Марк Симан

Код, который умещается в голове

Эвристики для разработчиков

ПИТЕР®

Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018
УДК 004.41
С37

Симан Марк

С37 Роберт Мартин рекомендует. Код, который умещается в голове: эвристики для разработчиков. — СПб.: Питер, 2023. — 400 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2293-6

Незаменимые практические советы по написанию кода в устойчивом темпе и по управлению сложностью, из-за которой проекты часто выходят из-под контроля. В книге описываются методы и процессы, позволяющие решать ключевые вопросы: от создания чек-листов до организации командной работы, от инкапсуляции до декомпозиции, от проектирования API до модульного тестирования. Автор иллюстрирует свои выводы фрагментами кода, взятыми из готового проекта. Написанные на языке C#, они будут понятны всем, кто использует любой объектно-ориентированный язык, включая Java, C++ и TypeScript. Для более глубокого изучения материала вы можете загрузить весь код и подробные комментарии к коммитам.

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

ББК 32.973.2-018
УДК 004.41

ISBN 978-0137464401 англ.

Authorized translation from the English language edition, entitled Code that Fits in your Head: Heuristics for Software Engineering, 1st Edition by Mark Seemann, published by Pearson Education, Inc, publishing as Addison Wesley Professional.

ISBN 978-5-4461-2293-6

КРАТКОЕ СОДЕРЖАНИЕ

ЧАСТЬ II
УСТОЙЧИВОСТЬ

ОГЛАВЛЕНИЕ

ЧАСТЬ I
РАЗВИТИЕ

ЧАСТЬ II
УСТОЙЧИВОСТЬ

Мы обязаны своим прогрессом в разработке ПО нашим предшественникам. Богатый опыт Марка позволил ему привести в своей книге философские и организационные идеи, а также дать конкретные рекомендации по написанию кода. Вам предоставлена удивительная возможность перенять этот опыт, чтобы помочь себе развиваться. Воспользуйтесь им.

Адам Ральф, спикер, преподаватель и специалист по упрощению особого программного обеспечения

Я читаю блог Марка уже много лет. Каждый его пост несет в себе что-то интересное, в то же время раскрывая глубокие технические познания автора. Книга «Код, который умещается в голове» написана по тому же принципу: она передает огромное количество исчерпывающей информации разработчикам ПО, желающим вывести свои навыки на новый уровень.

Адам Торнхилл, основатель CodeScene, автор книг Software Design X-Rays и Your Code as a Crime Scene

В этой книге мне больше всего нравится то, что в качестве рабочего примера используется единая база. В вашем распоряжении находятся не отдельные части кода, а единый репозиторий Git с полным приложением. Его история создана вручную, чтобы показать эволюцию кода наряду с концепциями, объясняемыми в книге. Читая о конкретном принципе или технике, вы найдете прямую ссылку на код, иллюстрирующий изложенную идею. Конечно, вы можете изучать проект выборочно, останавливаясь на интересующих вас этапах, чтобы проверить, отладить код и поэкспериментировать с ним. Никогда прежде я не встречал в книгах интерактив такого уровня, и это доставляет мне особую радость, потому что здесь по-новому используются преимущества уникальной разработки Git.

Энрико Кампидольо, независимый консультант, спикер и автор Pluralsight

Марк Симан не только опытный архитектор и разработчик больших программных систем, но и один из ведущих идеологов в области масштабирования и управления сложными взаимоотношениями между такими системами и командами, которые их создают.

Майк Хэдлоу, внештатный консультант
по программному обеспечению и блогер

Марк Симан известен тем, что ясно и подробно объясняет самые сложные концепции. В этой книге свой обширный опыт разработки ПО он превратил в структуру практических и прагматичных методов для написания устойчивого и читабельного кода. Эта книга — мастхэв для каждого программиста.

Скотт Влашин, автор книги
Domain Modeling Made Functional

Марк пишет: «Успешное ПО долговечно», и эта книга поможет вам писать как раз такое ПО.

Брайан Хоган, архитектор ПО, подкастер, блогер

Марк обладает необычайной способностью помогать другим глубоко задуматься об отрасли и профессии разработчиков ПО. После каждого шоу на «.NET Rocks!», которое я сам веду, я еще раз прослушиваю запись, чтобы по-настоящему разобраться во всем, что мы обсуждали.

Ричард Кэмпбелл, соведущий на шоу «.NET Rocks!»

Моим родителям.

Матери Улле Симан за внимание к деталям, которое я перенял от нее.

Отцу Лейфу Симану, который научил меня играть против правил.

ПРЕДИСЛОВИЕ РОБЕРТА МАРТИНА

Мой внук учится программировать.

Да-да, вы не ошиблись. Мой 18-летний внук учится программировать на компьютере. Его обучает моя младшая дочь, которая родилась в середине 80-х и полтора года назад решила сменить профессию инженера-химика на программиста. Где они оба работают? У моего старшего сына, который вместе со своим младшим братом открывает вторую консалтинговую компанию по разработке ПО.

Да, программное обеспечение рулит в нашей семье. Конечно же, и я не отстаю: тоже программирую уже долгое-долгое время.

Не так давно дочь привезла мне своего сына, и я целый час обучал его основам программирования. Мы начали с кортежей, затем я прочитал ему лекцию о том, что такое компьютеры, как они появились, как выглядели в самом начале и... Ну вы знаете.

