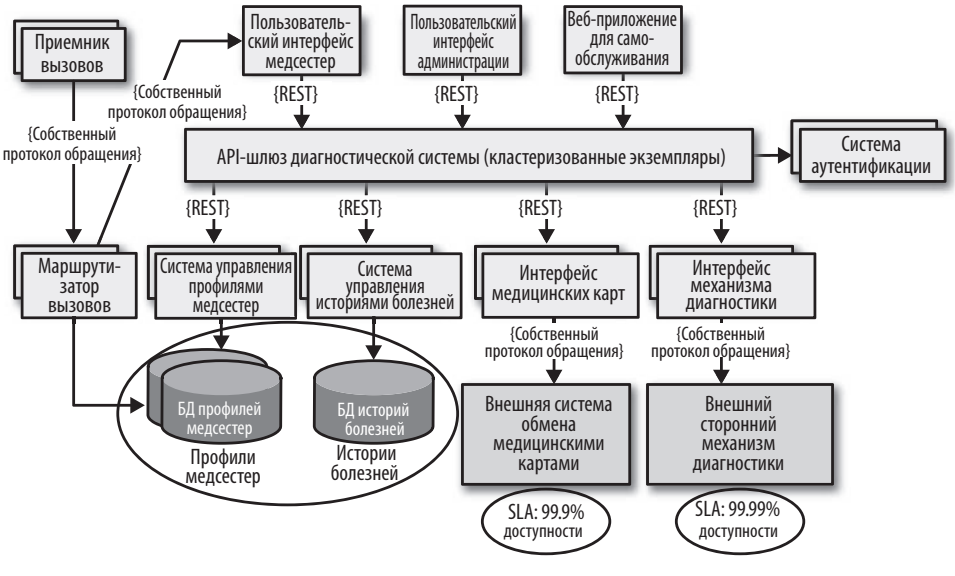


Рис. 20.11. Архитектурные изменения, направленные на устранение риска доступности

Адаптируемость

При проведении второго риск-штурма архитектор решил сосредоточиться на адаптируемости, то есть на способности справляться с взрывным ростом пользовательской нагрузки (которую также называют переменной масштабируемо-

стью). Хотя в системе всего лишь 250 медсестер (а большинство услуг обеспечивается за счет автоматического управления системой), не только медсестры, но и сервис самообслуживания может иметь доступ к механизму диагностики, что существенно увеличивает количество запросов к интерфейсу диагностики.

Участники риск-штурма были озабочены вспышками заболеваний и сезонами эпидемий гриппа, во время которых предполагаемая нагрузка на систему значительно возрастает. В ходе проведения риск-штурма все участники выявили высокую степень риска в области интерфейса механизма диагностики (9). Участники подсчитали, что при ожидаемой пропускной способности интерфейс механизма диагностики не сможет справиться даже с 500 запросами в секунду, особенно с текущим состоянием архитектуры, при котором в качестве протокола интерфейса используется REST-технология.

Одним из путей устранения этого риска станет использование асинхронных очередей (обмена сообщениями) между API-шлюзом и интерфейсом механизма диагностики, чтобы обеспечить точку обратного давления на случай, если механизм диагностики застопорится. Прием вполне подходящий, но он все же не снижает степень риска, поскольку медсестры (а также пациенты на самообслуживании) будут дожидаться ответа от механизма диагностики слишком долго, и соответствующие запросы будут, скорее всего, сброшены по истечении времени ожидания. Использование паттерна, известного как Ambulance[1], даст медсестрам более высокий уровень приоритета по сравнению с теми, кто обращается к механизму самостоятельно. Для этого понадобятся два канала сообщений. Хотя эта технология и хорошая, но в ней все еще нет времени учета ожидания. Участники решили, что в дополнение к технологии использования очередей, призванной обеспечить обратное давление, нужно воспользоваться кэшированием конкретных диагностических вопросов, связанных со вспышками заболеваемости, что избавит вызовы в период вспышек заболеваний и эпидемий гриппа от необходимости обращения к интерфейсу механизма диагностики.

Соответствующие архитектурные изменения показаны на рис. 20.12. Обратите внимание, что в дополнение к двум каналам очередей (одному для обращений со стороны медсестер, а второму для обращений со стороны пациентов на самообслуживании) появился новый сервис под названием *сервер кэша запросов при вспышках заболеваемости*, который обрабатывает все запросы, связанные с вопросами по конкретной вспышке или по эпидемии гриппа. При наличии такой архитектуры ограничительный фактор (в виде вызовов, направляемых к механизму диагностики) будет удален, что позволит одновременно принимать десятки тысяч запросов. Без проведения риск-штурма этот риск мог остаться

[1] https://www.developertoarchitect.com/lessons/lesson56.html

невыявленным до тех пор, пока не возникла бы вспышка заболеваемости или не случилась сезонная эпидемия гриппа.

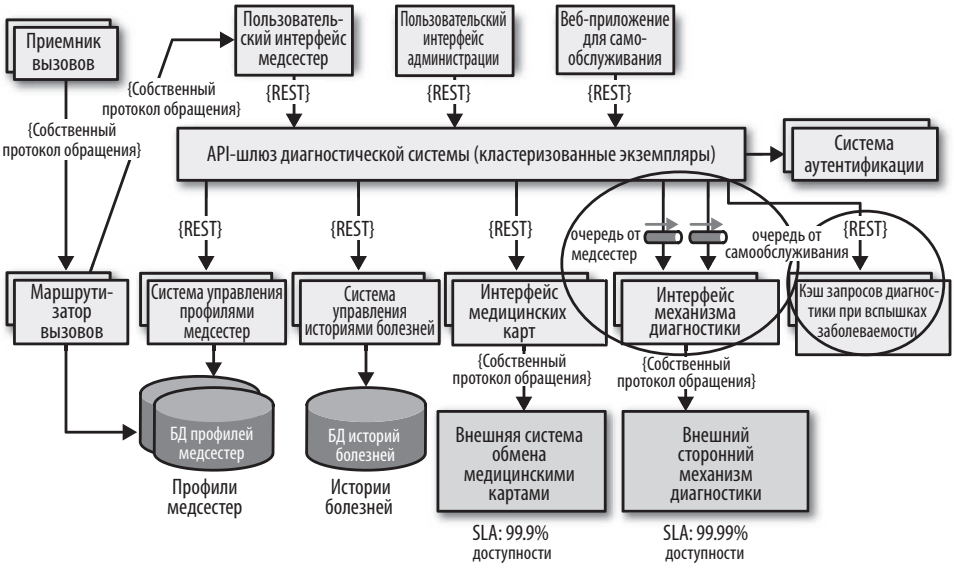


Рис. 20.12. Изменения архитектуры для устранения риска в области адаптируемости

Безопасность

Воодушевленный результатами и успешным проведением первых двух риск-штурмов, архитектор решает провести заключительный риск-штурм в отношении еще одного важного архитектурного свойства, которое должно поддерживаться в системе для обеспечения ее успешной работы, — безопасности. В соответствии с нормативными требованиями HIPAA, доступ к медицинской документации через интерфейс обмена медицинской информацией должен быть безопасным, позволяя получать доступ к медицинским картам только медсестрам. Архитектор считает, что благодаря проверкам безопасности в API-шлюзе (аутентификации и авторизации) здесь нет никаких проблем, но все же интересуется, найдут ли участники риск-штурма какие-либо другие элементы риска безопасности.

В ходе риск-штурма все участники выявили высокую степень риска (6) в API-шлюзе диагностической системы. Основанием для такой оценки послужила высокая степень вероятности того, что административные сотрудники или

пациенты на самообслуживании получат доступ к медкартам (3) в сочетании со средней вероятностью возникновения такого риска (2). Вероятность не оценивалась высоко, поскольку каждый API-вызов сопровождается проверкой безопасности, но она все же получила среднюю оценку, так как все вызовы (самообслуживания, со стороны администрации и со стороны медсестер) проходят через один и тот же API-шлюз. Архитектор, давший риску только низкую оценку (2), в ходе выработки консенсуса при проведении риск-штурма убедился, что риск действительно велик и его следует снизить.

Все участники согласились с тем, что наличие отдельных API-шлюзов для каждого типа пользователей (администраторов, пациентов, самостоятельно получающих диагностику, и медсестер) не позволит вызовам ни от пользовательского веб-интерфейса администратора, ни от пользовательского веб-интерфейса пациентов на самообслуживании достигать интерфейса системы обмена медкартами. Архитектор согласился с доводами и создал окончательный вариант архитектуры, показанный на рис. 20.13.

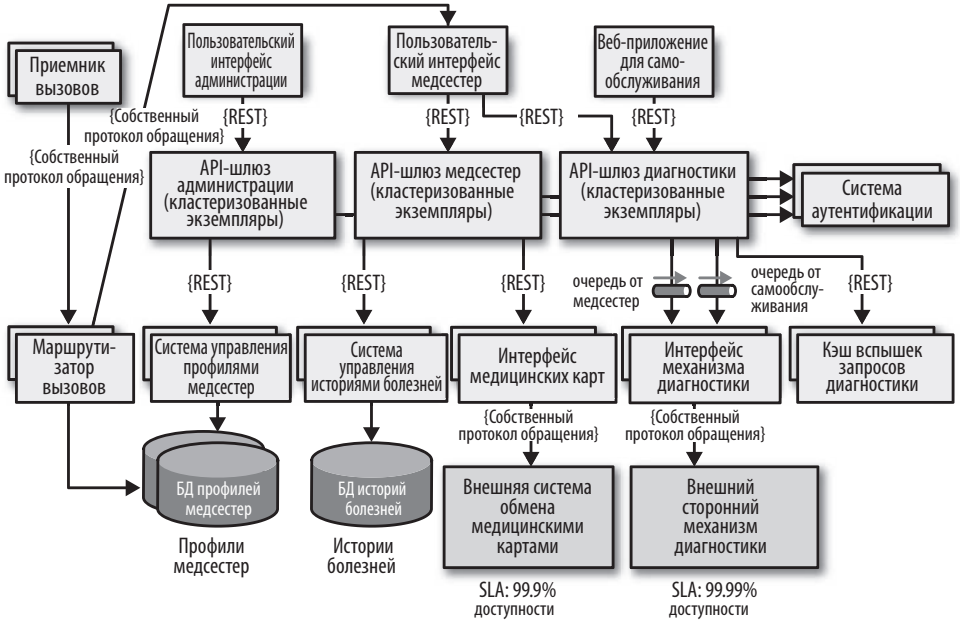


Рис. 20.13. Окончательные изменения архитектуры для устранения рисков безопасности

Предыдущий сценарий служит иллюстрацией эффективности риск-штурма. За счет сотрудничества с другими архитекторами, разработчиками и ключе-

выми стейкхолдерами по определению областей риска, жизненно важных для успешной работы системы, выявляются зоны риска, которые иначе остались бы незамеченными. Сравните изображения на рис. 20.9 и 20.13 и заметьте, что архитектура до проведения риск-штурма существенно отличается от архитектуры после его проведения. Эти важные изменения касаются решения проблем доступности, адаптируемости и безопасности в архитектуре.

Риск-штурм не одноразовая акция, скорее это непрерывный процесс, проводимый на протяжении всей жизни любой системы с целью выявления и сведения к минимуму зон риска еще до того, как они проявятся в продакшене. Частота проведения риск-штурмов зависит от множества факторов, включая частоту внесения изменений, работы по реструктуризации архитектуры и поэтапную разработку архитектуры. Обычно проведению риск-штурма подвергается какая-либо конкретная область после добавления важной функции или в конце каждой итерации.

ГЛАВА 21

Составление диаграмм и проведение презентаций архитектуры

Начинающие архитекторы часто говорят о том, насколько они были удивлены разнообразием деятельности, выходящей за рамки технических знаний и опыта. В частности, важнейшим фактором успешной работы является эффективная коммуникация. Какими бы блестящими ни были технические идеи архитектора, если он не может убедить руководство в необходимости их финансирования, а разработчиков — в важности их воплощения в жизнь, его гениальные замыслы так и останутся на бумаге.

Составление диаграмм и презентаций — это два важнейших гибких навыка. Хотя каждому из них посвящены целые книги, мы рассмотрим несколько конкретных и весьма важных моментов, которые стоит знать.

Эти две темы рассматриваются в комплексе из-за схожести характеристик: обе применяются для визуализации архитектурного видения, но разными средствами. Однако репрезентативная согласованность — это то, в чем они должны совпадать.

Когда дается визуальное представление архитектурного замысла, его создатель зачастую обязан показать различные виды архитектуры. Например, архитектор, скорее всего, даст общий вид всей архитектурной топологии, а затем препарирует отдельно взятые части, чтобы углубиться в подробности дизайна. Но если архитектор станет показывать часть без указания ее местонахождения в общей архитектуре, это запутает зрителей. *Репрезентативная согласованность* — это принцип, согласно которому всегда следует показывать взаимосвязь между частями архитектуры (либо на диаграммах, либо в виде презентаций).

К примеру, если архитектор хочет детально продемонстрировать взаимоотношения плагинов из ката Silicon Sandwiches, то он сначала покажет топологию архитектуры в целом, а затем углубится в структуру плагинов, показывая взаимоотношения между ними (рис. 21.1).

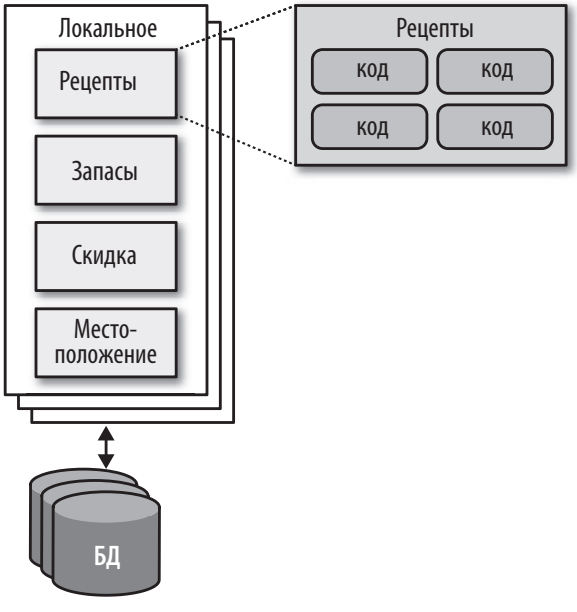


Рис. 21.1. Использование репрезентативной согласованности для обозначения контекста на более крупной диаграмме

Благодаря применению репрезентативной согласованности, зрители лучше воспринимают презентацию и понимают область применения представляемых средств.

Составление диаграмм

Топология архитектуры важна как для архитекторов, так и для разработчиков, поскольку она отражает общую структуру системы, а также способствует единому пониманию всех специалистов команды. Поэтому архитекторы должны оттачивать свои навыки составления диаграмм до остроты бритвенного лезвия.

Инструментарий

В настоящее время доступен богатый набор весьма мощных инструментов для составления диаграмм, и архитектору следует досконально освоить работу с выбранным инструментом. Однако не стоит забывать и о грубых набросках, особенно на ранней стадии процесса проектирования.

Создание первоначальных небрежных эскизов спасает архитекторов от чрезмерной привязанности к тому, что ими создается, то есть от попадания в антипаттерн, который мы назвали *иррациональной привязанностью к артефакту* (*Irrational Artifact Attachment*).

> **ИРРАЦИОНАЛЬНАЯ ПРИВЯЗАННОСТЬ К АРТЕФАКТУ**
>
> ...это прямо пропорциональная зависимость между иррациональной привязанностью человека к какому-либо артефакту и тем временем, которое затрачивается на его создание. Если архитектор два часа рисует красивую диаграмму с помощью Visio, у него прослеживается иррациональная привязанность к этому артефакту, которая примерно пропорциональна количеству затраченного времени, это также означает, что он будет больше привязан к той диаграмме, на которую уйдет не два, а целых четыре часа.
>
> Одно из преимуществ облегченного подхода, используемого в разработке ПО по Agile-технологии, заключается в том, что создание артефактов происходит в то время, когда в них возникает реальная необходимость, и по возможности без разведения всяческих церемоний (это помогает объяснить особую приверженность Agile-адептов к карточкам и стикерам). Использование простейших инструментов позволяет специалистам команды разработчиков избавляться от всего неправильного, освобождая время для проведения экспериментов и позволяя им раскрыть истинную природу артефакта путем проверки, сотрудничества и обсуждения.

Для демонстрации сюжета с любимым всеми архитекторами фото с сотового телефона, на котором изображена белая доска (с неизменным записанным на ней требованием «Не стирать!»), используется не сама белая доска, а планшет, подключенный к потолочному проектору. Этот вариант предлагает ряд существенных преимуществ. Во-первых, холст планшета ничем не ограничен и способен вместить в себя столько рисунков, сколько может понадобиться команде разработчиков. Во-вторых, он позволяет проводить операции копирования и вставки

при отработке сценариев типа «а что, если...», при которых на обычной белой доске забивается все исходное изображение. В-третьих, изображения, попавшие в планшет, уже оцифрованы и избавлены от неизбежных бликов, связанных с фотографированием белых досок камерой сотового телефона.

В конечном счете архитектору все же придется создать красивую диаграмму, используя специализированный инструмент, но до этого следует убедиться, что команда разработчиков выполнила достаточное количество итераций проекта и можно тратить время на диаграммы.

Эффективные средства составления диаграмм имеются на каждой платформе. Мы не станем расхваливать одни средства в ущерб другим (для рисования диаграмм, используемых в этой книге, мы вполне успешно обошлись применением OmniGraffle[1]), но архитекторы должны заняться поиском инструментов, обладающих хотя бы следующими основными возможностями:

Слои

Большинство графических пакетов поддерживают слои, и архитекторы должны как следует разобраться, как этим пользоваться. Слой позволяет логически связать группы элементов; можно скрывать или показывать отдельно взятые слои. Используя слои, архитектор может создать исчерпывающую диаграмму и при этом скрыть лишние подробности, если в них пока нет необходимости. Кроме того, слои помогают постепенно создавать изображения для последующих презентаций (см. далее раздел «Постепенное выстраивание» на с. 376).

Трафареты и шаблоны

Трафареты позволяют архитектору создать библиотеку часто используемых визуальных компонентов, из которых обычно составляются все основные фигуры. Например, в этой книге читатели видели стандартные изображения таких элементов, как микросервисы, имеющиеся в виде отдельных элементов в авторском трафарете. Создание трафарета для часто применяемых шаблонов и артефактов в рамках своей организации позволяет достичь однородности архитектурных диаграмм и дает возможность архитектору быстро создавать новые диаграммы.

Магниты

Многие инструменты рисования предлагают помощь при отрисовке линий между фигурами. Магниты представляют собой те места на этих фигурах,

[1] https://www.omnigroup.com/omnigraffle

где происходит автоматическая привязка линий, помогают в автоматическом выравнивании и т. п. Некоторые инструменты позволяют архитектору добавлять дополнительные магниты или создавать свои собственные для настройки внешнего вида связующих линий на диаграммах.

Кроме указанных полезных функций, инструмент, разумеется, должен поддерживать линии, цвета и другие визуальные артефакты, а также возможность экспорта в разнообразных форматах.

Стандарты составления диаграмм: UML, C4 и ArchiMate

Для технических диаграмм в сфере разработки ПО существует несколько формальных стандартов.

UML

Унифицированный язык моделирования (Unified Modeling Language, UML) был стандартом, объединившим три конкурировавшие философии проектирования, сосуществовавшие в 1980-х годах. Предполагалось, что он будет лучшим из всего имеющегося, но, как и многое разработанное комитетом, он не смог оказать существенного влияния (за исключением организаций, где его использование было обязательным).

Архитекторы и разработчики все еще используют UML-диаграммы классов и последовательностей для передачи структуры и рабочего процесса, но большинство других типов UML-диаграмм уже вышли из употребления.