К концу нашего занятия я рассказал о разработке алгоритма для умножения двух двоичных целых чисел на языке ассемблера PDP-8. Если вы не в курсе, в PDP-8 нет команды умножения — нам пришлось самостоятельно разрабатывать ее. Там даже не было команды вычитания: для этого нужно было использовать дополнение до двух и добавлять псевдоотрицательное число (чтобы вы понимали всю сложность ситуации).

Когда я закончил с примером кодирования, мне показалось, что я напугал своего внука до смерти. Я имею в виду, что в мои 18 лет этот процесс будоражил меня, но, вероятно, он был не так интересен мо-

лодому парню, чья тетя мечтала научить его писать простенький код на Clojure.

Так или иначе, я задумался, насколько сложен на самом деле процесс программирования. Да-да, именно так. Очень сложен. Думаю, это самое трудное, что сотворило человечество.

Я не имею в виду написание кода для вычисления набора простых чисел, или последовательности Фибоначчи, или простой пузырьковой сортировки. Это все не так уж и сложно. Но что насчет авиадиспетчерской системы? Системы выдачи багажа? Складского учета? Игры Angry Birds? Вот что действительно сложно. Очень, очень сложно.

Я знаком с Марком Симаном на протяжении вот уже нескольких лет, но не помню, чтобы мы встречались в жизни. Мы никогда не видели друг друга, но довольно часто общаемся в профессиональных новостных группах и социальных сетях. Он один из моих любимчиков, тот, с кем я всегда буду рад подискутировать.

Наши мнения расходятся во многих областях. Мы спорим о противостоянии статической и динамической типизаций, об операционных системах и языках, полемизируем на темы многих интеллектуально сложных вещей. Но к этому процессу нужно подходить очень осторожно, ведь логика аргументов Марка безупречна.

Увидев эту книгу, я подумал: «Как здорово будет прочесть ее и поспорить с автором». Именно так и произошло. После прочтения я остался не согласен с некоторыми аспектами и с воодушевлением начал искать способы изменить мнение Симана. Мне кажется, что в некоторых случаях у меня это даже получилось... по крайней мере в уме.

Но дело не в этом. Суть в том, что разработка ПО — это сложно, и бóльшая часть последних семи десятилетий потрачена на поиски способов упростить этот процесс. В своей книге Марк собрал все лучшие идеи за эти 70 лет под одной обложкой.

Более того, он структурировал эти мысли в набор эвристических методов и приемов и привел их в том порядке, в котором нужно их выполнять. Являясь взаимосвязанными, эти техники помогают переходить от этапа к этапу в процессе разработки ПО.

По сути, Марк работает над проектом ПО на протяжении всей книги, объясняя каждый этап и эвристические методы и приемы, которые будут полезны в том или ином случае.

Марк использует язык C# (с чем я не согласен ;-)), но это не главное. Код прост, и эвристические методы и приемы применимы к любому другому языку программирования. Автор охватывает такие темы, как контрольные списки, TDD (разработка через тестирование), CQS (разделение команд и запросов), Git, цикломатическая сложность, ссылочная прозрачность, вертикальные срезы, устранение легаси-кода и разработка outside-in («от общего к частному»), — это лишь некоторые из них.

Вообще, жемчужины знаний встречаются повсюду в этой книге. Я имею в виду, что, читая главу, вы можете внезапно встретить что-то вроде: «Разверните свою тестовую функцию на 90 градусов и посмотрите, сможете ли вы ее сбалансировать по принципу AAA», или «Цель не в том, чтобы писать код быстро. Цель — в устойчивом программном обеспечении», или «Сохраните схему базы данных в репозитории».

Некоторые из этих жемчужин — бесценные знания, некоторые — праздные комментарии, а некоторые — домыслы автора, но все они говорят о глубоком понимании темы, которое Марк приобрел за эти годы.

Внимательно прочтите эту книгу. Задумайтесь о безупречной логике Марка. Внедрите описанные эвристические подходы и приемы. Встретив одну из жемчужин, остановитесь на минутку и поразмышляйте, и, возможно, когда придет время читать лекции своим внукам, вы не напугаете их, как я.

Роберт Мартин

Будущее уже здесь — оно просто
не очень равномерно распределено.

Уильям Гибсон

ВВЕДЕНИЕ

Во второй половине 2000-х я устроился научным редактором в издательство. Ознакомившись с некоторыми из моих работ, редактор связался со мной, чтобы обсудить книгу о внедрении зависимостей.

Наши первые переговоры были немного странными. Обычно, когда ко мне обращались по поводу книги, у нее уже как минимум были план и автор, но не в этот раз. Редактор просто решил узнать, востребована ли такая тема.

Поразмышляв несколько дней, я решил, что тема интересная, но не видел необходимости в целой книге. Аудитории были доступны посты в блогах, техническая документация, журнальные статьи и даже несколько книг (на смежные темы).

Немного погодя, я понял, что доступная информация разбросана по источникам, а термины в ней непоследовательны и противоречивы. Эти знания нужно было объединить и поработать над общей терминологией.

Два года спустя я стал гордым автором опубликованной книги.

По прошествии нескольких лет я задумался о том, чтобы написать еще одну книгу. Не о внедрении зависимостей, а на какую-то другую тему. Потом у меня появилась третья, четвертая, но не хватало именно этой.

Еще через десять лет я начал понимать, что, консультируя команды по написанию лучшего кода, я предлагаю методы, которым научился у гораздо более крутых специалистов, чем я сам. И я снова осознал, что бóльшая часть этих знаний доступна, но разбросана по разным

источникам и мало кто связал их в последовательное описание принципов разработки ПО.

Опыт работы над первой книгой позволил мне понять, что полезно собирать разрозненную информацию и последовательно ее излагать. Это издание — очередная моя попытка создать такой ресурс.

ДЛЯ КОГО ЭТА КНИГА

Книга предназначена для опытных программистов. Предполагается, что они уже имели дело с неподдерживаемым кодом, с неудачными проектами по разработке ПО и стремятся к решению этих проблем.

Основная аудитория книги — разработчики корпоративного ПО, в частности бэкенд-разработчики. Я много лет работал в этой области — по сути, книга отражает мой личный опыт. Но если вы фронтенд-разработчик, разработчик игр, DevOps-инженер и т. п., то, думаю, вы все равно извлечете для себя много полезной информации.

Надеюсь, вам будет удобно читать код на компилируемом, объектно-ориентированном языке семейства C. Хотя бóльшую часть своей карьеры я программировал на C#, я многому научился из книг, где код был написан на C++ или Java[1]. В этой книге примеры кода приведены на C#, но я надеюсь, что программисты на Java, TypeScript и C++ тоже найдут их полезными.

Исходные требования

Это книга не для начинающих. Несмотря на то что в ней рассматриваются вопросы организации и структурирования исходного кода, вы не найдете здесь самых основ. Предполагается, что вы уже понимаете, чем полезны отступы для форматирования кода, почему длинные методы вызывают проблемы, а глобальные переменные — это плохо и т. д. Не обязательно зачитывать до дыр книгу «Совершенный код» [65], но желательно знать основную информацию, описанную там.

[1] Если вам интересно, какие книги я имею в виду, загляните в библиографию.

Примечание для архитекторов ПО

Термин «архитектор» каждый понимает по-своему, даже если контекст ограничен разработкой программного обеспечения. Одни архитекторы фокусируются на общей картине, помогают организации в целом преуспеть в своих начинаниях. Другие глубоко разбираются в коде и заботятся в основном об устойчивости конкретной кодовой базы.

Я отношусь к последнему типу. Моя задача — организовать исходный код так, чтобы он отвечал долгосрочным бизнес-целям. Я пишу о том, что знаю, поэтому в первую очередь эта книга будет полезна архитекторам именно этой направленности.

Здесь мы не будем говорить о методах анализа компромиссов в архитектуре (ATAM), анализа видов и последствий отказов (FMEA), об обнаружении сервисов и т. д. Описание таких архитектур выходит за рамки этой книги.

СТРУКТУРА КНИГИ

Хотя это пособие о разных методологиях, я структурировал его на примере кода, который используется на протяжении всей книги, чтобы материал был более убедительным, чем в типичном каталоге паттернов программирования. С этой же целью я внедряю практики и эвристические методы, лишь когда они соответствуют повествованию. Также отмечу, что методы здесь расположены именно в том порядке, в котором я обычно представляю их, когда обучаю команды.

Повествование структурировано вокруг примера системы бронирования столиков в ресторане. Исходный код доступен по адресу informit.com/title/9780137464401.

Чтобы эту книгу можно было использовать в качестве справочника, в конце я добавил приложение со списком всех приемов и ссылками на разделы.

О стиле кода

Примеры кода написаны на языке C#, который быстро развился за последние годы. В нем внедряется все больше синтаксических идей из функционального программирования. Например, пока я писал эту книгу, были выпущены *неизменяемые типы записей*. Некоторые из подобных новейших функций языка я решил не учитывать здесь.

Когда-то код Java был очень похож на код C#. Современный же код C# далек от него.

Я хочу, чтобы код смогли понимать как можно больше читателей, и надеюсь, что так же, как я многому научился из книг с примерами на Java, люди смогут использовать это издание, не зная новейшего синтаксиса C#. Поэтому здесь я буду придерживаться консервативного подхода разработки на C#, который должен быть понятен другим программистам.

Это никак не влияет на концепции из книги. Да, возможно, в некоторых случаях есть более лаконичная альтернатива, специфичная для C#, но это лишь означает наличие возможности дополнительных улучшений.

Типизировать явно или неявно

Ключевое слово `var` было введено в C# в 2007 году. Оно позволяет объявить переменную без явного указания ее типа. Вместо этого компилятор определяет тип из контекста. Для ясности: переменные, объявленные с помощью `var`, точно так же статически типизированы, как и те, что объявлены с явными типами.

Долгое время использование этого ключевого слова ставилось под сомнение, но сегодня его применяют большинство людей. Я тоже так делаю.

Хотя я использую `var` в своих проектах, написание кода для пособия — это немного другой контекст. Обычно интегрированная среда разработки (IDE) всегда под рукой. Современные среды могут быстро определить тип неявно типизированной переменной, в отличие от книги. Поэтому я иногда явно объявляю переменные.