C4

C4 представляет собой технологию составления диаграмм, разработанную Саймоном Брауном (Simon Brown) для устранения недостатков UML и модернизации используемого в нем подхода. Четыре «C» в C4 расшифровываются следующим образом:

Context (Контекст)

Представляет весь контекст системы, включая роли пользователей и внешние зависимости.

Container (Контейнер)

Физические (а также довольно часто логические) границы развертывания и контейнеры внутри архитектуры. Это представление может послужить

весьма веским поводом для проведения совещаний архитекторов со специалистами по сопровождению программных средств.

Component (Компонент)

Представление системы с позиции компонентов; оно наиболее точно соответствует взгляду архитектора на систему.

Class (Класс)

В C4 используется тот же стиль диаграмм классов, что и в UML, который вполне эффективен и поэтому не требует замены.

Если компания ищет средство стандартизации для составления диаграмм, то C4 — неплохой вариант. Но как и все технические инструменты, эта технология не всегда способна отразить все виды дизайна, которые могут быть задуманы архитектором. C4 больше подходит для монолитных архитектур, где иерархические отношения между контейнером и компонентом могут быть разными, и меньше — для таких распределенных архитектур, как микросервисы.

ArchiMate

ArchiMate (чье название составлено из частей слов Arch*itecture-Ani*mate) является языком с открытым исходным кодом для моделирования корпоративной архитектуры. Он поддерживает описания, анализ и визуализацию архитектуры внутри и между предметными областями бизнеса. ArchiMate представляет собой технический стандарт от Open Group и предлагает несложный язык моделирования экосистемы предприятия. Цель ArchiMate — сохранять предельную лаконичность, а не охватывать все граничные случаи. В силу этих особенностей он приобрел популярность у многих архитекторов.

Рекомендации по составлению диаграмм

Независимо от того, каким языком моделирования пользуется архитектор, своим собственным или одним из формальных, он должен сформировать свой стиль составления диаграмм и не стесняясь заимствовать те представления, которые ему кажутся особенно эффективными. Приведем несколько основных рекомендаций по составлению технических диаграмм:

Заголовки

Нужно убедиться в том, что все элементы, показанные на диаграмме, имеют заголовки или же хорошо известны целевой аудитории. Чтобы «прикрепить» заголовки к тем элементам, для которых они предназначены, и рационально ис-

пользовать имеющееся пространство, применяйте повороты и другие способы размещения текста.

Линии

Линии должны быть достаточной толщины, чтобы быть заметными. Если линии показывают информационный поток, используйте стрелки, обозначающие однонаправленный или двусторонний трафик. Семантика может быть предложена различными типами наконечников стрелок, но архитекторы должны сохранять последовательность в их применении.

Один из немногих стандартов, действующих в отношении архитектурных диаграмм, заключается в том, что сплошные линии указывают на синхронный, а пунктирные — на асинхронный обмен данными.

Фигуры

Хотя свои стандартные фигуры имеются у всех рассмотренных формальных языков моделирования, в мире разработки ПО не существует общепринятых стандартных фигур. Поэтому каждый архитектор стремится создать собственный набор фигур, порой распространяя его по всей организации, чтобы ввести некий стандартный язык.

Для обозначения развертываемых артефактов мы отдаем предпочтение трехмерным «коробкам», а для обозначения контейнеров — прямоугольникам, но каких-либо конкретных установок, кроме этих, у нас нет.

Надписи (метки)

Архитектор должен помечать надписями каждый элемент, присутствующий на диаграмме, особенно если есть шанс, что он будет не понят зрителями.

Цвет

Архитекторы часто пренебрегают использованием цветового оформления, поскольку в течение многих лет книги печатались в черно-белом формате, поэтому архитекторы и разработчики привыкли к монохромным рисункам. Хотя монохромное представление по-прежнему предпочтительнее, цвет все же применяется, когда он помогает отличить один артефакт от другого. Например, рассматривая стратегии обмена данными в микросервисах в разделе «Обмен данными» в главе 17 на с. 303, можно воспользоваться цветом, чтобы указать, что в координации участвуют два разных микросервиса, а не два экземпляра одного и того же сервиса (рис. 21.2).

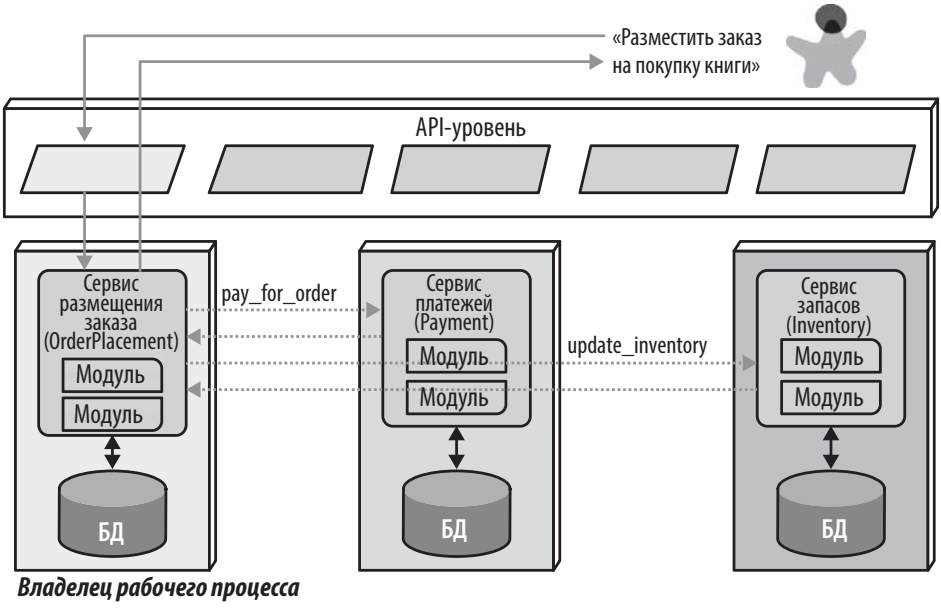


Рис. 21.2. Воспроизведение примера обмена данными между микросервисами, где разные сервисы показаны разными цветами[1]

Разъяснения

Если фигуры по каким-то причинам непонятны зрителям, добавляйте в схему разъяснения, четко указывающие на то, что собой представляет каждая фигура. Нет ничего ужаснее диаграммы с неверным толкованием — это еще хуже, чем отсутствие диаграммы.

Проведение презентаций

Еще одним гибким навыком, который необходим архитекторам, является умение проводить эффективные презентации с использованием таких инструментов, как PowerPoint или Keynote. Эти инструментальные средства являются языком общения современных организаций, и ожидается, что в любой компании люди умеют грамотно ими пользоваться. К сожалению, в отличие от текстовых процессоров и электронных таблиц, похоже, никому не хочется тратить время на изучение средств презентации.

[1] В черно-белом издании книги, к сожалению, не видно цветов. — *Примеч. ред.*

Нил, один из соавторов этой книги, несколько лет назад написал книгу о паттернах презентации под названием «Presentation Patterns»[1] (Addison-Wesley Professional), где рассказал, как применить подход с использованием паттернов/антипаттернов (распространенный в мире разработки ПО) к подготовке технических презентаций.

В книге «Presentation Patterns» есть важное замечание о фундаментальном различии между созданием документа и презентацией — фактор *времени*. В презентации докладчик управляет скоростью доведения замысла до аудитории, а читатель документа сам определяет свой темп чтения. Поэтому один из наиболее важных навыков, которым обязан овладеть архитектор, выбрав инструмент презентации, — это умение управлять временем.

Управление временем

Инструменты презентации предлагают два способа управления временем при демонстрации слайдов: с использованием переходов и с применением анимации. Переходы — это перемещения от слайда к слайду, а анимация позволяет разработчику презентации применять движение внутри слайда. Как правило, инструменты презентации позволяют задавать только один переход для каждого слайда, но привязывать к каждому элементу множество анимационных эффектов: входа (появления), выхода (исчезновения) и различных действий (например, перемещения, увеличения масштаба или другого динамического поведения).

Хотя инструментарий предлагает множество различных броских эффектов, архитекторы чаще используют переходы и анимацию для скрытия границ между слайдами. Один из широко известных антипаттернов, упомянутых в книге «Presentation Patterns» под названием Cookie-Cutter[2] (Резак для печенья), построен на утверждении, что замыслы не имеют предопределенного количества слов и, соответственно, дизайнеры презентации не должны искусственно дополнять контент, чтобы заполнить весь слайд. Аналогичным образом, демонстрация многих замыслов не умещается на один слайд. Сочетание переходов и анимации, например плавного исчезновения, позволяет докладчику скрывать границы отдельных слайдов, сшивая слайды вместе, чтобы довести до зрителей все повествование единым представлением. Чтобы обозначить завершение замысла и для предоставления зрителям визуальной подсказки о том, что они переключаются на рассмотрение другой темы, докладчик должен воспользоваться другим видом перехода (например, в виде двери или куба).

---

[1] https://presentationpatterns.com/

[2] https://presentationpatterns.com/glossary/#cookiecutter

Постепенное выстраивание

В книге «Presentation Patterns» есть также описание широко распространенного антипаттерна корпоративной презентации под названием Bullet-Riddled Corpse[1] (Тело, изрешеченное пулями), когда каждый слайд — это, по сути, заметки докладчика, проецируемые для всеобщего обозрения. У большинства читателей есть печальный опыт просмотра слайдов, заполненных текстом, который тут же прочитывается целиком (поскольку никто не в силах устоять от соблазна прочитать весь текст при его появлении на экране), и получается, что зрителю приходится бесцельно высиживать следующие десять минут, пока докладчик медленно произносит те же слова. Неудивительно, что многие корпоративные презентации навевают сплошную скуку!

Проводя презентацию, докладчик располагает двумя информационными каналами: вербальным и визуальным. Размещая на слайдах слишком много текста, а затем заставляя аудиторию выслушивать те же самые слова, докладчик перегружает один информационный канал и заставляет «голодать» другой. Лучшим решением этой проблемы будет использование в слайдах постепенного выстраивания с наращиванием (желательно в графическом представлении) объема информации по мере надобности.

Предположим, к примеру, что архитектор создает презентацию, объясняющую проблемы разветвления функциональности, и хочет поговорить о негативных последствиях слишком долгого хранения ветвей в активном состоянии. Рассмотрим графический слайд, показанный на рис. 21.3.

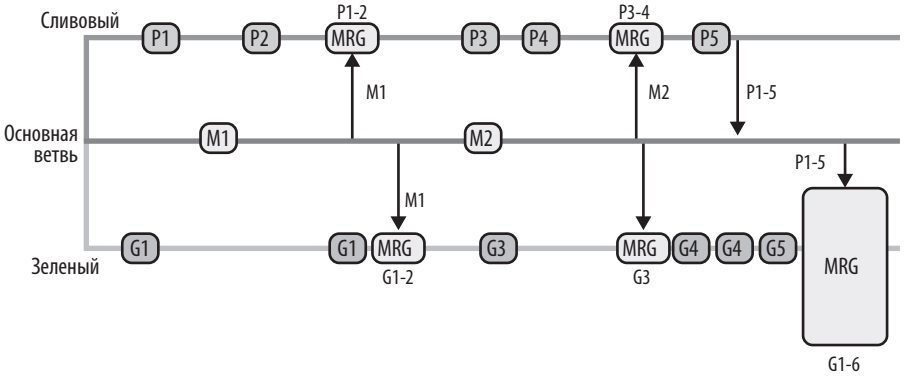


Рис. 21.3. Неудачная версия слайда, демонстрирующая негативный антипаттерн

[1] https://presentationpatterns.com/glossary/#bullet-riddledcorpse

Если докладчик сразу же покажет весь слайд (рис. 21.3), то аудитория увидит, что в конце происходит что-то плохое, но при этом ей придется ждать, пока объяснение не дойдет до этого места.

Вместо этого лучше воспользоваться тем же самым изображением, но скрывать его части при показе слайда (с помощью белого прямоугольника без границ) и показывать изображение порциями (рис. 21.4).

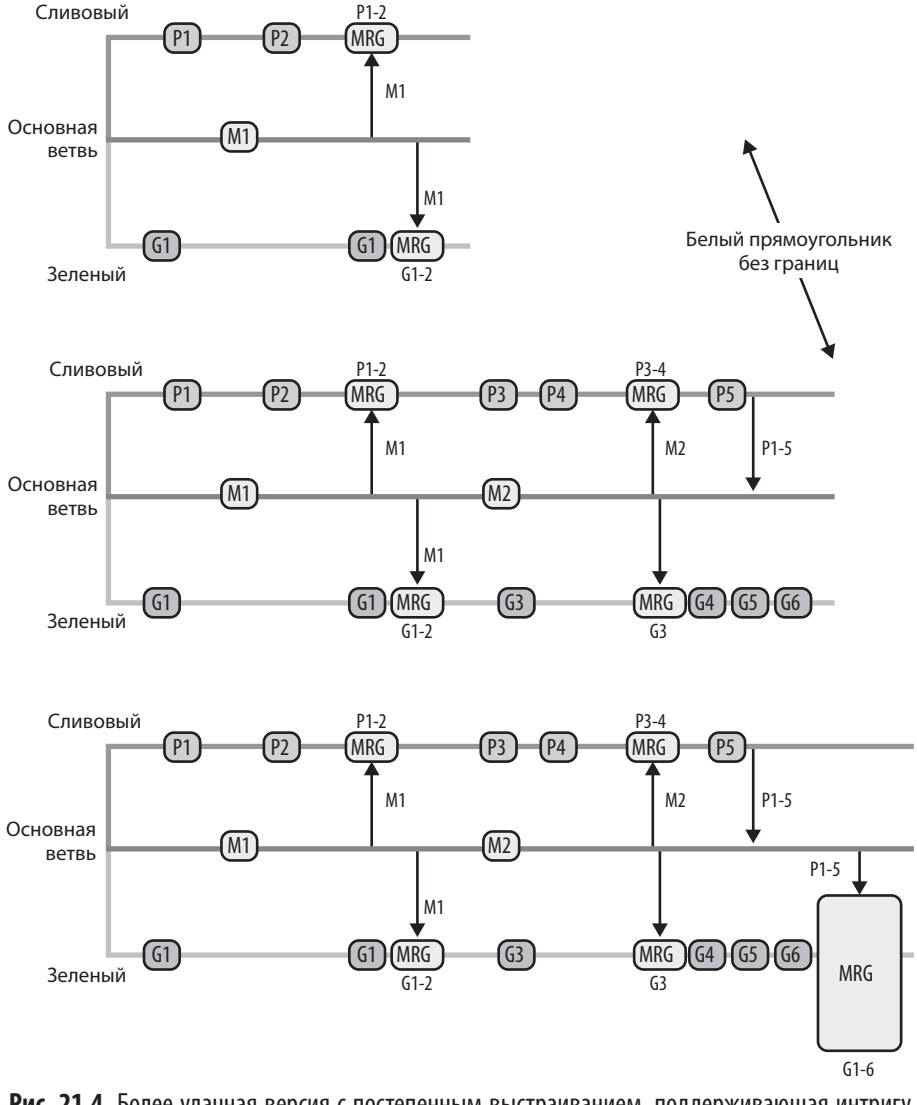


Рис. 21.4. Более удачная версия с постепенным выстраиванием, поддерживающая интригу

Еще более убедительные и увлекательные презентации докладчик сможет создавать при использовании анимации и переходов в сочетании с постепенным выстраиванием слайда.

Инфо-деки и презентации

Некоторые архитекторы создают набор слайдов в PowerPoint или Keynote, но никогда не устраивают их презентацию. Вместо этого они рассылают свою работу по электронной почте как журнальную статью, и каждый получатель просматривает ее в своем собственном темпе. *Инфо-деки* (*infodecks*) — это наборы слайдов, предназначенные не для проецирования на экран, а для краткого графического изложения информации. По сути, инструмент презентации в этом случае используется в качестве настольного издательского пакета.

Разница между этими двумя формами передачи информации заключается в полноте контента и в использовании переходов и анимации. Если кто-то будет листать набор слайдов как журнальную статью, автору слайдов не нужно добавлять никаких оживляющих элементов. Другим ключевым отличием инфодека от презентации является объем представляемого материала. Поскольку инфо-деки предназначены для автономного использования, они содержат всю информацию, которой захотели поделиться их создатели. В презентации слайды целенаправленно готовятся таким образом, чтобы стать всего лишь половиной презентации, а другой половиной будет человек, выступающий перед аудиторией.

Слайды — лишь половина всей истории

Распространенной ошибкой, совершаемой докладчиком, является встраивание в слайды всего содержимого презентации. Но если информация в слайдах исчерпывающая, докладчику стоило бы сэкономить время слушателей (которое они потратят, сидя на презентации) и просто отправить материалы по электронной почте в виде дека. Докладчики совершают ошибку, добавляя к слайдам слишком много материала, в то время как они могли бы придать больше убедительности самым важным моментам презентации. Следует помнить, что у докладчиков есть два информационных канала, поэтому, используя их стратегически, можно сделать свое выступление более сильным. Отличным примером этого является стратегическое использование затемнения.

Затемнение экрана

Затемнение представляет собой простой прием, при котором докладчик вставляет в презентацию пустой черный слайд, чтобы перевести все внимание на себя. Если имеются два информационных канала (слайды и докладчик) и один из них выключается (слайды), то это автоматически добавляет внимания докладчику. То есть если докладчик хочет высказаться, нужно вставить пустой слайд, и вся аудитория сконцентрируется на выступающем, поскольку он останется единственным объектом, представляющим интерес для аудитории.

Умение работать с инструментами презентации и знание приемов повышения качества презентаций — существенное дополнение к набору полезных навыков архитектора. Если у архитектора созрел великолепный замысел, но он не может найти способ его эффективно представить, у него никогда не появится шанс реализовать свою концепцию. Архитектура подразумевает сотрудничество, а чтобы привлечь единомышленников, архитекторы должны убедить других в состоятельности своей концепции. Средства проведения презентаций играют роль трибуны в современных корпорациях. Поэтому стоит как следует освоить их применение.

ГЛАВА 22

Эффективная команда

Кроме выстраивания технической архитектуры и принятия архитектурных решений, архитектор ПО также отвечает за то, чтобы направлять команду разработчиков в процессе внедрения задуманной архитектуры. Умелые архитекторы формируют эффективные команды, которые совместно решают проблемы и приходят к выигрышным решениям. Все это, конечно, может звучать как прописные истины, но уж слишком часто нам попадались архитекторы, которые игнорировали общение с командами разработчиков и уходили в глухую изоляцию при разработке архитектуры. Затем эта архитектура передавалась команде, которой с большим трудом удавалось завершить ее корректное внедрение. Способность добиться от команд высокой эффективности — одно из отличий результативного и успешного архитектора ПО. В этой главе будут представлены основные приемы, которые архитектор может взять на вооружение для достижения этой цели.

Рамки, устанавливаемые для команд

Наш опыт свидетельствует, что действия архитектора ПО могут предопределить как успех, так и провал команды разработчиков. Команды, не участвовавшие в цикле выработки замысла и изолированные от архитекторов ПО (а также от архитектуры), зачастую не получают ни должного уровня руководства, ни нужного объема сведений о различных ограничениях в системе, из-за чего они не в состоянии правильно реализовать архитектурный замысел.

Одна из ролей архитектора ПО заключается в выстраивании и доведении до команды ограничений, или же тех рамок, в которых разработчики могут заниматься реализацией архитектуры. Архитекторы могут выстроить слишком жесткие, слишком свободные или же четко выверенные рамки (рис. 22.1). Слишком жесткие или слишком свободные рамки напрямую влияют на способность команд успешно внедрить заданную архитектуру.

Рис. 22.1. Типы рамок, задаваемых архитектором ПО

Архитекторы, задающие слишком много ограничений, формируют жесткие рамки для команд разработчиков, перекрывая доступ ко многим инструментальным средствам, библиотекам и приемам программирования, необходимым для эффективной реализации системы. Внутри команды зреет недовольство, и разработчики обычно покидают проект в поиске более благоприятной и здоровой рабочей обстановки.

Может случиться и обратное. Архитектор ПО устанавливает слишком слабые ограничения (или вообще обходится без них), отдав на откуп команде разработчиков все важные архитектурные решения. В этом случае, который ничуть не лучше предыдущего, команда, по сути, возьмет на себя роль архитектора ПО, доказывая те или иные концепции и сражаясь за проектные решения без должного руководства, что приведет к непродуктивности, неразберихе и недовольству.

Грамотный архитектор стремится осуществлять правильный уровень руководства и установить разумные рамки, чтобы в распоряжении команды были необходимые инструменты и соответствующие библиотеки для продуктивной реализации архитектуры. Все остальное в этой главе посвящено способам формирования именно таких рамок.

Личные качества архитекторов

По личным качествам различаются следующие три типа архитекторов: *диктатор* (рис. 22.2), *кабинетный* (рис. 22.3) и *эффективный* (рис. 22.5). Каждый тип соответствует уровню рассмотренных в предыдущем разделе рамок: диктаторы устанавливают жесткие, кабинетные — свободные, а эффективные архитектуры — четко выверенные, разумные рамки.

Диктатор

Рис. 22.2. Диктатор (iStockPhoto)

Диктатор старается в процессе разработки программного продукта проконтролировать буквально каждую мелочь. Каждое принимаемое им решение слишком конкретизировано и доходит до самых низких уровней, в результате чего для команды разработчиков устанавливается слишком много ограничений.

Диктаторы устанавливают жесткие рамки. Такой архитектор может запретить загрузку библиотек, имеющих открытый код или принадлежащих стороннему разработчику, а вместо этого заставить команды разрабатывать все с нуля, используя API-интерфейс языка. Диктатор может также установить жесткие ограничения на соглашения об именах, конструкцию классов, длину методов и т. д. Он может пойти еще дальше и создать для команд разработчиков псевдокод. По сути, архитекторы-диктаторы крадут искусство программирования у разработчиков, вызывая их недовольство и неуважение к своей персоне.

Превратиться в диктатора очень просто, особенно если архитектором стал бывший разработчик. Роль архитектора — выстроить блоки приложения (компоненты) и определить, в чем будет заключаться суть взаимодействия между ними. А роль разработчика — получить эти компоненты и определить, как они будут реализованы с использованием диаграмм классов и паттернов проектирования.

Но при превращении из разработчика в архитектора у новоиспеченного архитектора возникает большой соблазн самому создать диаграммы классов и выбрать паттерны проектирования, поскольку он раньше именно этим и занимался.

Предположим, к примеру, что архитектор создает компонент (выстраивает блок архитектуры) для управления в системе справочными данными (reference data).

Эти справочные данные состоят из статических пар имя — значение, используемых на веб-сайте, а также из всевозможных кодов товаров и кодов складов (статических данных, используемых по всей системе). Роль архитектора — выявить компонент (в данном случае `Reference Manager`), определить основной набор операций для этого компонента (например, `GetData`, `SetData`, `ReloadCache`, `NotifyOnUpdate` и т. д.) и определить, какие компоненты нуждаются во взаимодействии с `Reference Manager`. Архитектор-диктатор может подумать, что наилучшим способом *реализации* этого компонента будет применение паттерна параллельного загрузчика, использующего внутренний кэш с конкретной структурой данных. Возможно, это удачный, но далеко не единственный дизайн. Важно, что придумывать внутренний дизайн `Reference Manager` должен уже не архитектор, а разработчик.

В разделе «Насколько жестким должен быть контроль?» на с. 386 будет сказано, что иногда архитектору полезно сыграть роль диктатора, все зависит от сложности проекта и профессионального уровня команды. Но в большинстве случаев архитектор-диктатор дезорганизует работу команды разработчиков, не обеспечивает должный уровень руководства, мешает и неэффективен в процессе реализации архитектуры.

Кабинетный архитектор

Рис. 22.3. Кабинетный архитектор (iStockPhoto)

Кабинетный архитектор — это специалист, давно отошедший от программирования (если вообще когда-либо им занимавшийся) и не учитывающий никакие

особенности будущей реализации архитектуры. Обычно он изолирован от команд разработчиков, никогда не взаимодействует с ними или же просто переключается на другой проект, стоит ему только выстроить исходные диаграммы архитектуры.

Порой кабинетный архитектор просто не мыслит в терминах технологической или бизнес-области, поэтому не может руководить или направлять команды при решении соответствующих проблем. Чем занимаются разработчики? Кодированием, очевидно. Сымитировать создание программного кода довольно трудно; разработчик либо выдает его, либо нет. А чем занимается архитектор? Никто не знает! Большинство архитекторов рисуют кучу линий и прямоугольников, но до какого уровня детализации они должны прорабатывать свои схемы? У архитектуры есть свой маленький неприличный секрет: пребывая в роли архитектора, ее довольно легко сымитировать!

Предположим, что кабинетный архитектор не разбирается в сути предмета или же не располагает временем для создания приемлемого архитектурного решения для системы биржевой торговли. В таком случае диаграмма архитектуры может быть похожа на то, что показано на рис. 22.4. С архитектурой здесь все в порядке, просто она слишком высокоуровневая, чтобы кому-то вообще могла бы пригодиться.

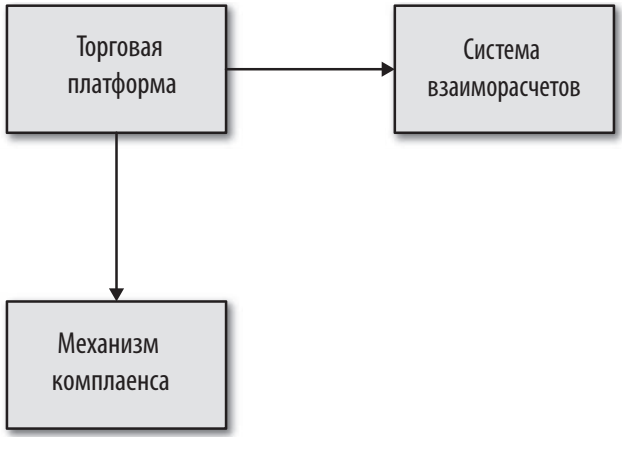


Рис. 22.4. Архитектура системы биржевой торговли, созданная кабинетным архитектором

Кабинетный архитектор устанавливает для команд разработчиков слишком свободные рамки. В таком случае командам в конце концов приходится брать на себя роль архитектора, по сути, делая за него всю его работу. В результате страдают темп и продуктивность, а команды не понимают, как должна работать система.

Стать кабинетным архитектором ничуть не сложнее, чем превратиться в архитектора-диктатора. Самый явный признак того, что архитектор начинает

превращаться в кабинетного, — нехватка времени для общения с командами разработчиков, реализующими архитектуру (или решение вообще не тратить на это время). Команды нуждаются в поддержке и руководстве со стороны архитектора, и им нужен специалист, способный ответить на возникающие вопросы из сферы технологий или бизнеса. Другими показателями, позволяющими отнести архитектора к разряду кабинетных, являются:

- недопонимание предметной области бизнеса, проблем бизнеса или используемой технологии;
- недостаточный практический опыт разработки программного обеспечения;
- игнорирование последствий, связанных с реализацией архитектурного решения.

Иногда архитектор не собирается становиться кабинетным, это происходит само собой из-за того, что он «распыляется» между большим количеством проектов или команд разработчиков и теряет связь с технологиями или бизнес-сферой. Архитектор может избежать этой участи, глубже вникая в технологию, используемую в проекте, и в особенности задач, стоящих перед бизнесом, а также в саму суть предметной области бизнеса.

Эффективный архитектор

Рис. 22.5. Эффективный архитектор ПО (iStockPhoto)

Благодаря грамотному руководству эффективного архитектора ПО обеспечивается слаженная работа всех команд. Он также должен быть уверен, что

команды располагают всеми необходимыми инструментами и технологиями. Кроме этого, такой архитектор устраняет любые препятствия, возникающие на пути разработчиков к достижению поставленных целей.

Какими бы простыми и очевидными ни казались эти показатели, на деле все далеко не так просто. Стать эффективным лидером команды разработчиков — целое искусство. Необходимо тесное сотрудничество и работа с командой, а также высокий авторитет. Все способы стать таким архитектором будут рассматриваться в следующих главах. А сейчас дадим несколько рекомендаций, чтобы сориентировать, насколько жестким должен быть контроль над командой разработчиков.

Насколько жестким должен быть контроль?

Эффективный архитектор знает, какой контроль следует устанавливать над той или иной командой разработчиков. Эта концепция известна как эластичное руководство (Elastic Leadership[1]) и широко проповедуется писателем и консультантом Роем Ошеровым (Roy Osherove). Мы немного отвлечемся от его наработок в данной области и сосредоточимся на конкретных факторах, важных для архитектуры ПО.

Степень авторитарности и кабинетности эффективного архитектора определяется пятью основными факторами. От них же зависит количество команд (или проектов), с одновременным управлением которых сможет справиться архитектор ПО:

Слаженность команды

Насколько хорошо специалисты команды знают друг друга? Есть ли у них опыт совместной работы над другим проектом? Как правило, чем лучше специалисты знают друг друга, тем меньше их нужно контролировать, поскольку из них начинает формироваться самоорганизующийся коллектив. И наоборот, чем больше в команде новичков, тем больше контроля требуется для того, чтобы облегчить сотрудничество среди членов команды и избавиться от обособленности отдельных групп.

Численность команды

Велика ли команда? (Мы считаем, что большой считается команда из более чем 12, а малой — из 4 и менее специалистов.) Чем больше команда, тем

[1] https://www.elasticleadership.com/

сильнее ее требуется контролировать. Чем она меньше, тем слабее нужен контроль. Более подробно этот вопрос рассматривается далее в разделе «Тревожные признаки в работе команд» на с. 391.

Совокупный опыт

Сколько в команде сениор-разработчиков? Сколько джуниоров? Насколько хорошо команда владеет технологией и знаниями бизнес-области? Команды с множеством джуниоров требуют более строгого контроля и наставничества, а контроль за командами с преобладанием сениоров может быть слабее. В последнем случае архитектор превращается из наставника в координатора.

Сложность проекта

Относится ли проект к особо сложным или же это просто разработка обычного веб-сайта? При разработке сложных проектов необходимо, чтобы архитектор был более доступен; он должен принимать непосредственное участие в разрешении возникающих проблем, то есть команда подвергается с его стороны более плотному контролю. Проблемы при разработке относительно несложных проектов возникают гораздо реже, поэтому потребности в повышенном контроле нет.

Продолжительность разработки проекта

Разработка проекта будет недолгой (два месяца), продолжительной (два года) или умеренно продолжительной (шесть месяцев)? Чем меньше сроки, тем слабее контроль, и наоборот, чем дольше разработка, тем больше требуется внимания.

Влияние большинства факторов на строгость контроля не вызывает сомнений, исключением здесь является фактор продолжительности. В предыдущем перечне отмечалось: чем короче проект, тем слабее контроль, и наоборот, чем он длиннее, тем жестче должен быть контроль. Интуиция подсказывает, что все должно быть наоборот, но это не так. Возьмем короткий двухмесячный проект. Времени для выработки требований, проведения экспериментов, разработки кода, тестирования каждого сценария и выпуска в продакшен мало. В этом случае архитектору больше подходит кабинетная роль, поскольку команда и так уже нацелена на срочность разработки. Архитектор-диктатор будет только путаться у нее под ногами, и скорее всего, сорвет сроки завершения проекта. А теперь возьмем проект продолжительностью в два года. Здесь разработчики расслаблены, не думают о сроках, обсуждают планы на отпуск и долго обедают. Чтобы не допустить отставания от графика разработки проекта и добиться приоритетности решения наиболее сложных задач, от архитектора требуется более строгий контроль.

Как правило, в большинстве проектов перечисленные факторы используются только для начального определения степени контроля, но по мере развития системы условия могут измениться. Поэтому мы рекомендуем постоянно анализировать эти факторы на протяжении жизненного цикла проекта, чтобы корректировать жесткость контроля над командой разработчиков.

Чтобы проиллюстрировать использование каждого из этих факторов для определения степени контроля над командой со стороны архитектора, предположим, что фиксированная шкала составляет 20 баллов для каждого фактора. Отрицательные значения соответствуют роли кабинетного архитектора (с меньшей степенью контроля и участия), а положительные — роли диктатора (с большей степенью контроля и участия). Эта шкала показана на рис. 22.6.

Рис. 22.6. Шкала степени контроля

Разумеется, применение такой шкалы не отличается высокой точностью, но она помогает определить относительную степень контроля над командой. Рассмотрим, к примеру, сценарий разработки проекта, показанный в табл. 22.1 и на рис. 22.7. В таблице показаны факторы, склоняющие к роли либо диктатора (+20), либо кабинетного архитектора (–20). Эти факторы складываются в общий балл 60, показывающий, что архитектор в основном может играть кабинетную роль и не вмешиваться в работу команды.

В первом сценарии эти факторы берутся в расчет, чтобы продемонстрировать, что эффективный архитектор ПО должен изначально играть роль координатора и не принимать слишком большого участия в повседневной работе команды. Он нужен для ответов на различные вопросы и для контроля правильного курса команды, но по большому счету, он не должен ни во что вмешиваться, позволяя

опытным специалистам заниматься тем, что они знают лучше него, — быстрой разработкой программных средств.

Таблица 22.1. Пример первого сценария определения степени контроля

Фактор	Значение	Оценка	Роль, исполняемая архитектором
Слаженность команды	Новички в команде	+20	Диктатор
Численность команды	Небольшая (4 человека)	–20	Кабинетный
Совокупный опыт	Все опытные	–20	Кабинетный
Сложность проекта	Относительно простой	–20	Кабинетный
Продолжительность разработки проекта	2 месяца	–20	Кабинетный
Общий балл		–60	Кабинетный

Рис. 22.7. Степень контроля для первого сценария

Рассмотрим сценарий другого рода, показанный в табл. 22.2 и на рис. 22.8, где все специалисты команды хорошо знают друг друга, но команда большая (12 человек) и состоит в основном из джуниоров (неопытных разработчиков). Проект относительно сложный с продолжительностью разработки 6 месяцев. В этом случае общий балл составляет –20, показывая, что эффективный архитектор должен ежедневно контролировать разработчиков и взять на себя роль наставника и коуча, но не настолько, чтобы нарушать работу команды.

Таблица 22.2. Пример второго сценария определения степени контроля

Фактор	Значение	Оценка	Роль, исполняемая архитектором
Слаженность команды	Хорошо знают друг друга	–20	Кабинетный
Численность команды	Большая (12 человек)	+20	Диктатор
Совокупный опыт	В основном джуниоры	+20	Диктатор
Сложность проекта	Высокая	+20	Диктатор
Продолжительность разработки проекта	6 месяцев	–20	Кабинетный
Общий балл		–20	Диктатор

Рис. 22.8. Степень контроля для второго сценария

Количественно оценить данные факторы весьма непросто, поскольку некоторые из них (например, совокупный опыт команды) могут быть весомее других. В таких случаях показатели легко изменить или добавить им вес с учетом любого конкретного сценария или складывающихся условий. Как бы то ни было, основная идея здесь заключается в том, что с помощью этих пяти факторов архитектор может определить жесткость контроля над командой и рамки, в пределах которых разработчики будут выполнять свою задачу (жесткие или же свободные).

Тревожные признаки в работе команд

Как показано в предыдущем разделе, численность команды является одним из факторов, влияющих на степень контроля со стороны архитектора. Чем больше команда, тем в большем контроле она нуждается, а чем она меньше, тем слабее потребность в контроле. При обсуждении численности наиболее эффективной команды разработчиков на первый план выходят три фактора:

- замедление процесса;
- плюралистическое невежество;
- размытость ответственности.

Фактор *замедления процесса*, известный также как закон Брукса, был придуман Фредом Бруксом (Fred Brooks) и изложен в его книге «The Mythical Man Month»[1] (издательство Addison-Wesley). Этот фактор основан на следующей идее: чем больше людей привлечено к проекту, тем дольше он будет разрабатываться. Как показано на рис. 22.9, *групповой потенциал* определяется суммой усилий, прилагаемых каждым специалистом команды. Но *фактическая производительность* любой команды будет ниже группового потенциала, и разница выразится в *замедлении процесса*.

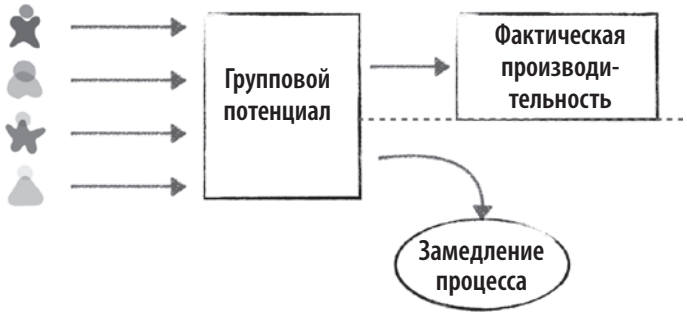


Рис. 22.9. Численность команды влияет на фактическую производительность (закон Брукса)[2]

[1] Брукс Ф. «Мифический человеко-месяц, или Как создаются программные системы». СПб., издательство «Питер».

[2] Брукс сформулировал это так: «Если проект не укладывается в сроки, то добавление рабочей силы задержит его еще больше». — *Примеч. ред.*

Эффективный архитектор будет наблюдать за командой разработчиков и отслеживать замедление процесса. С помощью этого фактора можно определить оптимальную численность команды для разработки конкретного проекта или для участия в его разработке. Одним из признаков замедления процесса являются частые конфликты при объединении кода в процессе его сохранения в репозиторий. Это показывает, что специалисты команды, возможно, наступают друг другу на пятки и работают над одним и тем же кодом. Одним из способов, позволяющих избежать замедления процесса, может стать поиск областей параллельного ведения разработок внутри команды, а также организация работы программистов над разными сервисами или областями приложения. Как только к проекту присоединяется новый разработчик и для него нет возможности создать параллельный рабочий поток, эффективный архитектор выясняет причину расширения состава команды и доводит до руководителя проекта суть негативного влияния, которое этот дополнительный специалистом окажет на команду.

Когда численность команды становится слишком большой, возникает *плюралистическое невежество*[1]. Суть его в том, что все вместе соглашаются с нормой (но втайне ее отвергают), поскольку полагают, что не замечают чего-то очевидного. Предположим, к примеру, что в большой команде практически единогласно принято использование обмена сообщениями между двумя удаленными сервисами. Но один из специалистов считает это глупой затеей, поскольку между сервисами стоит защитный брандмауэр. И все же, вместо того чтобы высказать свое мнение, он соглашается на применение обмена сообщениями (втайне отвергая эту идею), поскольку опасается, что мог просмотреть что-то очевидное либо будет выглядеть идиотом, если выскажет свои возражения. В данном случае человек, отвергающий норму, окажется прав: обмен сообщениями работать не будет, поскольку между двумя удаленными сервисами установлен защитный брандмауэр. Выскажи он свое мнение (при меньшей численности команды), исходное предложение было бы оспорено и вместо него был бы использован другой протокол (например, REST), что стало бы в данном случае более удачным решением.

Понятие плюралистического невежества приобрело известность благодаря сказке Ганса Христиана Андерсена «Голый король». Короля убедили, что его новые одежды не видят те, кто недостоин их видеть. И он прохаживался в абсолютно голом виде, спрашивая своих подданных, нравится ли им его новый наряд. Все подданные боялись прослыть глупыми или недостойными и отвечали королю, что новый наряд — лучший из всех, что им приходилось когда-либо видеть. Вся эта глупость продолжалась до тех пор, пока ребенок не крикнул, что король-то голый!

[1] https://ru.wikibrief.org/wiki/Pluralistic_ignorance

Эффективный архитектор ПО должен постоянно следить за выражением лиц и языком тела участников разных совещаний или дискуссий и выступать в роли посредника в случае проявления признаков плюралистического невежества. Он может прервать обсуждение и узнать мнение определенного специалиста о предлагаемом решении, занять его сторону и поддерживать, когда тот высказывает свое мнение.