В большинстве примеров по-прежнему используется ключевое слово `var`, так как оно делает код короче, что особенно важно для книги, ширина строки которой ограничена. Но в некоторых случаях я намеренно явно объявляю тип переменной, чтобы облегчить понимание кода при чтении в печатном формате.

Примеры кода

Большинство листингов взяты из одного примера кодовой базы. Я использовал репозиторий Git, а примеры кода брал с разных этапов разработки. Каждый такой пример содержит относительный путь к соответствующему файлу, а часть пути к файлу служит идентификатором коммита Git.

Например, в листинге 2.1 приведен такой относительный путь: Restaurant/f729ed9/Restaurant.RestApi/Program.cs. Это значит, что пример взят из коммита с идентификатором `f729ed9` из файла `Restaurant.RestApi/Program.cs`. Проще говоря, чтобы просмотреть эту конкретную версию файла, вам нужно перейти в следующий коммит:

```
$ git checkout f729ed9
```

После этого вы можете исследовать файл `Restaurant.RestApi/Program.cs` в его контексте.

ПРИМЕЧАНИЕ К БИБЛИОГРАФИИ

В библиографии вы найдете много ресурсов, включая книги, посты в блогах и видеоролики, большинство из которых есть в интернете, поэтому я указываю URL-адреса. При работе над книгой я старался добавлять в основном те источники, которые есть в свободном доступе в интернете.

Но все меняется. И если на момент, когда вы читаете эту книгу, URL-адрес недействителен, обратитесь к интернет-архиву. Сейчас лучший сайт для этого https://archive.org, но он тоже может исчезнуть в будущем.

О моих книгах

Помимо сторонних ресурсов, в библиографии вы найдете и список моих трудов. Я понимаю, что ссылаться на себя — не слишком научный подход, и не воспринимаю написанные мной публикации как некий козырь в рукаве. Свои работы я добавил для тех, кто заинтересован в получении более подробной информации. Я цитирую себя лишь для того, чтобы вы могли найти расширенный аргумент или более подробный пример кода в ресурсе, на который я ссылаюсь.

БЛАГОДАРНОСТИ

Я хотел бы поблагодарить свою жену Сесиль за любовь и поддержку на протяжении всех совместно прожитых лет и детей Линею и Ярла, которые старались не доставлять нам хлопот.

Также я благодарен своему бесценному старому другу Карстену Стрёбеку, который не только терпел мои выходки четверть века, но и стал первым рецензентом этой книги. Он помог мне с советами и приемами LaTeX и добавил больше записей в алфавитный указатель, чем я.

Хочу сказать спасибо Адаму Торнхиллу за его отзыв о разделе, посвященном его работе.

Я в долгу перед Дэном Нортом за идею названия этой книги, которая могла выйти еще в 2011 году [72].

ОТ ИЗДАТЕЛЬСТВА

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.

ОБ АВТОРЕ

Марк Симан — бывший экономист, который в итоге нашел себя в программировании. Трудился веб-разработчиком и разработчиком корпоративных продуктов с конца 1990-х годов. В молодости Марк мечтал стать рок-звездой, но, к сожалению, по его словам, не обладал ни талантом, ни внешностью — зато потом стал сертифицированным разработчиком Rockstar. Он написал удостоенную премии Jolt[1] книгу о внедрении зависимостей, принял участие в сотнях международных конференций и записал видеокурсы для Pluralsight и Clean Coders. Марк регулярно ведет блог с 2006 года. Сейчас он живет в Копенгагене с женой и двумя детьми.

[1] Премия Jolt Award известна как «“Оскар” индустрии программного обеспечения». — *Примеч. ред.*

I Развитие

Первая часть книги построена вокруг процесса программирования. Все примеры кода, от создания первого файла до завершения написания первой функции, соответствуют стандартному образцу кодовой базы.

В начале я буду подробно объяснять все изменения в коде, но по мере продвижения по главам некоторые детали будут опущены. Представленные примеры помогут вам в освоении новых методов и применении их на практике.

Если я упустил какую-то деталь, о которой вы хотели бы узнать больше, можете обратиться к репозиториям Git, ссылка на которые приведена в книге. Каждому листингу кода присвоен свой ID, идентифицирующий определенный источник.

Вам может показаться, что представленная история коммитов безупречна и что я не допустил ни одной ошибки. Но это не так.

Я, как и любой другой человек, могу ошибаться. И одно из преимуществ Git — в возможности переписывать свою историю. Чтобы достичь желаемого результата, мне несколько раз приходилось перебазировать эту часть репозитория. Все это я сделал не для сокрытия своих ошибок, а для того, чтобы они не мешали читателям, которые будут пользоваться моим репозиторием.

Представленные примеры помогут вам лучше понять и использовать описанные мною методы. В этой части книги вы увидите *развитие* кода от начала до развернутой функции. Но даже если вы не занимаетесь разработкой с нуля, методы, изложенные здесь, помогут вам в повышении личной эффективности.

1 ИСКУССТВО ИЛИ НАУКА?

Вы ученый или художник? Инженер или ремесленник? Садовник или повар? Поэт или архитектор? Наконец, вы программист или разработчик ПО? Кем вы себя считаете?