Третий фактор, позволяющий определить оптимальную численность команды, называется *размытостью ответственности*. Он основывается на том факте, что рост численности команды негативно сказывается на общении. Путаница в обязанностях и невыполненные задачи — явные признаки того, что численность команды слишком велика.

Посмотрите на рис. 22.10. Что вы видите?

Рис. 22.10. Размытость ответственности

На рисунке человек стоит у сломавшейся машины на обочине сельской дороги. Сколько людей в этой ситуации может остановиться и спросить водителя, все ли в порядке? Дорога узкая, в малонаселенном месте; наверное, это сделает каждый, кто будет проезжать мимо. А сколько раз водители останавливались из-за неисправности автомобиля на обочине оживленных магистралей в центре больших городов, и тысячи машин просто проезжали мимо, никто не притормаживал и не интересовался, все ли в порядке. Всем это знакомо. Это хороший пример размытости ответственности. Чем оживленнее и густонаселеннее город,

тем чаще люди думают, что водитель уже вызвал аварийку или она уже в пути, поскольку случившаяся с ним неприятность была у всех на виду. Но в большинстве подобных случаев помощь еще даже не выезжала, а водитель оказывался брошенным на произвол судьбы, проклиная себя за севший или забытый дома мобильник и не имея возможности вызвать техпомощь.

Эффективный архитектор не только помогает команде разработчиков пройти через весь процесс реализации архитектуры, но также обеспечивает здоровую атмосферу, хорошее настроение и слаженность в работе на пути к достижению общей цели. Отслеживание появления всех трех рассматриваемых здесь тревожных признаков и последовательное оказание помощи по их устранению обеспечивает высокую эффективность команды.

Чек-листы

Пилоты самолетов, даже самые опытные, используют чек-листы в каждом полете. У пилотов имеются чек-листы для взлета, посадки и тысячи других как обычных, так и редко случающихся критических ситуаций. Их польза очевидна, потому что всего одна пропущенная настройка (например, установка закрылков на угол 10 градусов) или процедура (например, получение разрешения на вход в диспетчерскую зону управления аэропорта) может быть той самой чертой, за которой безопасный полет превращается в катастрофу.

Доктор Атул Гаванде (Atul Gawande) написал замечательную книгу под названием «The Checklist Manifesto»[1] (издательство Picador), в которой дал описание важности чек-листов для выполнения хирургических процедур. Встревоженный высоким уровнем случаев появления стафилококковой инфекции в больницах, доктор Гаванде создал хирургические чек-листы, пытаясь снизить этот уровень. В своей книге он показывает, что уровень стафилококковой инфекции в больницах, использующих такие инструкции, снизился почти до нуля, в то время как ее уровень в наблюдаемых больницах, не использующих чек-листы, продолжал расти.

Чек-листы доказали свою эффективность. Они являются прекрасным средством, позволяющим все охватить и ничего не упустить. А если они хорошо работают, то почему бы их не применить в индустрии создания программного обеспечения? Исходя из нашего личного опыта, мы твердо убеждены, что чек-листы в немалой степени способствуют повышению эффективности всей деятельности команд разработчиков. Но это утверждение не обходится без уточнений. Практически никто

[1] Гаванде А. «Чек-лист. Как избежать глупых ошибок, ведущих к фатальным последствиям».

из разработчиков программных средств не управляет самолетами и не проводит хирургические операции. Иными словами, разработчикам программных средств не требуются чек-листы абсолютно для всех сфер их деятельности. Ключевым фактором превращения команд в эффективно работающий механизм является четкое понимание, когда следует, а когда не следует применять чек-листы.

Рассмотрим чек-лист для создания новой таблицы базы данных, показанный на рис. 22.11.

Выполнено	Описание задачи
☐	Определить имена и типы полей в столбцах
☐	Заполнить форму запроса на создание таблицы базы данных
☐	Получить разрешение на новую таблицу базы данных
☐	Отправить форму запроса группе разработки базы данных
☐	Проверить таблицу после ее создания

Рис. 22.11. Пример неудачного чек-листа

Это не чек-лист, а набор процедурных действий, который в таком виде не может фигурировать в данном документе. Например, таблица базы данных не может быть проверена, если форма еще не была отправлена! В чек-листе не должно быть никаких процессов, имеющих процедурный поток зависимых задач. Также не следует в него включать простые, хорошо известные процессы, выполняемые часто и без ошибок.

Подходящими кандидатами для включения в чек-лист являются процессы, не имеющие какого-либо процедурного порядка или зависимых задач, а также процессы, в которых могут быть допущены ошибки или пропущены какие-то действия. Ключ к созданию эффективного чек-листа — не переусердствовать, включая в него все подряд. Архитекторы поняли, что чек-листы повышают эффективность работы команд разработчиков, и стали создавать их абсолютно для всего, попадая под действие *закона убывающей полезности* (law of diminishing returns). Чем больше чек-листов создает архитектор, тем меньше шансов, что разработчики ими воспользуются. Другим ключевым фактором успеха является предельный минимализм в сочетании с охватом всех необходимых этапов процесса. Разработчики не будут работать со слишком длинными чек-листами. Нужно исключать те пункты, которые могут выполняться в автоматическом режиме.

К трем ключевым чек-листам, которые мы посчитали наиболее эффективными, относятся *чек-лист завершения разработки кода*, *чек-лист модульного и функцио-*

нального тестирования и *чек-лист релиза ПО*. Каждый из них рассматривается в следующих разделах.

Не бойтесь включать в чек-лист очевидное. Именно очевидные действия чаще всего пропускаются или забываются.

ЭФФЕКТ ХОТОРНА

Одна из проблем чек-листов заключается в том, чтобы заставить разработчиков пользоваться ими надлежащим образом. Слишком часто бывает, что некоторые разработчики не укладываются в график и просто помечают пункты чек-листа как завершенные без фактического выполнения задач.

Одним из способов решения этой проблемы является разговор с командой о важности использования чек-листов и о том, как они могут повлиять на работу команды. Пусть все в команде прочтут упомянутую выше книгу Атула Гаванде, чтобы в полной мере осознать пользу и целесообразность чек-листов. Помогает также совместное определение сотрудниками содержания чек-листа.

Когда ничто уже не срабатывает, архитекторы могут вызвать себе в помощь так называемый эффект Хоторна[1] (Hawthorne effect). Суть его в том, что если люди знают, что за ними наблюдают или следят, их поведение изменяется, и они, как правило, поступают правильно. В качестве примеров можно привести обзорные камеры на зданиях и вокруг них, которые на самом деле не работают или же фактически не ведут никаких записей (что случается довольно часто!), или же программное обеспечение для мониторинга веб-сайтов (сколько из составленных этими программами отчетов реально просматривается?).

Эффект Хоторна можно использовать и для того, чтобы повлиять на порядок применения чек-листов. Архитектор может рассказать команде, что применение чек-листов имеет решающее значение для эффективной работы, поэтому будут регулярные проверки их выполнения (хотя на самом деле такой контроль будут осуществляться лишь изредка). Под воздействием эффекта Хоторна разработчики станут гораздо реже пропускать пункты или помечать их выполненными без фактического завершения.

[1] https://ru.wikibrief.org/wiki/Hawthorne_effect

Чек-лист завершения разработки кода

Чек-лист завершения разработки кода является весьма действенным инструментом, особенно когда разработчик программных средств утверждает, что закончил свою работу. Этот список помогает также определить критерии готовности. Если все пункты чек-листа выполнены, разработчик может заявить, что он фактически завершил работу над кодом.

Вот некоторые пункты чек-листа завершения разработки кода:

- Стандарты кодирования и форматирования, не включенные в автоматизированные средства.
- Часто упускаемые из виду пункты (например, поглощенные исключения).
- Специфические стандарты проекта.
- Специальные инструкции или процедуры команды.

Пример чек-листа завершения разработки кода показан на рис. 22.12.

Выполнено	Описание задачи
☐	Запустить очистку и форматирование кода
☐	Запустить настраиваемый инструмент проверки исходного кода
☐	Убедиться, что журнал аудита ведется для всех обновлений
☐	Убедиться в отсутствии поглощенных исключений
☐	Проверить наличие жестко заданных значений и преобразовать их в константы
☐	Проверить, что вызов setFailure() осуществляется только из публичных методов
☐	Включить @ServiceEntrypoint в API-класс сервиса

Рис. 22.12. Пример чек-листа завершения разработки кода

Обратите внимание на очевидные задачи, такие как «Запустить очистку и форматирование кода» и «Убедиться в отсутствии поглощенных исключений». Сколько раз разработчик спешил или тянул с выполнением задачи до последнего дня итерации и забывал запускать очистку кода и форматирование из IDE-среды? Да тысячу раз. В книге «The Checklist Manifesto» Гаванде отмечал то же самое явление в отношении хирургических процедур: очевидное зачастую просто упускалось из виду.

Также обратите внимание на пункты второй, третий, шестой и седьмой. Несмотря на то что они вполне подходят для включения в чек-лист, архитектор должен всегда пересматривать весь список задач, чтобы находить в нем те пункты, выполнение которых может быть автоматизировано или написано в виде плагина для проверки правильности кода. Например, если пункт «Включить @ServiceEntrypoint в API-класс сервиса», возможно, не поддается автоматической проверке, то пункт «Проверить, что вызов setFailure() осуществляется только из публичных методов» точно можно автоматизировать (это обычная автоматическая проверка в инструменте последовательного обхода кода любого типа). Проверка наличия областей автоматизации помогает сократить как объем, так и засорённость контрольного списка, повышая его эффективность.

Чек-лист модульного и функционального тестирования

Наверное, к числу наиболее эффективных следует отнести чек-лист модульного и функционального тестирования. В этом списке содержатся некоторые нестандартные тесты и тесты граничных случаев, о проведении которых разработчики ПО нередко забывают. Когда кто-нибудь из отдела QA обнаруживает проблему программного кода при прогоне конкретного тестового сценария, этот сценарий включается в соответствующий чек-лист.

Рассматриваемый чек-лист обычно один из самых длинных, поскольку код проверяется запуском множества разнообразных тестов. Его цель состоит в создании максимально полного программного кода, чтобы после выполнения всех пунктов из списка код был готов к продакшен.

Вот несколько пунктов, встречающихся в обычном чек-листе модульного и функционального тестирования:

- спецсимволы в текстовых и числовых полях;
- диапазоны минимальных и максимальных значений;
- необычные и экстремальные тестовые сценарии;
- отсутствующие поля.

Как и в чек-листе завершения разработки кода, любой пункт, который может быть реализован за счет автоматизированного теста, должен быть исключен. Предположим, к примеру, что в чек-листе есть пункт для приложения биржевой торговли по тестированию на отрицательные значения количества акций (например, на BUY для –1000 акций Apple [AAPL]). Если эта проверка автоматизирована и проводится модульным или функциональным тестом из тестового набора, такой пункт нужно исключить из списка.

Иногда разработчики при написании модульных тестов не знают, с чего начать или сколько таких тестов нужно создать. Чек-лист гарантирует включение в процесс разработки общих или специализированных тестовых сценариев. Он также эффективен для преодоления недопонимания между разработчиками и тестировщиками в ситуациях, когда эти две задачи выполняются разными командами. Чем более полное тестирование выполняет команда разработчиков, тем проще работа команды тестировщиков и больше возможностей сконцентрироваться на конкретных бизнес-сценариях, не предусмотренных в чек-листах.

Чек-лист релиза ПО

Выпуск программного продукта в продакшен является, наверное, одним из самых уязвимых для ошибок этапов в жизненном цикле разработки ПО, и поэтому для этого процесса составляется довольно длинный чек-лист. Этот список помогает избежать появления неудачных сборок и нестабильных развертываний и существенно снижает количество проблем, возникающих во время выпуска ПО.

Чек-лист выпуска программного продукта обычно относится к наиболее изменчивым из всех контрольных списков, поскольку для устранения вновь выявленных ошибок и исключения обстоятельств, сложившихся при очередном неудачном или проблемном развертывании, в нем постоянно что-то нужно менять.

Вот некоторые пункты, обычно включаемые в этот чек-лист:

- изменения конфигурации на серверах или на внешних конфигурационных серверах;
- библиотеки сторонних производителей, добавленные к проекту (JAR, DLL и т. д.);
- обновления базы данных и соответствующих сценариев миграции базы данных.

При каждом сбое в процессе сборки или развертывания архитектор должен проанализировать главную причину сбоя и добавить соответствующий пункт в чек-лист. Тогда этот пункт будет проверен при следующей сборке или развертывании, что предотвратит повторение случившегося сбоя.

Выдача рекомендаций

Архитектор ПО может повысить эффективность работы команд, выдавая рекомендации на основе принципов проектирования. Это также поможет сформировать

рамки (определить ограничения) в соответствии с описанием, предоставленным в первом разделе этой главы, в пределах которых разработчики смогут реализовывать архитектуру. Эффективное доведение до исполнителей этих принципов проектирования — один из ключей к формированию успешной команды.

ВЛИЯНИЕ БИЗНЕС-ОБОСНОВАНИЙ

Один из наших авторов (Марк) был ведущим архитектором в весьма сложном проекте на основе языка Java, к работе над которым была привлечена крупная команда разработчиков. Один из специалистов команды был одержим языком программирования Scala и очень хотел использовать его в проекте. Это острое желание использовать Scala сыграло настолько деструктивную роль, что несколько ключевых специалистов команды проинформировали Марка о своих планах покинуть проект и перейти в другую, «менее токсичную» рабочую среду. Марк убедил этих сотрудников ненадолго отложить решение о переходе и поговорил с поклонником языка Scala. Марк сказал ему, что поддержит использование Scala в проекте, но для этого ему требуется бизнес-обоснование, поскольку обучение персонала и переписывание кода не обойдутся без дополнительных затрат. Любитель Scala пришел в восторг, заявил, что без промедления займется этим, и, покидая совещание, крикнул: «Спасибо, ты лучший!».

На следующий день этот энтузиаст пришел в офис совершенно в другом настроении. Он тут же подошел к Марку и попросил разрешения поговорить с ним. Они удалились в конференц-зал, и любитель Scala смиренно произнес: «Спасибо». Он объяснил Марку, что мог бы привести все имеющиеся в мире технические обоснования по использованию Scala, но ни одно из предлагаемых при этом преимуществ не имело какой-либо бизнес-ценности с точки зрения необходимых архитектурных свойств: ни по затратам, ни по бюджету, ни по срокам выполнения проекта. Фактически энтузиаст Scala понял, что увеличение затрат, бюджета и сроков не будет скомпенсировано никакой пользой.

Осознав деструктивность своей позиции, он быстро превратился в одного из лучших и наиболее полезных специалистов команды, а все потому, что ему было предложено дать бизнес-обоснование его желания. Это возросшее понимание необходимости обоснований не только сделало его лучшим разработчиком ПО, но также укрепило и оздоровило команду. И кстати, те два ключевых разработчика так и остались в проекте до самого конца.

Чтобы убедиться в правильности этого утверждения, рассмотрим выдачу рекомендаций команде разработчиков по использованию так называемого *многоуровневого стека* — коллекции сторонних библиотек (например, JAR-файлов и DLL-библиотек), из которой составляется приложение. Команды разработчиков обычно задают множество вопросов, касающихся принятия самостоятельных решений по использованию библиотек и связанных с ними ограничений.

Используя этот пример, эффективный архитектор ПО может выдавать рекомендации команде разработчиков, предварительно попросив их ответить на следующие вопросы:

1. Имеются ли какие-нибудь совпадения между предлагаемыми библиотеками и существующей в системе функциональностью?
2. Чем обосновывается применение предлагаемой библиотеки?

Первый вопрос стимулирует разработчиков пристально изучить существующие библиотеки, чтобы понять, нужна ли еще одна. Разработчики часто игнорируют подобные действия, создавая массу дубликатов функциональности, особенно при разработке крупных проектов с большой командой.

Второй вопрос заставляет разработчика задуматься, для чего нужна новая библиотека или функциональная возможность. Эффективный архитектор ПО обязательно запросит техническое и бизнес-обоснование необходимости использования дополнительной библиотеки. Это может стать весьма действенным методом убеждения команды разработчиков внимательнее относиться к процессу принятия решений.

Возвращаясь к примеру управления многоуровневым стеком, приведем еще один эффективный прием передачи принципов проектирования: графическая демонстрация тех областей, где команда разработчиков может принимать решения, и тех, где она не может этого делать. На рис. 22.13 показано, как может выглядеть такое графическое представление (а также рекомендации) для управления многоуровневым стеком.

Используя графическое представление, показанное на рис. 22.13, архитектор приводит примеры того, что будет содержаться в каждой категории сторонней библиотеки, а затем выдает свои рекомендации (принципы проектирования), разъяснив, что разработчики могут, а чего не могут делать (то есть определяет те самые рамки, приводившиеся в первом разделе главы).

Рис. 22.13. Выдача рекомендаций для многоуровневого стека

Вот, например, три категории, определенные для любой сторонней библиотеки:

Специальное назначение

Это специализированные библиотеки, используемые, к примеру, для отображения информации в формате PDF, сканирования штрихкода и решения других задач, не требующих написания собственного кода.

Общее назначение

Это библиотеки, помещенные в надстройку API-интерфейса языка и включающие в себя такие функциональные средства, как Apache Commons и Guava для Java.

Среда разработки

Это библиотеки, используемые, к примеру, для сохранения данных (такие, как Hibernate) и инверсии управления (такие, как Spring). Иными словами, эти библиотеки составляют целый уровень или структуру приложения и являются глубоко встраиваемыми в исходный код.

После определения категорий (предыдущие категории приведены только в качестве примера, в действительности их может быть намного больше) архитектор выстраивает рамки вокруг этих принципов проектирования. Заметьте, что

в примере, показанном на рис. 22.13, для данного конкретного приложения или проекта архитектор указал, что в отношении библиотек специального назначения решение может принимать разработчик, не консультируясь с архитектором. В отношении библиотек общего назначения он указал, что разработчик может проанализировать дублирование и дать обоснованные рекомендации, но эта категория библиотек требует одобрения архитектора. И наконец, в отношении библиотек среды разработки отмечено, что все они выбираются по решению архитектора; иными словами, команды разработчиков не должны даже проводить анализ этих типов библиотек; архитектор берет на себя всю ответственность.

Итоги

Превращение команд в эффективно работающий механизм — великий труд. Для него требуется богатый опыт и наработанная практика, а также вполне определенные навыки работы с людьми (которые будут рассматриваться в следующих главах книги). И тем не менее рассмотренные в этой главе простые приемы эластичного руководства, использования чек-листов и выдачи рекомендаций с корректным информированием о принятых принципах проектирования реально работают и доказали свою эффективность.

Возможно, кто-то поставит под сомнение роль архитектора в такой деятельности, возложив все усилия по повышению эффективности работы команд на руководителя разработки или на руководителя проекта. Но с такой постановкой вопроса мы категорически не согласны. Архитектор ПО не только осуществляет техническое руководство командой, но и сопровождает ее в процессе реализации архитектуры. Тесное сотрудничество архитектора и команды разработчиков позволяет архитектору наблюдать за ходом работы команды и, следовательно, вносить изменения, позволяющие повысить эффективность ведения разработки. Именно это отличает эффективного архитектора ПО от технического архитектора.

ГЛАВА 23

Навыки лидерства и ведения переговоров

Навыки лидерства и ведения переговоров приобретаются нелегко. Чтобы стать эффективным архитектором ПО, нужно долго учиться, практиковаться и «набивать шишки». Конечно, книга не сделает в одночасье из архитектора искусного лидера и переговорщика, но приемы, представленные в этой главе, дадут хорошую отправную точку для получения этих ценных навыков.