Мой ответ таков: «Никем из вышеперечисленного. Я программист, но в то же время немного и повар, и садовник, и художник, и строитель».

Важно задавать такие вопросы. Индустрии разработки ПО в общей сложности около 70 лет, и мы все еще не до конца разобрались в этой области. Основная проблема здесь в том, *как мы думаем* о ней. Процесс разработки программного обеспечения похож на строительство дома? Или он напоминает стихосложение?

Десятилетиями программирование сравнивали с чем угодно, даже с возделыванием сада, но лучшей метафоры так и не нашли.

Я считаю, что то, как мы думаем о разработке ПО, влияет на то, как мы работаем. Программист должен подстраиваться под свои проекты. Он должен понимать, что и для чего он пишет.

1.1. СТРОИТЕЛЬСТВО ЗДАНИЯ

Многие годы разработку ПО сравнивали со строительством здания.

Кент Бек сказал об этом так:

> *«К сожалению, разработка программного обеспечения была скована метафорами физического проектирования» [5].*

Это одно из самых распространенных, неоднозначных и вводящих в заблуждение мнений.

1.1.1. Проблема проекта

Полагая, что разработка ПО похожа на строительство здания, вы совершаете первую ошибку — думаете об этом процессе как о *проекте*. У проекта есть начало и конец. Как только вы дойдете до конца, работа будет сделана.

Полностью завершить можно только неудачное ПО, успешное же будет долговечным. Качественное программное обеспечение предполагает постоянное развитие, которое может длиться годами, а иногда и десятилетиями[1].

После того как здание построено, люди могут в него заселиться. Чтобы поддерживать дом в исправном состоянии, его нужно обслуживать, но затраты на это будут в разы меньше по сравнению с затратами на его проектирование. Конечно, такой софт есть. Например, вы создали внутреннее бизнес-приложение для какой-нибудь корпорации, оно завершено, и пользователи привязаны к нему. По завершении разработки такое ПО переходит в режим обслуживания и сопровождения.

Но большинство конкурентоспособных программных продуктов никогда не будут завершены. Если вы все еще связываете процесс разработки со строительством здания, можете сравнивать его с серией проектов. Вы можете запланировать выпуск следующей версии своего

[1] Эта книга сверстана в LaTeX — ПО, первая версия которого была выпущена в 1984 году!

продукта через девять месяцев, но, к своему ужасу, обнаружите, что ваш конкурент внедряет улучшения каждые три.

Вы начинаете усердно работать над тем, чтобы сократить свои «проекты». И к моменту, когда у вас наконец получится выпускать продукт каждые три месяца, ваш конкурент будет внедрять обновления уже ежемесячно. Вы понимаете, к чему все идет?

Это может превратиться в бесконечную погоню за наращиванием функциональности и выпуском новых версий [49] или привести к разорению. В книге «Ускоряйся!» [29] приводятся научно подкрепленные аргументы того, что ключевым свойством, отличающим высокоэффективные команды от низкоэффективных, служит способность мгновенно обновлять и распространять информацию.

Если вы будете способны это сделать, понятие *проекта* разработки программного обеспечения потеряет свою актуальность.

1.1.2. Этапы разработки

Еще одно заблуждение, связанное с метафорой строительства: ПО нужно разрабатывать в несколько *этапов*. Перед началом работ архитектор создает чертеж. Далее оценивается логистика, на площадку поставляются материалы, и только после этого можно приступать к постройке здания.

В случае если метафора уместна, вы назначаете *архитектора программного обеспечения*, в обязанности которого входит создание плана. Только после этого можно бужет начать разработку ПО. Этот этап — этап проектирования — довольно сложный интеллектуальный процесс. Если возвращаться к аналогии со строительством, то он похож на фактический этап самих строительных работ, где разработчики — это взаимозаменяемые сотрудники[1], вроде машинистов.

Но это очень отдаленное сравнение. Как указал Джек Ривз в 1992 г. [87], этап *создания* программного обеспечения — это когда вы компилируете исходный код. По сравнению со строительством здания этот про-

[1] Ничего не имею против строителей — мой отец был каменщиком.

цесс можно назвать почти бесплатным. Вся работа происходит на этапе проектирования, или, как выразился Кевлин Хенни:

> *«Недвусмысленное описание программы и программирование — это один и тот же процесс» [42].*

В рамках разработки ПО мы не можем говорить об этапе строительства. Это не означает, что проектирование бесполезно, но указывает на то, что метафора с с постройкой здания здесь неприменима.

1.1.3. Зависимости

Строительство ведется в соответствии с определенными нормами и требованиями: сначала нужно заложить фундамент, затем возвести стены и только потом можно устанавливать крышу. Другими словами, все эти процессы взаимосвязаны и взаимозависимы.

Такая аналогия внушает ложную идею того, что зависимостями можно управлять. Я знаком с менеджерами, которые для планирования проекта составляли сложные диаграммы Ганта.

Я работал со многими командами, и большинство из них начинают любой проект с разработки схемы реляционной базы данных (БД). БД — основа большинства онлайн-сервисов, и ни один разработчик не будет спорить с тем, что можно спроектировать пользовательский интерфейс еще до появления базы данных.