Переговоры и фасилитация

В самом начале книги были перечислены основные требования к архитектору, одним из которых является необходимость понимать политическую атмосферу предприятия и ориентироваться в ней. Суть в том, что практически каждое принимаемое решение будет восприниматься критически — как со стороны разработчиков, полагающих, что они гораздо компетентнее архитектора и поэтому могут применить более удачный подход, так и со стороны других архитекторов организации, считающих, что у них есть более удачный замысел или подход к решению задачи. И наконец, критическую оценку решение может получить от стейкхолдеров, уверенных, что оно обойдется слишком дорого или его реализация займет слишком много времени.

К примеру, архитектор решил для снижения риска в отношении общей доступности системы воспользоваться кластеризацией и интеграцией баз данных (используя отдельные физические экземпляры баз данных для каждой предметной области). Для решения проблемы доступности базы данных это вполне разумное решение, но отнюдь не дешевое. В этом примере архитектор должен переговорить с ключевыми бизнес-стейкхолдерами (которые платят за создание системы) и прийти с ними к согласию насчет компромисса между доступностью и стоимостью.

Умение вести переговоры — один из наиболее важных навыков, которыми должен обладать архитектор ПО. Эффективные архитекторы разбираются в политике организации, обладают развитыми навыками ведения переговоров и фасилитации и способны преодолевать возникающие разногласия и вырабатывать решения, устраивающие всех заинтересованных лиц.

Ведение переговоров с бизнес-партнерами

Рассмотрим следующий сценарий из реальной жизни (сценарий 1) с участием ключевого бизнес-стейкхолдера и ведущего архитектора.

Сценарий 1

> Главный спонсор проекта настаивает, что новая торговая система должна поддерживать доступность в пять девяток (99.999%). Но ведущий архитектор, основываясь на исследованиях, подсчетах и знаниях бизнес-области, убежден, что будет достаточно доступности и в три девятки (99.9%). Проблема в том, что спонсор проекта не привык ошибаться и не терпит возражений и высокомерных оппонентов. Спонсор не силен в технике (но сам так не считает), в результате чего склонен вмешиваться в нефункциональные аспекты проекта. Архитектор путем переговоров должен убедить спонсора проекта, что трех девяток доступности (99.9%) будет вполне достаточно.

В ведении подобного рода переговоров архитектор должен вести себя сдержанно и доводить до оппонента результаты проведенной аналитической работы без излишнего напора и самомнения, но в то же время не пропустить ничего, что могло бы доказать его неправоту в ходе переговоров. Чтобы успешно вести подобные переговоры, архитектор может воспользоваться несколькими ключевыми приемами.

Чтобы лучше понять ситуацию, обращайте внимание на «громкие слова» и грамматические конструкции в речи оппонента.

Высказывания вроде «у нас должно быть нулевое время простоя» или «мне эти функции нужны были еще вчера» по своей сути бессмысленны и все же дают архитектору ценную информацию о том, с чего можно начать переговоры. Например, когда спонсора проекта спрашивают, когда понадобится конкретная функция, а он отвечает, что «она была нужна еще вчера», архитектор ПО понимает, что этому стейкхолдеру важно время выхода на рынок. А фраза типа «эта система должна просто летать» означает, что приоритет отдается произво-

дительности. Высказывание о «нулевом простое» означает, что для приложения главное — доступность. Эффективный архитектор ПО проанализирует эти слегка бессмысленные фразы, чтобы лучше разобраться в реальных проблемах, и использует это в ходе переговоров.

Вернемся к рассмотренному выше сценарию 1. Ключевой спонсор проекта захотел доступности в пять девяток. Для архитектора это будет означать, что приоритет отдан доступности. Тогда в ход идет второй прием ведения переговоров:

Прежде чем вступить в переговоры, нужно собрать как можно больше информации.

Фраза «пять девяток» — фигура речи, означающая высокую доступность. Но что конкретно означают пять девяток доступности? Проведенные до переговоров исследования и сбор информации позволили составить табл. 23.1.

Таблица 23.1. Девятки доступности

Время работы в процентах	Простои в год (в день)
90.0% (одна девятка)	36 дней 12 часов (2.4 ч)
99.0% (две девятки)	87 часов 46 минут (14 мин)
99.9% (три девятки)	8 часов 46 минут (86 с)
99.99% (четыре девятки)	52 минут 33 секунды (7 с)
99.999% (пять девяток)	5 минут 35 секунд (1 с)
99.9999% (шесть девяток)	31.5 секунды (86 мс)

«Пять девяток» доступности — это 5 минут и 35 секунд простоя в год, или одна секунда в день незапланированного простоя. Весьма амбициозный и в то же время крайне дорогостоящий и абсолютно ненужный уровень доступности. В разговоре лучше придерживаться не пресловутых девяток, а их перевода в часы и минуты (а в данном случае в секунды).

При ведении переговоров по сценарию 1 нужно убедиться в правильном понимании интереса оппонента («Насколько я понял, для системы очень важна высокая доступность»), а затем перевести переговоры с девяток на вполне понятные часы и минуты незапланированных простоев. Три девятки (которые архитектор считает вполне приемлемыми) — это примерно 86 секунд незапланированных

простоев в день, — конечно же, вполне разумный показатель, учитывая контекст глобальной торговой системы, описанной в сценарии. Архитектор всегда может прибегнуть к следующему совету:

Когда отвергаются все остальные аргументы, излагайте свою позицию в понятиях затрат финансов и времени.

Мы рекомендуем приберечь эту тактику ведения переговоров напоследок. Мы видели слишком много переговоров, начинавшихся неудачно из-за начального утверждения «это будет стоить больших денег» или «у нас нет на это времени». Деньги и время (прилагаемые усилия), конечно же, являются ключевыми факторами любых переговоров, но они должны использоваться как крайнее средство; сначала в ход идут другие обоснования и разумные объяснения.

Как только соглашение будет достигнуто, можно перейти к разговору о стоимости и времени, если они важны в обсуждении данной темы.

Еще один прием ведения переговоров, особенно в ситуации из сценария 1, заключается в следующем:

Чтобы уточнить требования, используйте правило «разделяй и властвуй».

Древний китайский стратег Сунь-Цзы в книге «Искусство войны» писал: «Если его силы едины, разделите их». Точно такая же тактика разделения и властвования может применяться архитектором при ведении переговоров. Рассмотрим тот же сценарий 1. В этом случае спонсор проекта настаивает на пяти девятках (99.999%) доступности для новой торговой системы. Но нужны ли эти пять девяток доступности всей системе? Сведение требования к конкретной области системы, действительно нуждающейся в пяти девятках доступности, сократит область сложных требований (и затрат на их удовлетворение), а также сузит саму область переговоров.

Переговоры с другими архитекторами

Рассмотрим следующий реальный сценарий (сценарий 2) ведения переговоров между ведущим архитектором и его коллегой в рамках того же проекта:

Сценарий 2

> Ведущий архитектор проекта уверен, что правильным подходом к обмену данными между группами сервисов будет асинхронный обмен сообщениями, что позволит поднять как производительность, так и масштабируемость. Но другой архитектор проекта в очередной раз категорически не согласен и уверяет, что лучшим выбором будет технология REST, поскольку она всегда быстрее обмена сообщениями и может так же хорошо масштабироваться («сам погугли и посмотри!»). Это не первый жаркий спор между ними, да и, наверное, не последний. Ведущий архитектор должен убедить другого архитектора в том, что обмен сообщениями является правильным решением.

В этом сценарии ведущий архитектор, конечно же, мог сказать коллеге, что его мнение никакой роли не играет и может быть проигнорировано, поскольку главную роль в проекте играет ведущий архитектор. Но это бы только привело к их дальнейшей вражде и к нездоровым отношениям, далеким от сотрудничества, что негативно бы повлияло на команду разработчиков в будущем. Разрешить подобные ситуации поможет следующий прием:

Всегда помните, что наглядный пример лучше спора.

Чтобы не спорить с другим архитектором по поводу использования REST вместо обмена сообщениями, ведущий архитектор может продемонстрировать ему, что обмен сообщениями в их конкретной среде будет более удачным выбором. Среды отличаются друг от друга, поэтому получить правильный ответ простым обращением к поисковику Google невозможно. Спора, вероятно, можно было бы избежать, сравнив два варианта в среде, схожей с той, в которой им предстояло работать, и показав другому архитектору полученные результаты.

Еще один весьма действенный в данной ситуации прием ведения переговоров заключается в следующем:

Избегайте острых споров и не переходите на личности — переговоры выигрываются за счет спокойного лидерства в сочетании с ясными и краткими рассуждениями.

Этот прием является весьма действенным средством в борьбе с натянутыми отношениями, подобными тем, что показаны в сценарии 2. Когда спор обостряется

и приобретает слишком личностный оттенок, лучше всего прекратить переговоры и перенести их на более поздний срок, когда обе стороны успокоятся. Споры между архитекторами, конечно же, будут, но подход к подобным ситуациям с позиции спокойного лидерства обычно заставляет оппонента умерить свой пыл.

Переговоры с разработчиками

Эффективные архитекторы ПО не пользуются своим служебным положением, чтобы указывать разработчикам, что им следует делать. Они трудятся вместе с разработчиками, зарабатывая авторитет, чтобы озвучивание требований к команде не заканчивалось спором или недовольством. Временами такая совместная деятельность дается нелегко. Зачастую команды разработчиков считают, что архитектура их не касается, поэтому в большинстве случаев они даже не в курсе решений, принимаемых архитектором. Это классический пример известного в архитектуре антипаттерна Башни из слоновой кости (Ivory Tower). Архитекторы, подпадающие под этот антипаттерн, просто навязывают свои решения, указывая командам, что делать, и не считаясь с их мнением или с их интересами. Обычно это приводит к утрате авторитета и, в конечном счете, к резкому снижению темпов работы команды. В данной ситуации может пригодиться еще один прием ведения переговоров, заключающийся в том, чтобы всегда предоставлять обоснование:

Убеждая разработчиков принять то или иное архитектурное решение или выполнить конкретную задачу, предоставьте обоснование, а не «указывайте сверху».

Когда объясняется причина, по которой что-то должно быть сделано, исполнитель с большей вероятностью согласится выполнить указание. Рассмотрим, к примеру, следующий разговор архитектора с разработчиком о создании простого запроса в рамках обычной многоуровневой архитектуры:

> **Архитектор:** Для этого вызова вам нужно пройти через уровень бизнес-логики.
>
> **Разработчик:** Нет. Намного быстрее будет вызвать базу данных напрямую.

В разговоре есть ряд недочетов. Первым делом обратите внимание на использование выражения «вам нужно». Подобный командный тон не только унизителен для оппонента, но и является наименее выигрышной завязкой переговоров или общения. Также заметьте, что разработчик отвечает на требования архитектора

контраргументами (проход через уровень бизнес-логики будет медленнее и затратнее по времени). А теперь рассмотрим другой подход к этому требованию:

> **Архитектор:** Поскольку контроль изменений для нас наиболее важен, мы сформировали архитектуру с закрытыми уровнями. То есть все вызовы к базе данных должны проходить через уровень бизнес-логики.
>
> **Разработчик:** Хорошо, я понял, но как тогда быть с производительностью в случае простых запросов?

Заметьте, что здесь архитектор обосновывает требование о том, что все запросы должны проходить через уровень бизнес-логики приложения. Объяснение причин — это всегда правильный подход к делу. Чаще всего, когда человек слышит нечто, с чем он не согласен, он перестает воспринимать информацию. Когда сразу указывается причина, архитектор может быть уверен, что обоснования будут услышаны. Также обратите внимание, что архитектор убрал все личное из этого требования. Не говоря «вы обязаны» или «вам нужно», архитектор изящно перевел требование в простую констатацию факта («То есть...»). А теперь посмотрите на ответ разработчика. Заметьте, что разговор сместился от несогласия с ограничениями, накладываемыми многоуровневой архитектурой, к вопросу о повышении производительности для простых вызовов. Теперь архитектор и разработчик могут вести конструктивный диалог в поиске способов ускорения простых вызовов, не подвергая сомнению закрытость уровней в архитектуре.

Еще одна эффективная тактика ведения переговоров, когда требуется убедить разработчика принять решение, с которым тот не согласен, заключается в том, чтобы разработчик пришел к этому решению самостоятельно. Это создает для архитектора беспроигрышную ситуацию. Предположим, к примеру, что архитектор стоит перед выбором между двумя средами разработки, фреймворком X и фреймворком Y. Архитектор понимает, что фреймворк Y не отвечает требованиям безопасности, нужным системе, и, конечно же, выбирает фреймворк X. Разработчик команды совершенно не согласен и настаивает на том, что самым удачным выбором будет фреймворк Y. Архитектор не вступает в спор и заявляет разработчику, что команда будет использовать фреймворк Y, если разработчик сможет показать, как можно будет выполнить требования безопасности при использовании этой среды разработки. Есть два варианта развития событий:

> Разработчик не сможет показать, что фреймворк Y будет отвечать требованиям безопасности, и до него наконец дойдет, что эту среду нельзя использовать. Благодаря тому что разработчик сам пришел к решению, архитектор автома-

тически получит поддержку и согласие на использование фреймворка X, по сути, превращая все это в решение разработчика. Это победа.

Разработчик найдет способ выполнения требований безопасности при использовании фреймворка Y и покажет его архитектору. Это тоже победа. В данном случае архитектор что-то упустил в оценке фреймворка Y, и выбранная среда разработки оказалась лучше других.

Если разработчик не согласен с решением, нужно позволить ему прийти к решению самостоятельно.

Только через сотрудничество с командой разработчиков архитектор может приобрести авторитет и сформировать наилучшее решение. Чем больше разработчики уважают архитектора, тем проще ему будет вести с ними переговоры.

Архитектор ПО в роли лидера

Архитектор ПО — это лидер, который ведет команду разработчиков через процесс реализации архитектуры. Мы утверждаем, что половина успеха эффективного архитектора состоит в умении работать с людьми, а также в организаторских и лидерских навыках. В этом разделе будут рассмотрены некоторые ключевые приемы, которые эффективный архитектор может использовать для управления разработчиками.

Четыре «С» архитектуры

С каждым днем все усложняется, будь то бизнес-процессы, системы и даже архитектура. Развитие идет как внутри архитектуры, так и в разработке программного обеспечения. Некоторые архитектуры отличаются особой сложностью, например, архитектура, поддерживающая шесть девяток доступности (99.9999 %), что равносильно незапланированным простоям общей продолжительностью примерно 86 мс в день, или 31.5 с в год. Сложность такого рода известна как *необходимая, или существенная, сложность* — иными словами, «у нас большая проблема».

Одна из ловушек, в которую могут угодить многие архитекторы, заключается в добавлении ненужной сложности к решениям, диаграммам и документации.

Архитекторы (как и разработчики), похоже, любят трудности. Процитируем Нила:

> *Разработчиков притягивают сложности, как мотыльков огонь, и зачастую результат один и тот же.*

Рассмотрим схему, показанную на рис. 23.1, на которой изображены основные информационные потоки для систем обработки данных в очень крупном глобальном банке. Неужели так уж необходима вся эта сложность? Ответа на этот вопрос не знает никто, а все потому, что сложность задана архитектором. Сложность такого рода называется *случайной, или несущественной, сложностью* — иными словами, «мы сделали проблему большой». Иногда архитекторы делают это, чтобы доказать свою ценность, когда задача кажется слишком простой, или чтобы гарантировать, что они всегда в курсе решений, принимаемых в отношении бизнеса или архитектуры. А иногда делают это, чтобы сохранить рабочие места. Какой бы ни была причина, добавление ненужной сложности во что-то, что не является сложным, — один из лучших способов стать неэффективным лидером.

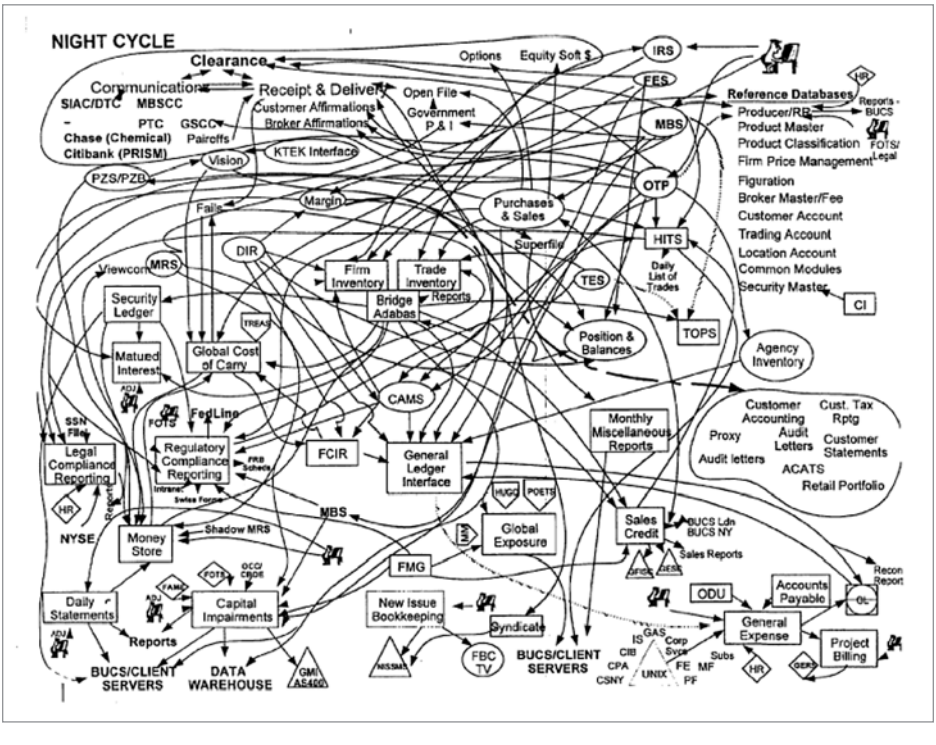


Рис. 23.1. Добавление в задачу несущественной сложности

Эффективным способом избежать несущественной сложности является применение к архитектуре так называемых четырех «С»: communication, collaboration, clarity и conciseness, то есть общение, сотрудничество, ясность и лаконичность. Все эти составляющие (рис. 23.2) необходимы эффективному руководителю команды.

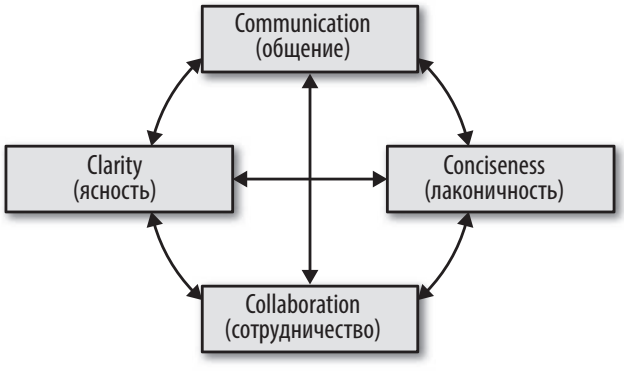


Рис. 23.2. Четыре «С» архитектуры

Крайне важно, чтобы архитектор ПО как лидер, фасилитатор и переговорщик был способен к активному общению в ясной и лаконичной манере. Не менее важно, чтобы архитектор мог сотрудничать с разработчиками, бизнес-стейкхолдерами и другими архитекторами для обсуждения и совместного формирования решений. Сосредоточенность на четырех «С» архитектуры помогает архитектору завоевать авторитет в команде и стать тем самым человеком, с которым можно поговорить о проекте и к которому все приходят не только с вопросами, но и за советами, наставничеством и руководством.

Будьте прагматичны, но дальновидны

Эффективный архитектор ПО должен быть прагматичным, но дальновидным. Все это не так просто, как кажется, и для этого требуется довольно высокий уровень зрелости и немалая практика. Чтобы лучше понять данный тезис, рассмотрим определение дальновидности:

Дальновидность

Размышления о будущем или перспективное планирование на основе богатого воображения или мудрости.