Некоторым командам иногда даже не удается создать полностью рабочее ПО. После того как БД спроектирована, они решают, что необходимо создать так называемый каркас приложения, или *фреймворк*. Они продолжают заново изобретать ORM (Object-Relational Mapping, объектно-реляционное отображение), этот Вьетнам computer science [70].

Метафора строительства дома вредна — она заставляет вас думать о разработке ПО определенным образом. Вы упустите возможности, которых не видите из-за того, что ваша точка зрения не совпадает с реальностью. Образно говоря, разработку программного обеспечения вы *можете* начать с установки крыши. Немного позже я подкреплю эти слова примером.

1.2. ВОЗДЕЛЫВАНИЕ САДА

Мы выяснили, что аналогия со строительством не подходит, но, возможно, другие подойдут больше. В 2010-х годах становится популярной метафора возделывания сада. Не случайно Нат Прайс и Стив Фримен назвали свою книгу *Growing Object-Oriented Software, Guided by Tests* [36].

В этом примере ПО сравнивается с живым организмом, который требует особенного отношения, вложения большого количества сил и внимания. Это еще одна вполне убедительная метафора. Вы когда-нибудь думали о том, что кодовая база живет своей жизнью?

Возможно, будет правильнее рассматривать программное обеспечение именно с этой точки зрения? По крайней мере, так мы сможем не ограничиваться только строительством здания.

Теперь, в рамках аналогии с взращиванием сада, мы делаем акцент на обрезке (сокращении). Если не ухаживать за зеленью, она начнет разрастаться. Предотвратить этот процесс может садовник, избавляясь от сорняков и поддерживая культурные растения. При разработке ПО это помогает сосредоточиться на действиях, *предотвращающих* «загнивание» кода, таких как рефакторинг и удаление мертвого кода.

Но мне кажется, что эта метафора тоже не дает нам полного представления о процессе разработки ПО.

1.2.1. Что заставляет сад расти?

Мне нравится, что аналогия с садоводством делает упор на действиях, предотвращающих беспорядок. Точно так же, как вы должны ухаживать за своим садом, вам необходимо проводить рефакторинг и погашать технический долг в своих кодовых базах.

С другой стороны, эта метафора мало говорит о том, откуда берется код. В саду растения растут сами по себе: все, что им нужно, — удо-

брения, солнечный свет и вода. ПО само по себе развиваться не будет. Вы не можете просто забросить компьютер, чипсы и колу в темную комнату и ожидать, что из этого вырастет программное обеспечение. Все это не будет работать без самой важной составляющей — программистов.

Код должен быть написан кем-то. Это активный процесс, и здесь аналогия с садом становится уже не такой актуальной. Как вы решаете, что писать, а что — нет? Как принимаете решение о том, *как* структурировать код?

Мы должны ответить на эти вопросы, если хотим улучшить индустрию разработки программного обеспечения.

1.3. С ТОЧКИ ЗРЕНИЯ ИНЖЕНЕРИИ

Для разработки ПО есть и другие метафоры. Например, термин *«технический долг»*, который я упоминал ранее, можно сравнить с финансовым кредитом. А процесс *написания* кода напоминает некоторые виды авторской деятельности. Все эти аналогии в какой-то степени верны, но ни одна из них не будет абсолютно правильной.

Но я начал эту книгу именно с аналогии со строительством здания. И на то есть несколько причин. Во-первых, это сравнение довольно распространено. Во-вторых, оно кажется настолько неправильным, что его уже невозможно спасти.

1.3.1. Программирование как ремесло

К выводу, что аналогия со строительством вредна, я пришел много лет назад. И как правило, после отказа от одной теории вы обычно начинаете искать другую. Я нашел ее в *ремесле программного обеспечения.*

Давайте рассмотрим разработку ПО как ремесло, ведь, по сути, это и есть *квалифицированный труд.* Вы *можете* получить образование

в области computer science, но это вовсе не обязательно. У меня, например, его нет[1].

Навыки, необходимые профессиональным разработчикам ПО, обычно зависят от ситуации. Изучите, как устроена конкретная кодовая база, научитесь использовать конкретный фреймворк, потратьте время на исправление ошибок в продакшене. От вас будет требоваться что-то вроде этого.

Чем больше вы что-то делаете, тем опытнее вы становитесь. Если вы останетесь в одной компании и будете годами работать с одной и той же кодовой базой, вы можете стать специалистом. Но как это поможет вам при устройстве на другую работу?

Вы будете развиваться быстрее, переходя от одной кодовой базы к другой. Освойте бэкенд- и фронтенд-разработку. Изучите программирование игр или машинное обучение. Так вы гарантированно сможете накопить полезный опыт.

Становление разработчика ПО подобно старой традиции *странствующего подмастерья* в Европе. Ремесленник, плотник или кровельщик путешествовал по разным городам и странам, вкладывая все свои силы в ремесло. Все это открывало большие возможности и позволяло оттачивать свои навыки. В книге «Программист-прагматик» даже есть раздел «Путь от подмастерья к мастеру» [50].

Если это утверждение верно, мы должны соответствующим образом структурировать нашу отрасль. У нас должны быть ученики, работающие вместе с мастерами. Мы могли бы даже организовать гильдии.

Так ведь?