Дальновидность означает использование стратегического мышления для решения проблемы, то есть именно то, что и требуется от архитектора. Суть архитектуры и заключается в планировании на будущее и сохранении жизнеспособности архитектуры (ее актуальности) на долгое время. Но выходит так, что во время планирования и проектирования архитекторы углубляются в теорию, создавая решения, которые слишком сложны не только для реализации, но даже для понимания. А теперь рассмотрим определение прагматизма:

> *Прагматизм*
>
> Разумное и реалистичное решение задач на основе практических, а не теоретических соображений.

Архитекторам нужно быть дальновидными, но также очень важно уметь принимать практичные и реалистичные решения. Прагматичность при создании архитектурного решения — это принимать во внимание все следующие факторы и ограничения:

- бюджетные ограничения и другие факторы, связанные с финансовыми затратами;
- ограничения по времени и другие факторы, связанные с затратами времени;
- набор и уровень навыков команды разработчиков;
- компромиссы и последствия, связанные с архитектурным решением;
- технические ограничения предлагаемого архитектурного проекта или решения.

Хороший архитектор ПО — это тот, кто стремится найти баланс между прагматизмом и творческим подходом при решении стоящих перед ним задач (рис. 23.3). Рассмотрим, к примеру, ситуацию, при которой архитектор сталкивается с трудной проблемой, связанной с адаптируемостью (в силу неизвестного ранее внезапного и существенного роста одновременной нагрузки со стороны пользователей). Дальновидный архитектор мог бы предложить продуманный подход к решению этой проблемы путем использования сложной сетки данных (data mesh[1]), представляющей собой набор распределенных баз данных на основе предметных областей. В теории это могло бы стать неплохим подходом, но прагматизм предполагает проявление здравого смысла и практичности. Например, неплохо было бы поинтересоваться: применяла ли компания когда-либо прежде технологию data mesh? На какие компромиссы придется пойти при использовании этой технологии? Это действительно решит проблему?

[1] https://martinfowler.com/articles/data-monolith-to-mesh.html

Прагматичный архитектор в первую очередь посмотрит, что является ограничивающим фактором в тот момент, когда требуется высокий уровень адаптируемости. Является ли база данных узким местом? Или проблема в некоторых вызываемых сервисах либо других необходимых внешних источниках данных? Первым практическим шагом к решению этой задачи станет нахождение и изоляция узкого места. Даже если это база данных, могут ли некоторые данные кэшироваться, чтобы за ними вообще не пришлось обращаться к БД?

Рис. 23.3. Хороший архитектор найдет баланс между прагматизмом и дальновидностью

Сохранение баланса между прагматизмом и дальновидностью — прекрасный способ завоевать авторитет в качестве архитектора. Бизнес-партнеры оценят дальновидные решения, вписывающиеся в набор ограничений, а разработчики оценят практичное (а не теоретическое) решение по реализации архитектуры.

Лидерство на основе личного примера

Плохой архитектор ПО добивается от команд нужных ему действий, злоупотребляя своим положением. Эффективный архитектор поступает по-другому: руководит командой, подавая личный пример. Тем самым он завоевывает авторитет у разработчиков, бизнес-стейкхолдеров и других специалистов организации (например, у руководителя отдела эксплуатации, менеджеров по развитию и владельцев продукта).

Классическая история «авторитет, а не должность» происходила на войне с капитаном и сержантом. Будучи старшим по званию, некий капитан, находясь вдалеке от подчиненных, приказывает занять с боем важную, но хорошо обороняемую высоту. Но вместо того чтобы подчиниться капитану, солдаты, полные сомнений, смотрят на сержанта, который младше капитана по званию, — подаст

он какой-либо сигнал на взятие высоты или нет. Сержант оценивает обстановку, слегка кивает головой, и солдаты тут же с уверенностью идут в атаку.

Мораль сей истории такова: звание и должность, когда речь заходит о лидерстве, практически ничего не значат. Специалист в области computer science Джеральд Вайнберг (Gerald Weinberg) знаменит своим высказыванием: «Какой бы ни была проблема, это всегда проблема людей». Многие полагают, что решение технических проблем не имеет ничего общего с навыками работы с людьми и связано только лишь *с техническими знаниями*. Конечно, высокая техническая квалификация необходима, но это лишь часть общего уравнения для решения любой задачи. Предположим, к примеру, что архитектор проводит совещание с командой разработчиков, чтобы решить проблему, возникшую в процессе эксплуатации. Один из разработчиков вносит предложение, а архитектор в ответ заявляет: «Ну, это и вовсе глупая затея». Мало того что этот разработчик больше ничего не предложит, так и все остальные не посмеют больше ничего сказать. В этом случае архитектор фактически отстранил всю команду от совместной работы над решением.

Завоевание авторитета и лидерства — основные навыки работы с людьми. Рассмотрим следующий диалог между архитектором и заказчиком, клиентом или командой разработчиков, касающийся проблем производительности приложения:

Разработчик: Ну и как нам решать эту проблему слабой производительности?

Архитектор: Вам следует воспользоваться кэшем. Тогда проблема будет решена.

Разработчик: Не нужно говорить, что мне следует делать.

Архитектор: Я говорю, что это позволит решить проблему.

Используя выражения «Вам следует...» или «Вы должны», архитектор навязывает свое мнение разработчику, по сути, прекращая сотрудничество с ним. Это наглядный пример доведения до сведения, а не сотрудничества. А теперь рассмотрим другой диалог:

Разработчик: Ну и как нам решать эту проблему слабой производительности?

Архитектор: А вы рассматривали возможность использования кэша? С его помощью можно было бы решить проблему.

Разработчик: Нет, мы об этом не подумали. И какие у вас мысли на этот счет?

Архитектор: Мы могли бы поставить кэш вот здесь...

Обратите внимание на использование в диалоге выражений «А вы рассматривали...» или «Какие у вас мысли на этот счет?». Задавая вопрос, архитектор возвращает нить разговора разработчику или клиенту и выстраивает диалог, где оба участвуют в принятии решения. Применение правильной речи при попытке выстроить атмосферу сотрудничества играет крайне важную роль. Лидерство архитектора заключается не только в умении сотрудничать со всеми остальными специалистами при создании архитектуры, но и в организации совместной работы в команде с помощью роли фасилитатора. Постарайтесь с позиции архитектора понаблюдать за динамикой работы команды и отследите ситуации, похожие на ту, что сложилась в первом диалоге. Обучение специалистов правильной коммуникации друг с другом не только оживит работу команды, но и поможет завоевать авторитет.

Еще одним основным навыком работы с людьми, способствующим повышению авторитета и оздоровлению отношений между архитектором и командой разработчиков, является неизменное обращение к людям во время разговоров или переговоров по именам. И дело не только в том, что людям нравится слышать свое имя в ходе разговора, — подобная манера общения сближает людей. Заведите привычку запоминать имена людей и часто их использовать. Учитывая, что имена порой трудно произносятся, убедитесь, что делаете это правильно, и тренируйтесь в произношении, пока не добьетесь желаемого результата. Когда спрашиваете у человека, как его зовут, повторите ему его имя и уточните, правильно ли вы его произносите. Если нет, повторяйте до тех пор, пока не получится произнести правильно.

Если архитектор встречает кого-то впервые или же видит его от случая к случаю, он обязательно должен обменяться с ним рукопожатием, глядя ему в глаза. Рукопожатие — важная сторона общения со времен Средневековья. Физический контакт, возникающий во время простого рукопожатия, дает вам обоим понять, что вы друзья, а не враги, и формирует связь между вами. Но несмотря на то что в этом нет ничего сложного, порой бывает нелегко добиться правильного простого рукопожатия.

Пожимая кому-то руку, крепко сожмите его ладонь (но не переусердствуйте), глядя ему прямо в глаза. Отведение взгляда во время рукопожатия является знаком неуважения, и многие сразу это заметят. Также не удерживайте ладонь человека во время рукопожатия слишком долго. Крепкое рукопожатие в две-три секунды — это все, что нужно, чтобы поприветствовать кого-то или начать разговор. Можно ведь перестараться и дать человеку почувствовать себя неловко, отбивая у него желание общаться или сотрудничать с вами. Представьте, к примеру, архитектора ПО, который каждое утро приходит на работу и начинает всем пожимать руки. Это не только выглядит довольно странно, но и создает неловкую ситуацию. А теперь представьте архитектора, который должен раз в месяц

встречаться с руководителем производственного отдела. Это прекрасная возможность встать, сказать: «Привет, Рут, рад снова тебя видеть», — и обменяться с ним быстрым, крепким рукопожатием. Понимание того, когда обмениваться рукопожатием, а когда не следует этого делать, является частью весьма непростого искусства общения с людьми.

Архитектор ПО, пребывая в качестве лидера, фасилитатора и переговорщика, должен проявлять осмотрительность, соблюдая границы, существующие между людьми на всех уровнях. Рассмотренное ранее рукопожатие является эффективным и профессиональным приемом формирования физического контакта с собеседниками и с сотрудниками. Но если рукопожатия не вызывают никаких возражений, то от объятий в профессиональной обстановке, независимо от окружающей среды, следует воздержаться. Архитектор может подумать, что таким образом можно проявить особую привязанность, но иногда в рабочей обстановке они ставят другого человека в весьма неловкое положение, и, что еще важнее, могут рассматриваться как нечто совершенно недопустимое. Постарайтесь вообще обойтись без объятий, заменив их рукопожатиями (если только все в команде не обнимаются друг с другом, что со стороны выглядело бы весьма и весьма странно).

Иногда, чтобы заставить кого-то сделать что-либо против его воли, лучше превратить требование в просьбу оказать услугу. Люди вообще-то не любят, когда им указывают, что делать, но зато они обычно отзываются на просьбу о помощи. Такова уж природа человека. Рассмотрим следующий разговор между архитектором и разработчиком относительно усилий, прилагаемых к рефакторингу архитектуры в ходе весьма напряженной итерации:

> **Архитектор:** Чтобы повысить на сайте отказоустойчивость и масштабируемость, мне нужно, чтобы ты разбил сервис оплаты на пять отдельных сервисов, каждый из которых должен содержать функциональные возможности для всех видов принимаемых нами платежей: кредит в магазине, кредитная карта, PayPal, подарочная карта и бонусные баллы. Это не займет много времени.
>
> **Разработчик:** Не получится. Слишком занят, на этой итерации никак. Извини, не смогу это сделать.
>
> **Архитектор:** Послушай, это важно и должно быть сделано именно на этой итерации.
>
> **Разработчик:** Извини, не могу. Может быть, какой-нибудь другой разработчик возьмется за это. А я очень занят.

Обратите внимание на ответ разработчика. Он сразу же отвергает задачу, несмотря на то что архитектор предоставляет обоснование, говоря о повышении

отказоустойчивости и масштабируемости. В данном случае заметьте, что архитектор *поручает* разработчику выполнить дополнительную работу в условиях его слишком высокой занятости. Также заметьте, что требование высказано без обращения к разработчику по имени!

А теперь рассмотрим прием превращения требования в просьбу:

> **Архитектор:** Привет, Нейл! Послушай, я в серьезной запарке. Чтобы повысить отказоустойчивость и масштабируемость, мне крайне важно разбить платежный сервис на отдельные сервисы по каждому виду платежей, и я слишком долго ждал реализации этого замысла. Может, все-таки найдется какой-нибудь способ втиснуть эту задачу в текущую итерацию? Ты меня этим просто спасешь.
>
> **Разработчик** (*задумываясь*): Конечно, в этой итерации я занят выше крыши, но думаю, что-то может получиться. Посмотрю, что можно сделать.
>
> **Архитектор:** Спасибо, Нейл, я очень ценю твою помощь. И я у тебя в долгу.
>
> **Разработчик:** Не беспокойся, я прослежу, чтобы это было сделано в текущей итерации.

Во-первых, заметьте, что к разработчику в ходе разговора несколько раз обращаются по имени. Это придает разговору более личный характер и сближает собеседников, в отличие от обезличенного профессионального требования. Во-вторых, обратите внимание на то, что архитектор признает, что он «в серьезной запарке» и что разбиение сервиса его «просто спасет». Этот прием срабатывает не всегда, но расчет на естественную реакцию человека в плане взаимопомощи повышает вероятность успеха по сравнению с первым разговором. Опробуйте этот пример в следующий раз, когда столкнетесь с подобной ситуацией, и посмотрите на результат. В большинстве случаев он будет намного более позитивным, чем при указании, что нужно сделать.

Чтобы вести за собой команду и стать успешным лидером, архитектор ПО должен попробовать стать для команды тем самым человеком, к которому разработчики будут обращаться со своими вопросами и проблемами. Эффективный архитектор ПО воспользуется этой возможностью, проявит инициативу в организации работы независимо от своей должности или роли в команде. Когда архитектор ПО видит, что кто-то бьется над решением технической проблемы, ему необходимо вмешаться и предложить помощь или консультацию. Это же справедливо и для ситуаций, не связанных с техническими проблемами. Предположим, один из специалистов команды пришел на работу подавленным и обеспокоенным, — очевидно, что-то не так. В такой ситуации эффективный архитектор ПО заметит неладное и начнет разговор примерно так: «Привет, Ан-

тонио, я собираюсь пойти выпить кофе. Составишь мне компанию?», — а потом наедине спросит, все ли у него в порядке. Это, по крайней мере, поспособствует более личной беседе, а в лучшем случае даст возможность что-либо подсказать или навести на какую-нибудь ценную мысль. И в то же время по различным вербальным признакам и по выражению лица сотрудника эффективный лидер поймет, что он слишком навязчив, и отступит.

Еще один способ завоевать уважение и стать для команды человеком, к которому разработчики пойдут со своими вопросами и проблемами, — это периодически устраивать обеды в неформальной обстановке, чтобы поговорить о конкретной технике или технологии. У каждого читателя этой книги есть свои особые навыки или знания. На таких обедах можно не только продемонстрировать свою техническую состоятельность, но и попрактиковаться в ведении переговоров и управлении командой. Организуйте, к примеру, обеденное совещание, посвященное обзору паттернов проектирования или новому функционалу, появившемуся в последней выпущенной версии языка программирования. Это не только даст разработчикам ценную информацию, но и выделит вас как лидера и наставника в команде.

Взаимодействие с командой разработчиков

Обычно рабочий график архитектора заполнен совещаниями, многие из которых перекрываются другими совещаниями, как показано на рис. 23.4. Как при таком расписании найти время для взаимодействия с командой разработчиков, чтобы помогать им и наставлять их, а также быть доступным для возникающих у них вопросов или опасений? К сожалению, совещания в мире информационных технологий — неизбежное зло. Они бывают часто, и от них никак не избавиться.

Чтобы быть эффективным архитектором ПО, нужно уделять больше времени команде разработчиков, а следовательно, надо уметь управлять совещаниями. Архитектор может участвовать в совещаниях двух видов: навязанных (на которые он приглашен) и назначаемых (которые он созывает сам). Эти виды совещаний показаны на рис. 23.5.

Сложнее всего управлять *навязанными* совещаниями. Поскольку стейкхолдеров, с которыми архитектору необходимо общаться и сотрудничать, достаточно много, его приглашают почти на каждое запланированное совещание. Когда эффективный архитектор приглашается на встречу, он обязательно должен спросить организатора, с какой целью его позвали. Зачастую архитекторов зовут просто для того, чтобы они ознакомились с какой-то информацией. Но для этого существуют протоколы. Спрашивая «зачем?», архитектор может выбрать,

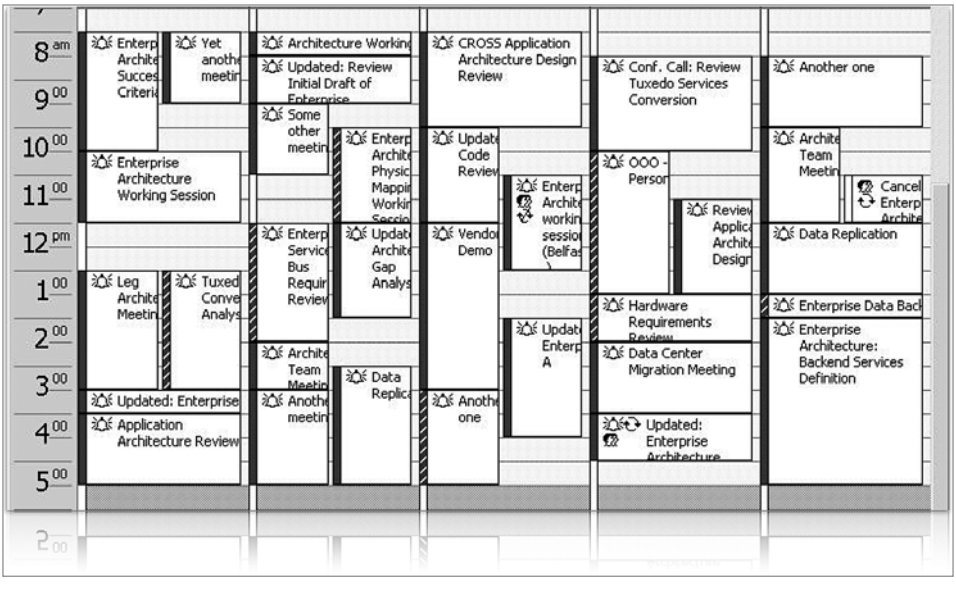


Рис. 23.4. Типичный календарь архитектора программного обеспечения

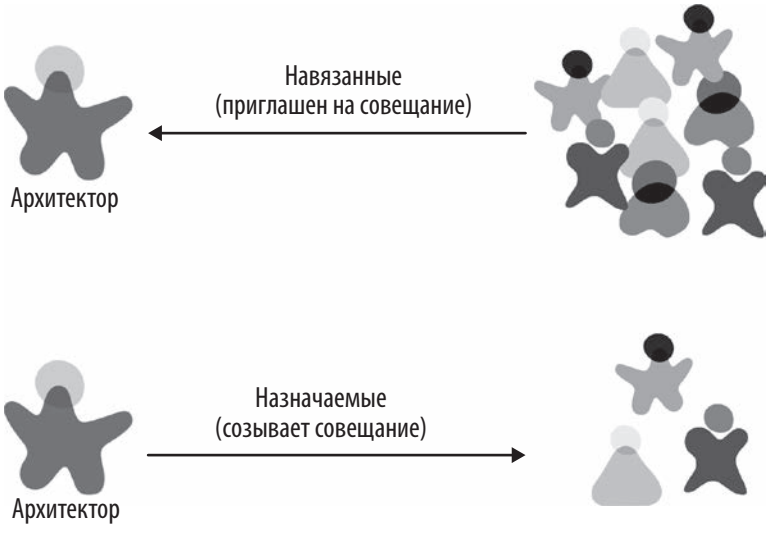


Рис. 23.5. Виды совещаний

какие совещания ему следует посетить, а какие можно пропустить. Еще один похожий прием, помогающий сократить количество совещаний с присутствием архитектора, — это заранее запросить тему. Просмотрев повестку, архитектор

может решить, действительно ли его присутствие необходимо, даже если организатор считает иначе. Кроме того, зачастую необязательно присутствовать в течение всей встречи. Зная повестку, архитектор может оптимизировать свое время и появиться в момент обсуждения интересующей его темы либо покинуть совещание после завершения соответствующего обсуждения. Не тратьте зря время на совещание, если можете его провести с командой разработчиков.

Предварительный запрос повестки дня совещания помогает определить, нужно оно вам или нет.

Еще один эффективный способ удержать команду разработчиков на правильном пути и завоевать ее уважение — быть представителем этой команды, когда разработчики также приглашены на встречу. Вместо того чтобы звать на совещание техлида, замените его собой, тем более когда на встречу приглашены и техлид, и архитектор. Это позволит команде разработчиков сосредоточиться на поставленной задаче и не тратить впустую время на совещаниях. Хотя изолирование ценных сотрудников команды от совещаний увеличивает время присутствия на них для архитектора, это повышает продуктивность разработчиков.

Совещания, назначаемые самим архитектором, время от времени необходимы, но их количество следует сокращать. Это те встречи, которые контролируются самим архитектором. Эффективный архитектор ПО всегда уточняет, не отвлечет ли он специалистов, которые могут быть заняты более важной работой. Общение по электронной почте позволяет сэкономить уйму времени. Созывая совещание по архитектурным вопросам, всегда устанавливайте повестку дня и строго ее придерживайтесь. Слишком часто обсуждение прерывается другими вопросами, которые могут быть неактуальны для других участников встречи. Кроме того, обращайте пристальное внимание на «поток» разработчика и не нарушайте его совещанием. Поток — это особое состояние сознания разработчика, в которое он попадает, когда его разум на все 100% занят решением конкретной задачи, что позволяет достичь максимальной сосредоточенности и проявить наибольшую креативность. Например, разработчик может заниматься особенно сложным алгоритмом или фрагментом кода, совершенно не обращая внимания на время. Назначая совещание, архитектор всегда должен стараться выбрать для него либо утренние часы, либо время сразу после обеда или ближе к концу дня, но не в течение дня, когда большинство разработчиков пребывают «в потоке».

Помимо совещаний, для более плотного сотрудничества с командой эффективный архитектор ПО может сделать еще одну вещь: находиться с этой командой в одном помещении. Когда архитектор работает в отдельном кабинете, это соз-

дает ощущение его «особости», а наличие стен подразумевает, что его нельзя лишний раз отвлекать и беспокоить. Если архитектор сидит рядом с командой разработчиков, это дает почувствовать его принадлежность к команде и открытость к вопросам и проблемам по мере их возникновения. Физически показывая, что он является частью команды разработчиков, архитектор укрепляет свой авторитет и расширяет возможности по оказанию помощи и наставничеству.

Иногда нет возможности организовать рабочее место рядом с командой разработчиков. В таком случае лучшее, что он может сделать, — это постоянно приходить к команде и быть у нее на глазах. Архитекторы, которые постоянно находятся на другом этаже или не выходят из своих офисов (кабинетов), не смогут оперативно помогать разработчикам в реализации архитектуры. Выделяйте время утром, после обеда или в конце рабочего дня, чтобы поговорить с командой, помочь с решением проблем, ответить на вопросы и заняться базовым коучингом и наставничеством. Сотрудники оценят такой способ общения и будут уважать вас за то, что вы уделяете им время. То же самое касается и других заинтересованных сторон. Заглянуть к руководителю отдела эксплуатации и поздороваться с ним, когда идете выпить чашку кофе, — отличный способ поддерживать открытое и доступное общение как с бизнес-партнерами, так и с другими ключевыми стейкхолдерами.

Итоги

Советы по ведению переговоров и лидерству, о которых мы рассказали в этой главе, призваны помочь архитектору ПО наладить более тесное сотрудничество с командой разработчиков и другими заинтересованными сторонами. Гибкие навыки необходимы специалисту, который стремится стать эффективным архитектором ПО. Советы, представленные в этой главе, являются неплохой стартовой точкой для того, чтобы стать еще более сильным лидером. Но возможно, самый лучший совет дал Теодор Рузвельт, 26-й президент США:

> Самая главная формула успеха — знание, как обращаться с людьми.
>
> *Теодор Рузвельт*

ГЛАВА 24

Карьерный путь

Для того чтобы стать архитектором ПО, нужно приложить немало времени и усилий. Но карьерный путь после того, как вы уже стали архитектором, ничуть не проще, как вы уже, наверное, поняли, читая эту книгу. Мы не в силах предоставить конкретный план действий для продвижения по карьерной лестнице, но можем рассказать о практических приемах, которые неплохо работают.

Архитектор должен продолжать учиться на протяжении всей своей карьеры. Мир технологий меняется с головокружительной скоростью. Один из бывших коллег Нила был весьма известным специалистом по Clipper. Он сильно сожалел, что не может взять свой огромный (и абсолютно бесполезный) багаж знаний и заменить его чем-то другим. Его также беспокоил (и продолжает беспокоить до сих пор) вопрос: а было ли в истории другое профессиональное сообщество, которое сначала накопило, а затем просто выбросило за ненадобностью все свои высокопрофессиональные наработки, как это сделали разработчики ПО?

Каждый архитектор должен следить за технологическими и бизнес-ресурсами и пополнять свой личный багаж знаний. К сожалению, перемены в этой сфере настолько стремительны, что перечислять источники информации в данной книге нет никакого смысла. Узнать о последних новостях, веб-сайтах и группах, активных в нужной области интересов, можно из разговоров с коллегами или специалистами о том, какими ресурсами они пользуются для получения актуальной информации. Архитекторы должны ежедневно отводить время для изучения этих материалов.

Правило 20 минут

Как показано на рис. 2.7 в главе 2, для архитектора важнее широта, а не глубина технического кругозора. Но на изучение нового материала уходит немало времени и сил, и этим необходимо заниматься ежедневно. Неужели у кого-то есть

время заходить на разные сайты, читать статьи, смотреть презентации и слушать подкасты? Следует признать, что это удается немногим. И разработчики, и архитекторы пытаются выдержать разумный баланс между основной работой и семейными делами, найти время на общение с детьми, на интересы и увлечения и на попытки выстраивания карьеры, а при этом еще и не упустить из виду последние тенденции и не отстать от трендов.

Один из приемов, который мы применяем для поддержания этого баланса, называется *правилом 20 минут*. Суть его (рис. 24.1) заключается в том, чтобы уделять изучению нового материала или определенной темы не менее 20 минут в день. Это время может быть достаточным для изучения таких ресурсов, как InfoQ[1], DZone Refcardz[2] и ThoughtWorks Technology Radar[3] (рис. 24.1). Этот обязательный минимум можно потратить на поиск в Google незнакомых слов (того, что в главе 2 мы отмечали как «неизвестные неизвестные вещи»), в общих чертах понять, что они означают, и переместить в разряд «известных неизвестных вещей». Или же эти 20 минут можно посвятить углубленному изучению конкретной темы, чтобы хотя бы немного продвинуться в ее освоении. Суть приема состоит в том, чтобы выкроить немного времени для профессионального самосовершенствования и постоянного расширения технического кругозора.

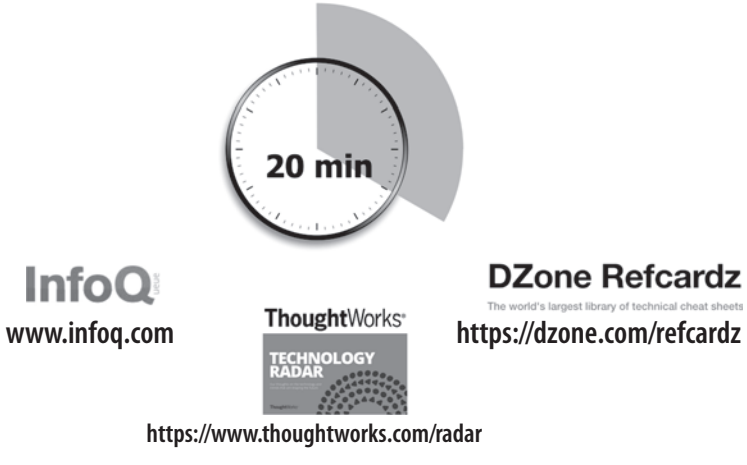


Рис. 24.1. Правило 20 минут

[1] https://www.infoq.com/

[2] https://dzone.com/refcardz

[3] https://www.thoughtworks.com/radar

Многие архитекторы соглашаются с этим и даже планируют выделять 20 минут либо в обеденный перерыв, либо вечером после работы. Мы на собственном опыте убедились, что эти намерения редко воплощаются в жизнь. Обеденные перерывы становятся все короче и короче, превращаясь в возможность наверстать упущенное в работе. С вечерами еще хуже: меняется ситуация, строятся планы, общение с семьей становится важнее, и эти 20 минут никогда не случаются.

Мы настоятельно рекомендуем применять правило 20 минут по утрам, в самом начале рабочего дня. Но и здесь нужны уточнения. К примеру, чем в первую очередь занимается архитектор, придя на работу? Ну конечно же, не отказывает себе в чашечке кофе или чая. Во вторую очередь после обязательной чашки кофе или чая просматривает почту. А далее необходимо ответить на сообщения, перенаправить некоторые другим исполнителям, а там глядишь — и день подошел к концу. Поэтому мы настоятельно рекомендуем первым делом соблюдать по утрам правило 20 минут — сразу же после того, как в руке оказалась чашка кофе или чая, и непосредственно перед тем, как все внимание будет переключено на проверку электронной почты. Приходите на работу чуть пораньше. Тогда будет время на расширение профессионального кругозора и на освоение знаний, необходимых для того, чтобы стать эффективным архитектором ПО.

Разработка персонального радара

Большую часть 90-х годов и начало нулевых Нил был техническим директором небольшой учебно-консалтинговой компании. Вначале основной платформой там был Clipper, инструментальное средство быстрой разработки DOS-приложений, представляющих собой надстройку над dBASE-файлами. Еще в те времена, когда эта платформа доживала последние дни, компания заметила взлет Windows, но бизнес-рынок до поры до времени оставался в сфере DOS... и вдруг ситуация кардинально изменилась. Этот урок оставил неизгладимое впечатление: игнорировать развитие технологий можно только себе во вред.

Нил также усвоил важный урок о технологических пузырях. Когда разработчик вкладывает много сил в технологию, он начинает жить в некоем воображаемом пузыре, похожем на своеобразную эхо-камеру. Особенно опасны пузыри, создаваемые вендорами, потому что внутри этого пузыря разработчики вообще не слышат никаких честных оценок. Но самая большая опасность жизни внутри пузыря в том, что как только этот пузырь начинает лопаться,

разработчики, находящиеся внутри, не замечают этого, пока не становится слишком поздно.

Им не хватает технологического радара: живого документа для оценки рисков и выгод существующих и зарождающихся технологий. Это понятие было введено компанией ThoughtWorks; сначала мы расскажем о его появлении, а затем рассмотрим, как создать свой персональный радар.

ThoughtWorks Technology Radar

Консультативный совет по технологиям — ThoughtWorks Technology Advisory Board (TAB) — представляет собой группу ведущих специалистов-технологов компании ThoughtWorks, созданную в помощь техническому директору доктору Ребекке Парсонс (Rebecca Parsons) для выбора технологических направлений и стратегий компании и ее клиентов. Эта группа собирается на очные встречи два раза в год. Одним из результатов такой встречи стал технологический радар — Technology Radar[1]. Со временем он постепенно превратился в технологический радар, появляющийся раз в два года.

Позже TAB стал выпускать обновленный радар каждое полугодие. А затем, как это часто бывает, проявился неожиданный побочный эффект. На некоторых конференциях, где выступал Нил, участники разыскивали его, благодарили за помощь в выпуске радара и говорили, что их компания начала создавать свою собственную версию.

Нил также понял, что это был ответ на вопрос, задаваемый практически на всех конференциях: «А как вы (докладчики) умудряетесь быть в курсе всех новых технологий? Как определяете, куда двигаться дальше?». Ответ был в том, что у всех докладчиков был некий внутренний радар.

Части

ThoughtWorks Radar состоит из четырех квадрантов, на которые разделяется сфера разработки программного обеспечения:

Инструменты

Инструментами в сфере разработки ПО считается все, начиная с инструментария разработки вроде IDE-сред и заканчивая инструментами интеграции корпоративного уровня.

[1] https://www.thoughtworks.com/radar

Языки и фреймворки

Языки программирования, библиотеки и фреймворки, как правило, с открытым исходным кодом.

Технические приемы

Любые практические действия, помогающие в разработке программного обеспечения; к ним могут относиться процессы разработки ПО, инженерные методики и советы.

Платформы

Технологические платформы, включая базы данных, решения поставщиков облачных вычислений и операционные системы.

Кольца

У радара имеются четыре кольца, перечисленные далее в направлении от внешнего кольца к внутренним:

Hold (Отложить)

Исходная идея этого кольца была: «пока что воздержаться», и оно предназначалось для слишком новых технологий, пока не поддающихся разумной оценке — наделавших много шума, но еще не доказавших свою состоятельность. Потом она превратилась в предостережение: «не стоит затевать что-то новое с применением этой технологии». Ничего страшного, если она будет использоваться в уже существующих проектах, но прежде чем встраивать ее в новые разработки, следует дважды подумать.

Assess (Оценить)

Принадлежность технологии к кольцу оценки означает, что ее следует получше изучить и разобраться, как она может повлиять на работу организации. Для этого архитекторам нужно приложить определенные усилия (такие, как активная разработка, исследовательские проекты и участие в конференциях). Например, через этот этап явно пришлось пройти многим крупным компаниям при формировании мобильной стратегии.

Trial (Опробовать)

Кольцо пробы предназначено для технологий, которым стоит последовать; важно понять, как раскрыть их возможности. Имеет смысл запустить пробный проект с низким уровнем риска, чтобы архитекторы и разработчики как следует смогли разобраться в технологии.

Adopt (Принять)

Технологии в этом кольце, по мнению компании ThoughtWorks, должны быть внедрены в реальные рабочие проекты.

Примерный вид радара показан на рис. 24.2.

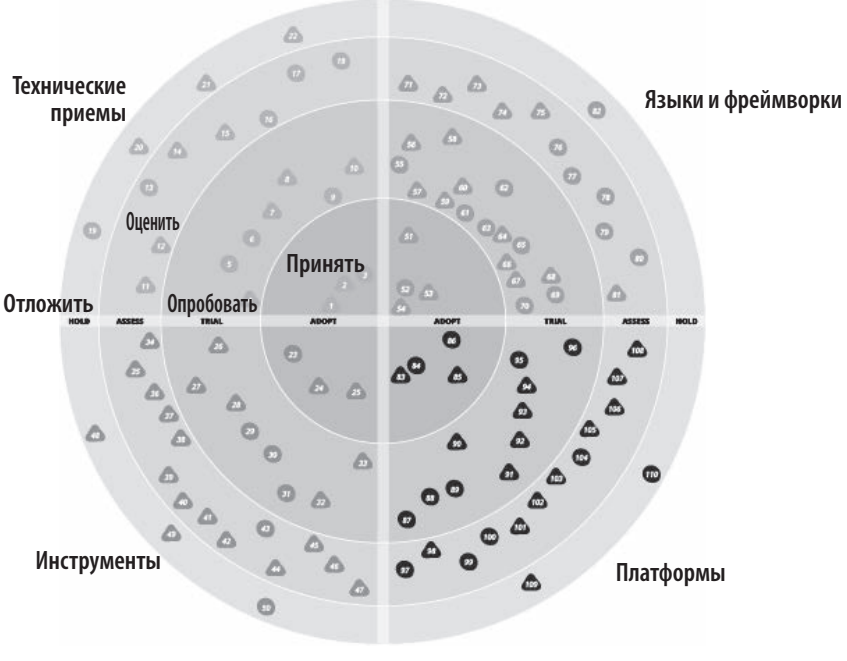


Рис. 24.2. Образец ThoughtWorks Technology Radar

Каждая метка на рис. 24.2 представляет собой технологию или технический прием с соответствующим кратким описанием.

Компания ThoughtWorks использует радар, чтобы рассказать всем о состоянии дел в мире разработки ПО, а разработчики и архитекторы с его помощью структурируют процесс собственной оценки различных технологий. Архитекторы могут применить инструмент, о котором рассказано в разделе «Средства визуализации с открытым исходным кодом» на с. 431, для создания визуальных представлений, подобных созданному в ThoughtWorks, чтобы упорядочить свои размышления о том, на что следует тратить свое время и усилия.

При использовании радара в личных целях мы предлагаем воспользоваться следующими кольцами вместо рассмотренных ранее:

Hold (Отложить)

Сюда архитектор может включать не только технологии и технические приемы, от которых следует воздержаться, но и привычки, от которых он пытается избавиться. Например, какой-нибудь архитектор склонен выискивать на форумах новости и сплетни о климате, складывающемся внутри команды. Чтение этой ленты хоть и увлекательно, но бесполезно с профессиональной точки зрения. Указание привычки в кольце выжидания станет для архитектора напоминанием о том, что от нее следует избавиться.

Assess (Оценить)

В это кольцо следует отнести перспективные технологии, о которых архитектор слышал, но пока не нашел времени на то, чтобы дать им собственную оценку (см. далее раздел «Социальные сети» на с. 431). Таким образом формируется основа для будущих более серьезных исследований.

Trial (Опробовать)

Здесь имеет смысл собрать все, что находится в активном исследовании и разработке, например то, с чем архитектор проводит отдельные эксперименты в крупной кодовой базе. К этому кольцу относятся технологии, на которые стоит потратить время, чтобы провести эффективный анализ всех возможных компромиссов.

Adopt (Принять)

В последнем кольце собираются все те новые вещи, которые вызывают особый энтузиазм архитектора, а также лучшие приемы решения конкретных проблем.

Опасно относиться с безразличием к портфелю технологий. Большинство специалистов делают выбор просто по ситуации, ориентируясь на то, что круто или же на чем настаивает работодатель. Технологический радар помогает архитектору упорядочить свои представления о технологиях и проанализировать все плюсы и минусы своего выбора (в качестве примера можно взять фактор «это крутая технология» и меньшую вероятность получить новую работу и сравнить это с огромным выбором вакансий, но менее интересной работой). Архитекторам следует рассматривать портфолио технологий как финансовый портфель: они во многом похожи. Что специалист по финансовому планированию говорит людям насчет их портфелей? Диверсифицируйте!

Архитекторам нужно выбрать ряд широко востребованных технологий и/или приемов работы и отслеживать спрос на их применение. Но также можно по-

пробовать и некоторые технологические новинки вроде разработок с открытым исходным кодом или развития мобильных направлений. Существует масса историй о разработчиках, освободившихся от кабалы за счет работы (по ночам) над проектами с открытым исходным кодом, которые обрели популярность, стали продаваться и стали в конечном итоге их высшим достижением. Это еще одна причина, по которой следует сосредоточиться на широте взглядов, а не на глубине освоения какой-то одной технологии.

Архитекторам следует выделять время на расширение своего технологического портфолио, и радар поможет структурировать это портфолио. Но процесс еще важнее результата. Создание визуализации дает возможность задуматься об этих вещах, а для слишком занятых архитекторов это способ настроиться на нужный образ мышления и выкроить время в своем плотном графике.

Средства визуализации с открытым исходным кодом

По многочисленным просьбам компания ThoughtWorks в ноябре 2016 года выпустила программу, с помощью которой специалисты могут создавать собственную визуализацию радара. Когда ThoughtWorks создает радар для компании, результаты обсуждений сводятся в электронную таблицу, где каждому квадранту отведена отдельная вкладка. Утилита ThoughtWorks Build Your Own Radar использует электронную таблицу Google для ввода данных, а сама визуализация создается с помощью HTML5. Таким образом, хотя основной задачей процесса является обобщение результатов обсуждения, в этой программе также можно создавать полезные визуализации.

Социальные сети

Где же архитектору найти новые технологии и технические приемы для помещения в кольцо оценки своего радара? В книге Эндрю Макафи (Andrew McAfee) «Enterprise 2.0» (издательство Harvard Business Review Press) встречается интересное наблюдение, касающееся средств социального общения и соцсетей в целом.

Если говорить на тему взаимоотношений между людьми, можно выделить три категории, как показано на рис. 24.3.

Сильные связи — это члены вашей семьи, коллеги и ближний круг общения. Если вы сможете ответить, что человек ел на обед хотя бы в один день на прошлой неделе, то ваше с ним общение можно назвать близким. К категории *слабых связей* относятся случайные знакомые, дальние родственники или люди, с которыми

вы видитесь лишь несколько раз в году. До появления средств социального общения поддерживать связи с людьми этого круга было непросто. И наконец, *потенциальные связи* относятся к людям, которые еще не встретились на вашем жизненном пути.