Программирование как ремесло — еще одна метафора. Это похоже на ослепляющий свет истины, но он может таить в себе тени скрытого подтекста. Как говорится, чем ярче свет, тем темнее кажутся тени (рис. 1.1).

---

[1] Если вам любопытно, у меня есть высшее образование в сфере экономики, но, кроме как для работы в министерстве экономики Дании, оно мне больше не пригодилось.

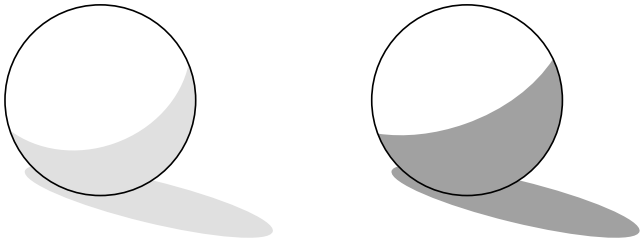


Рис. 1.1. Чем ярче освещена фигура, тем темнее отбрасываемая ею тень

Кажется, на рисунке все еще чего-то не хватает.

1.3.2. Эвристика

Годы, когда я занимался разработкой ПО, в некотором смысле были периодом полнейшего разочарования. Я рассматривал ремесло исключительно с точки зрения накопления опыта. Мне казалось, что нет никаких методологий и правил, что все зависит только от обстоятельств. Я думал, что не существует правильного или неправильного способа что-то делать.

Программирование было своего рода творчеством, что меня вполне устраивало. Мне всегда нравилось искусство. В молодости я даже хотел стать художником[1].

С этой точкой зрения проблема кажется невероятно сложной. Чтобы «взрастить» новых программистов, вам придется их обучать. А на то, чтобы преодолеть путь от подмастерья до мастера, у них может уйти несколько лет.

Еще одна проблема, связанная с отношением к программированию как к искусству или ремеслу, состоит в том, что эта аналогия тоже

[1] Я давно мечтал стать художником комиксов в европейской традиции. Позже, будучи подростком, я взял в руки гитару и решил стать рок-звездой. Но выяснилось, что, хотя мне нравилось и рисовать, и играть, я не был особо талантлив.

не соответствует действительности. Примерно в 2010 году я начал задумываться над тем [106], что все то время, когда программировал, я следовал эвристике — эмпирическим правилам и рекомендациям.

Поначалу я не придал этому большого значения. Но в дальнейшем, в процессе обучения других разработчиков, я часто определенным образом формулировал доводы для написания кода.

Я начал понимать, что ошибался в своих смелых утверждениях, и понял: рекомендации могут стать ключом к превращению программирования в техническую дисциплину.

1.3.3. Ранние представления о разработке ПО

О программной инженерии заговорили ближе к концу 1960-х годов[1]. Это было связано с *кризисом ПО* того времени — с появлением идеи о том, что программировать *сложно*.

На тот момент программисты действительно хорошо понимали, что делают. Многие выдающиеся деятели нашей отрасли трудились в то время: Эдсгер Дейкстра, Тони Хоар, Дональд Кнут, Алан Кей. Если бы вы тогда у них спросили, станет ли в 2020-х годах программирование отдельным предметом для изучения, вероятно, они бы ответили утвердительно.

Вы могли заметить, что я рассматриваю разработку ПО как амбициозную цель, а не как повседневную рутину. Вполне возможно, что в мире есть центры реальной разработки программного обеспечения[2], но, по моему опыту, бóльшая часть разработки программного обеспечения ведется в иной манере.

Я не одинок, предполагая, что разработка ПО — это наше будущее.

---

[1] Термин может быть старше. Я не могу ничего об этом рассказать, так как тогда я еще не родился. Но тот факт, что две конференции НАТО, в 1968 и 1969 годах, популяризировали термин «программная инженерия», не вызывает сомнений [4].

[2] НАСА выглядит достаточно близким к тому, чтобы быть одним из таких центров.

Адам Барр сказал следующее:

> *«Если вы похожи на меня, то вы мечтаете о том дне, когда разработка программного обеспечения будет изучаться вдумчиво и методично, а руководство, предоставленное программистам, будет основываться на экспериментальных результатах, а не на зыбучих песках индивидуального опыта» [4].*

Адам Барр объясняет, что программная инженерия стремительно развивалась, но затем появилось нечто, что помешало ей, — персональные компьютеры. Благодаря их развитию стали появляться разработчики, которые научились программировать самостоятельно. Поскольку они сами могли разбираться с компьютерами, они оставались в неведении относительно уже существовавших знаний.

Такая ситуация сохраняется и по сей день. Алан Кей описывает *компьютерную поп-культуру* так:

> *«Поп-культура презирает историю. Она дает ощущение самобытности и вовлеченности, причастности. Это не имеет ничего общего с сотрудничеством, прошлым или будущим, — это жизнь в настоящем. Я думаю, то же относится и к большинству людей, которые пишут код за деньги. Они понятия не имеют, откуда взялась их культура» [52].*

Возможно, мы потеряли полвека, добившись незначительного развития программной инженерии, но я думаю, что мы могли достичь прогресса в других направлениях.

1.3.4. Становление и развитие программной инженерии

Что делает инженер? Проектирует и контролирует каждый этап строительных работ, начиная от больших сооружений, таких как мосты (рис. 1.2), туннели, небоскребы и электростанции, до крошечных объектов, таких как микропроцессоры[1]. Инженер помогает создавать физические объекты.