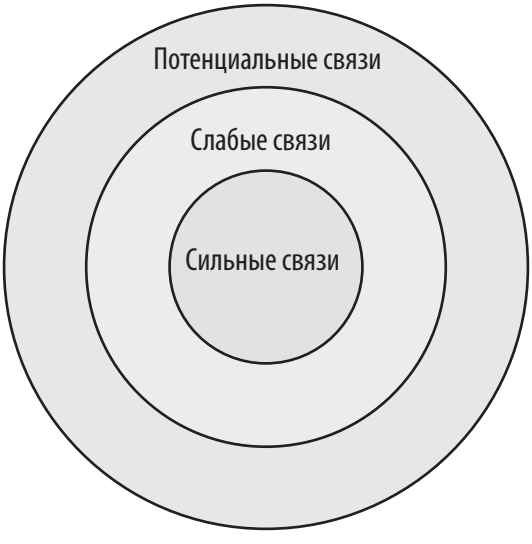


Рис. 24.3. Социальные круги общения человека

Самым интересным наблюдением, сделанным Макафи об этих связях, было то, что следующая работа скорее найдется в результате общения с людьми из категории слабых, а не сильных связей. Людям из категории сильных связей в рамках своей группы известно все обо всех, поскольку каждый постоянно на виду у других. А от тех, кто относится к категории слабых связей, могут поступать весьма неожиданные предложения, включая приглашение на новую работу.

С помощью социальных сетей архитекторы могут расширять свой технический кругозор. Профессиональное использование таких средств социального общения, как Twitter, позволяет архитекторам разыскивать специалистов, к чьим советам можно прислушаться, и становиться их подписчиками. Это позволяет архитекторам сформировать базу новых и интересных технологий, требующих изучения, а также следить за изменениями в мире технологий.

Напутствия

> Как создаются великие дизайнеры? Великие дизайнеры, разумеется, занимаются дизайном.
>
> *Фредерик Брукс*

> Откуда возьмутся великие архитекторы, если они будут заниматься архитектурой едва ли пять-шесть раз за всю их карьеру?
>
> *Тед Ньюард*

Практика — проверенный способ развития навыков и совершенствования во всех сферах жизни... включая архитектуру. Мы настоятельно рекомендуем новым и уже состоявшимся архитекторам продолжать оттачивать свои навыки как в области расширения технического кругозора, так и в области проектирования архитектуры.

Для этого ознакомьтесь с архитектурными ката[1] на сайте, сопровождающем книгу. Используйте эти примеры для отработки навыков проектирования архитектуры.

Нас часто спрашивают о ката: есть ли где-нибудь сборник ответов? К сожалению, такого решебника не существует. Процитируем Нила, одного из авторов книги:

> В архитектуре нет верных или неверных ответов, есть только *компромиссы*.

Когда в ходе реальных занятий мы стали в качестве упражнений использовать архитектурные ката, была мысль сохранить ответы, собрав работы учеников и создав из них некий сборник. Но вскоре мы отказались от этой затеи, поскольку поняли, что информация будет неполной. Другими словами, команды фиксировали только топологию и обосновывали в классе свои решения, но у них не было времени делать записи архитектурных решений. Конечно, реализация решений представляла определенный интерес, но причины их принятия были намного интереснее, поскольку здесь уже речь шла о выбранных компромиссах. Сохранение ответов на вопрос «как?» было бы лишь половиной дела, главное — «почему».

Итак, наше напутствие будет следующим: все время учитесь, все время практикуйтесь, а еще — *идите и займитесь разработкой архитектуры*!

---

[1] http://fundamentalsofsoftwarearchitecture.com/katas/

ПРИЛОЖЕНИЕ

Контрольные вопросы

Глава 1. Введение

1. Назовите четыре главные особенности, определяющие архитектуру программного обеспечения.
2. Чем архитектурное решение отличается от принципа проектирования?
3. Перечислите восемь основных ожиданий от работы архитектора.
4. Что гласит Первый закон архитектуры программного обеспечения?

Глава 2. Архитектурное мышление

1. Дайте описание традиционному подходу «архитектура vs разработка ПО» и объясните, почему этот подход больше не работает.
2. Перечислите три уровня пирамиды знаний и приведите пример для каждого из них.
3. Почему архитектору важнее широта технического кругозора, а не глубина технических знаний?
4. Назовите несколько способов, как можно поддерживать широту технического кругозора, оставаясь при этом практикующим архитектором.

Глава 3. Модульность

1. Что означает термин «коннасценция»?
2. В чем разница между статической и динамической коннасценцией?
3. Что означает коннасценция типа? К какой коннасценции она относится: к статической или к динамической?

4. Какая форма коннасценции самая сильная?
5. Какая форма коннасценции самая слабая?
6. Что предпочтительнее иметь в кодовой базе: статическую или динамическую коннасценцию?

Глава 4. Основные свойства архитектуры

1. Каким трем критериям должен соответствовать атрибут, чтобы считаться архитектурным свойством?
2. В чем разница между явным и неявным свойством? Приведите пример каждого из них.
3. Приведите пример эксплуатационных свойств.
4. Приведите пример структурных свойств.
5. Приведите пример сквозных свойств.
6. Какое архитектурное свойство важнее: доступность или производительность?

Глава 5. Выбор архитектурных свойств

1. Почему рекомендуется ограничивать количество обязательно поддерживаемых архитектурных свойств (с названиями, оканчивающимися на *-ость*)?
2. Верно ли, что большинство архитектурных свойств берется из бизнес-требований и пожеланий заказчиков?
3. Если бизнес-стейкхолдеры заявляют, что важнее всего — время выхода на рынок (например, внедрение новых функций или релиз исправленной и обновленной версии ПО), то какие архитектурные свойства необходимо поддерживать?
4. В чем разница между масштабируемостью и адаптируемостью?
5. Вы узнали, что ваша компания планирует слияния и поглощения для существенного расширения клиентской базы. О каких архитектурных свойствах следует побеспокоиться?

Глава 6. Измерение параметров архитектурных свойств и управление их соблюдением

1. Почему цикломатическая сложность является в архитектуре таким важным показателем для анализа?

2. Что такое архитектурная функция пригодности? Как такие функции могут использоваться для анализа архитектуры?
3. Приведите пример функции пригодности для оценки степени масштабируемости архитектуры.
4. Каков самый важный критерий какого-либо архитектурного свойства, на основании которого у архитекторов и разработчиков есть возможность создавать функции пригодности?

Глава 7. Область действия архитектурных свойств

1. Что такое архитектурный квант и почему он так важен?
2. Предположим, что система состоит из единственного пользовательского интерфейса с четырьмя независимо развернутыми сервисами, каждый из которых содержит собственную отдельную базу данных. Сколько квантов будет в системе, один или четыре? Почему?
3. Предположим, что в системе имеется административная часть, управляющая статическими справочными данными (например, каталогом товаров и данными по складу), и клиентская часть, управляющая размещением заказов. Сколько должно быть квантов в этой системе и почему? Если вы полагаете, что должно быть несколько квантов, могут ли административный квант и клиентский квант иметь общую базу данных? Если да, то в каком кванте она должна находиться?

Глава 8. Компонентно-ориентированное мышление

1. Мы определяем *компонент* как строительный блок приложения, то есть то, из чего делается приложение. Компонент, как правило, состоит из группы классов или исходных файлов. Как компоненты обычно проявляются в приложении или сервисе?
2. В чем разница между техническим и предметным разбиением? Приведите пример каждого из них.
3. В чем преимущество предметного разбиения?
4. При каких обстоятельствах техническое разбиение будет выгоднее предметного?
5. Что такое «Западня сущности»? Почему следование этому антипаттерну считается неудачным подходом к идентификации компонента?
6. Когда при идентификации ключевого компонента можно выбирать подход, основанный на рабочем потоке, а не на подходе актор — действие?

Глава 9. Архитектурные стили. Основы

1. Перечислите восемь заблуждений, связанных с распределенными вычислениями.
2. Назовите три проблемы распределенных архитектур, не встречающиеся в монолитных архитектурах.
3. Что такое связанность по образцу?
4. Какими способами можно решить проблему связанности по образцу?

Глава 10. Многоуровневая архитектура

1. В чем разница между открытым и закрытым уровнем?
2. Опишите концепцию уровней изоляции и назовите преимущества применения этой концепции.
3. Что собой представляет антипаттерн «архитектурная воронка»?
4. Какие основные архитектурные свойства могут побудить к выбору многоуровневой архитектуры?
5. Почему в многоуровневой архитектуре слабая поддержка тестируемости?
6. Почему в многоуровневой архитектуре слабая поддержка гибкости?

Глава 11. Конвейерная архитектура

1. Могут ли каналы в конвейерной архитектуре быть двунаправленными?
2. Назовите четыре типа фильтров и объясните их назначение.
3. Может ли фильтр отправлять данные сразу в несколько каналов?
4. К какому типу разбиения относится конвейерный архитектурный стиль: к техническому или к предметному?
5. Каким образом конвейерная архитектура поддерживает модульность?
6. Приведите два примера конвейерного архитектурного стиля.

Глава 12. Микроядерная архитектура

1. Как еще называют микроядерный архитектурный стиль?
2. При каких обстоятельствах допускается зависимость подключаемых компонентов от других подключаемых компонентов?

3. Какими инструментами и фреймворками можно воспользоваться для управления плагинами?
4. Что бы вы сделали, если бы у вас был сторонний плагин, не соответствующий стандартному контракту на плагины в ядре?
5. Приведите два примера микроядерного архитектурного стиля.
6. К какому типу разбиения относится микроядерный архитектурный стиль: к техническому или к предметному?
7. Почему микроядерная архитектура всегда имеет только один архитектурный квант?
8. Что такое изоморфизм между предметной областью и архитектурой?

Глава 13. Архитектура на основе сервисов

1. Сколько сервисов в типичной архитектуре на основе сервисов?
2. Нужно ли разбивать на части базу данных в архитектуре на основе сервисов?
3. При каких обстоятельствах могла бы возникнуть неоходимость разбить базу данных на части?
4. Каким приемом можно воспользоваться для управления изменениями в базе данных в архитектуре на основе сервисов?
5. Нужен ли контейнер (например, Docker) для запуска предметных сервисов?
6. Какие архитектурные свойства хорошо поддерживаются архитектурами на основе сервисов?
7. Почему адаптируемость плохо поддерживается в архитектурах на основе сервисов?
8. Как можно увеличить количество квантов в архитектуре на основе сервисов?

Глава 14. Архитектура, управляемая событиями

1. В чем основные различия между топологиями брокера и медиатора?
2. Какую топологию вы примените для более жесткого контроля рабочего процесса: медиатора или брокера?
3. Что обычно используется в топологии брокера, модель публикации и подписки или модель двухточечного обмена сообщениями с очередями?

4. Назовите два основных преимущества асинхронного обмена данными.
5. Приведите пример типичного запроса в модели, основанной на запросах.
6. Приведите пример типичного запроса в модели, основанной на событиях.
7. В чем разница между инициирующим событием и событием обработки в архитектуре, управляемой событиями?
8. Можете ли вы назвать методы предотвращения потерь данных при отправке и получении сообщений из очереди?
9. Какие три архитектурных свойства побуждают к использованию архитектуры, управляемой событиями?
10. Какие архитектурные свойства имеют слабую поддержку в архитектуре, управляемой событиями?

Глава 15. Архитектура на основе пространства

1. Откуда взялось название архитектуры на основе пространства?
2. Назовите основную особенность архитектуры на основе пространства, отличающую ее от всех других архитектурных стилей.
3. Назовите четыре компонента, входящие в состав виртуализированного связующего программного обеспечения в архитектуре на основе пространства.
4. Какова роль сетки обмена сообщениями?
5. Какова роль средства записи данных в архитектуре на основе пространства?
6. При каких условиях сервис будет нуждаться в доступе к данным через устройство чтения данных?
7. Чему способствует небольшой размер кэша: увеличению или уменьшению шансов на возникновение конфликтов данных?
8. В чем разница между реплицированным и распределенным кэшем? Какой из них обычно используется в архитектуре на основе пространства?
9. Перечислите три архитектурных свойства, которые лучше всего поддерживаются в архитектуре на основе пространства.
10. Почему у архитектуры на основе пространства такой низкий показатель тестируемости?

Глава 16. Оркестрированная сервис-ориентированная архитектура

1. Что является главной причиной для выбора сервис-ориентированной архитектуры?

2. Какие четыре основных типа сервисов применяются в сервис-ориентированной архитектуре?
3. Назовите некоторые факторы, из-за которых сервис-ориентированная архитектура пришла к своему закату.
4. К какому типу разбиения относится сервис-ориентированная архитектура: к техническому или к предметному?
5. Как в сервис-ориентированной архитектуре решается вопрос многократного использования кода предметных областей? А как этот же вопрос решается в отношении кода эксплуатационной части?

Глава 17. Архитектура микросервисов

1. Почему для архитектуры микросервисов такую важную роль играет концепция ограниченного контекста?
2. Каковы три способа определения надлежащей гранулярности микросервисов?
3. Какие функции могут быть заложены в sidecar-компонент?
4. В чем разница между оркестровкой и хореографией? Что из них поддерживается микросервисами? Не проще ли в микросервисах воспользоваться одним стилем обмена сообщениями?
5. Что такое сага в микросервисах?
6. Почему в микросервисах высокий уровень поддержки гибкости, тестируемости и развертываемости?
7. Каковы две причины возникновения в микросервисах проблем с производительностью?
8. К какому типу разбиения относится архитектура микросервисов: к техническому или к предметному?
9. Дайте описание топологии, при которой в экосистеме микросервисов может быть только один квант.
10. Как в архитектуре микросервисов решается вопрос многократного использования кода предметных областей? А как этот же вопрос решается в отношении кода эксплуатационной части?

Глава 18. Выбор подходящего архитектурного стиля

1. Каким образом архитектура данных (структура логических и физических моделей данных) влияет на выбор архитектурного стиля?

2. Как она влияет на ваш выбор используемого архитектурного стиля?
3. Опишите шаги, предпринимаемые архитектором для определения стиля архитектуры, принципа разбиения данных и стилей обмена данными.
4. Какие факторы могут повлиять на выбор распределенной архитектуры?

Глава 19. Архитектурные решения

1. При каких обстоятельствах может формироваться антипаттерн Covering Your Assets?
2. Какие приемы позволяют избежать антипаттерна Email-Driven Architecture?
3. Назовите пять факторов, определенных Майклом Нейгардом для идентификации архитектурной значимости.
4. Назовите пять главных разделов записи архитектурного решения (ADR).
5. К какому разделу ADR обычно добавляется обоснование архитектурного решения?
6. Предположим, что вам не нужен особый раздел «Альтернативы», тогда в каком именно разделе ADR можно перечислить альтернативы предлагаемому решению?
7. Назовите три критерия, на основе которых можно сделать в статусе ADR пометку «Предложено».

Глава 20. Анализ архитектурных рисков

1. Назовите два измерения матрицы оценки рисков.
2. Как можно показать направление конкретного риска при его оценке? Можете ли вы придумать другие способы индикации повышения или снижения степени риска?
3. Почему риск-штурм должен быть коллективным мероприятием?
4. Почему этап выявления рисков в рамках риск-штурма должен происходить в индивидуальном, а не в коллективном порядке?
5. Что вы станете делать, если три участника определили для конкретной области архитектуры высокую степень риска (6), а все остальные участники дали ей только среднюю оценку (3)?
6. Какую оценку степени риска (от 1 до 9) вы поставите неопробованным или неизвестным технологиям?

Глава 21. Составление диаграмм и проведение презентаций архитектуры

1. Что такое иррациональная привязанность к артефакту и как это влияет на документирование и построение диаграмм?
2. Что означают четыре «С» в технологии составления диаграмм C4?
3. В чем суть антипаттерна Bullet-Riddled Corpse? Как не попасть в него при проведении презентаций?
4. Какие два основных информационных канала есть у докладчиков при проведении презентации?

Глава 22. Эффективная команда

1. Назовите три типа архитекторов на основе личных качеств. Какие рамки для команды разработчиков создают эти типажи?
2. Какие пять факторов используются для определения степени контроля над командами?
3. По каким трем тревожным признакам можно заметить, что численность команды слишком велика?
4. Перечислите три основных чек-листа, подходящих для команды разработчиков.

Глава 23. Навыки лидерства и ведения переговоров

1. Почему переговоры играют важную роль в профессии архитектора?
2. Назовите некоторые приемы ведения переговоров, когда бизнес-стейкхолдер настаивает на пяти девятках доступности, а на самом деле необходимы только три девятки.
3. Что можно понять из заявления бизнес-стейкхолдера: «Мне это нужно было еще вчера»?
4. Почему в переговорах речь о времени и деньгах следует вести в последнюю очередь?
5. В чем суть правила «разделяй и властвуй»? Как его можно применить при ведении переговоров с бизнес-стейкхолдером об архитектурных свойствах? Приведите пример.
6. Перечислите четыре «С» архитектуры.
7. Объясните, почему для архитектора одинаково важно быть прагматичным и дальновидным.

8. Каковы приемы управления совещаниями и сокращения числа тех совещаний, на которые вы приглашены?

Глава 24. Карьерный путь

1. В чем суть правила 20 минут и когда его лучше всего применять?
2. Что за четыре кольца в технологическом радаре ThoughtWorks и что они означают? Как их можно применить в вашем радаре?
3. Опишите разницу между глубиной и широтой знаний применительно к архитектурам программного обеспечения. Что из этого архитекторы должны стремиться доводить до максимума?

Об авторах

Марк Ричардс — архитектор программного обеспечения с большим опытом, занимающийся выстраиванием, дизайном и реализацией архитектур на основе микросервисов и других распределенных архитектур. Он является основателем DeveloperToArchitect.com, веб-сайта, цель которого — помогать разработчикам на пути их превращения в архитекторов ПО.

Нил Форд — директор, архитектор ПО и идейный вдохновитель компании ThoughtWorks, являющейся глобальным IT-консультантом по разработке и поставке программного обеспечения на всех стадиях процесса. До работы в ThoughtWorks Нил был главным технологом DSW Group Ltd., общенациональной фирмы по развитию и подготовке кадров.

Иллюстрация на обложке

На обложке книги изображен красный веерный попугай (Deroptyus accipitrinus). Эта птица обитает в Южной Америке и известна в тех местах под несколькими именами. На испанском ее называют loro cacique, а на португальском anacã, papagaio-de-coleira или vanaquiá. Она гнездится в кронах и дуплах деревьев тропических лесов Амазонки, питаясь плодами дерева цекропии, прозванными «змеиными пальцами», а также твердыми плодами различных пальм.

Красный веерный попугай является единственным представителем рода Deroptyus, он отличается темно-красным оперением, покрывающим его загривок. Свое название он получил из-за повадки «распускать» это оперение, почуяв опасность или угрозу, демонстрируя ярко-синюю каемку, очерчивающую кончик каждого пера. На его голове — белая корона, у него желтые глаза и коричневые щеки с белыми просветами. Грудь и живот попугая имеют такое же красное оперение с синей каемкой, контрастирующее с наслоениями ярко-зеленых перьев на спине.

В декабре-январе красные веерные попугаи ищут себе пару на всю жизнь. Каждый год самка откладывает от двух до четырех яиц. Все 28 дней инкубационного периода самец заботится о самке, высиживающей яйца. Примерно через 10 недель птенцы готовы к самостоятельной сорокалетней жизни в дикой природе крупнейшего на планете тропического леса.

Красный веерный попугай считается видом, находящимся под угрозой исчезновения, как и многие другие животные, изображенные на обложках книг издательства O'Reilly. Все они важны для нашего мира.

Иллюстрацию для обложки выполнила Карен Монтгомери (Karen Montgomery) на основе старинной гравюры из книги 1894 года Ричарда Лидеккера (Richard Lydekker) «The Royal Natural History».

Марк Ричардс, Нил Форд

Фундаментальный подход к программной архитектуре: паттерны, свойства, проверенные методы

Перевел с английского С. Черников

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Е. Строганова*
Научный редактор	*Д. Зимина*
Литературный редактор	*Ю. Зорина*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 05.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.11 — Учебники печатные общеобразовательного назначения.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 17.04.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 36,120. Тираж 700. Заказ 0000.