[1] У меня когда-то был друг, инженер-химик по образованию. После окончания университета он стал пивоваром в Carlsberg. Так что инженеры могут даже варить пиво.

Рис. 1.2. Мост королевы Александрины (Queen Alexandrine Bridge) — арочный мост через пролив Ульвсунн между островами Зеландия и Мён в Дании. Был открыт в 1943 году

Но программисты так не делают, ведь ПО неосязаемо. Как сказал Джек Ривз [87], поскольку физического объекта нет, проектирование не будет ничего вам стоить. Разработка программного обеспечения — это прежде всего проектная деятельность. Написание кода больше похоже на построение плана инженером, а не на работу строителей на объекте.

Настоящие инженеры следуют методологиям, которые, как правило, приводят к успешным результатам. Мы, программисты, тоже хотим так делать. Но мы должны быть предельно внимательны и выполнять только те действия, которые будут целесообразны в нашем контексте. Когда проектируется физический объект, реальная конструкция стоит дорого. Вы не можете просто взять и попробовать построить мост, проверить его в течение какого-то времени, а потом решить, что он не особо хорош, разрушить его и начать все сначала. Поскольку на реальное строительство уходит много средств, инженеры изначально

занимаются расчетами и моделированием. Для расчета прочности моста нужно меньше времени и материалов, чем для его строительства.

Есть целая инженерная дисциплина, связанная с логистикой. Люди занимаются тщательным планированием — самым безопасным и наименее затратным способом создания физических объектов.

Это часть инженерии, которую нам *не нужно* копировать.

Но есть и много других приемов, которые могут нас вдохновить. Инженеры способны и на творческую работу, но она, как правило, четко структурирована. За одними конкретными действиями должны следовать другие. Специалисты контролируют и утверждают работу друг друга, следуют контрольному списку (чек-листу) [40].

Вы тоже можете так сделать.

Я считаю, что изучение эвристики — весьма полезное и занимательное занятие, но в то же время и очень сложное. Адам Барр называет это *зыбучими песками индивидуального опыта*.

Я считаю, что это отражает текущее состояние нашей отрасли. Любой, кто полагает, что у нас есть четкие научные доказательства, должен прочесть The Leprechauns of Software Engineering («Лепреконы программной инженерии») [13].

1.4. ЗАКЛЮЧЕНИЕ

Думая об истории разработки ПО, вы, вероятно, представляете себе успешное развитие отрасли и большое количество достижений. Но многие из этих достижений связаны с аппаратным, а не с программным обеспечением. И все-таки в разработке ПО за последние 50 лет мы добились существенного прогресса.

Сегодня у нас есть гораздо более продвинутые языки программирования, чем полвека назад, доступ к интернету (включая сервисы взаимопомощи наподобие Stack Overflow), объектно-ориентированное и функциональное программирование, автоматизированные среды тестирования, Git, интегрированные среды разработки и пр.

С другой стороны, мы все еще боремся с *программным кризисом*. Хотя можно ли назвать кризисом то, что длится уже полвека?

Несмотря на серьезные усилия, индустрия разработки ПО все еще не похожа на инженерную дисциплину. Между инженерией и программированием есть некоторые фундаментальные различия, и пока мы этого не поймем, мы не сможем добиться прогресса.

Хорошая новость в том, что программисты могут многое почерпнуть у инженеров: образ мыслей и набор процессов, которым можно следовать.

Как отметил научный фантаст Уильям Гибсон, будущее уже наступило, просто оно еще неравномерно распределено[1].

Как говорится в книге «Ускоряйся!», одни организации сегодня используют передовые методики, а другие отстают [29]. Будущее действительно распределено неравномерно. Хорошая новость в том, что прогрессивные возможности — в свободном доступе и, как вы будете их использовать, зависит только от вас.

В главе 2 вы познакомитесь с конкретными методами, которые сможете применить на практике.

---

[1] Это довольно расплывчатая цитата. Идея и формулировка в целом принадлежат У. Гибсону, но неясно, когда именно он впервые заявил об этом [76].

2 ЧЕК-ЛИСТЫ: ИСТОРИЯ, ВИДЫ, ПРЕИМУЩЕСТВА

Как программисту стать инженером-программистом? Я не утверждаю, что в книге есть однозначный ответ на этот вопрос, но надеюсь, что, прочитав ее, вы сможете выбрать верный для себя путь.

Что касается разработки ПО, то я думаю, что многое нам еще не дано понять. С другой стороны, мы не можем ждать, пока во всем разберемся. Мы извлекаем уроки и учимся на собственном опыте. Действия и методологии из этой книги уже давно придуманы великими людьми[1]. Эти практики до сих пор актуальны для меня и многих специалистов, которых я обучал. Надеюсь, что они будут полезными для вас или вдохновят вас на разработку своих, более совершенных методов.

2.1. КАК НИЧЕГО НЕ ЗАБЫТЬ

Основная проблема разработки ПО в том, что происходит огромное количество процессов, а наш мозг не способен решать несколько задач одновременно.

[1] Достойных людей слишком много, поэтому я не буду их всех здесь перечислять, но вы можете обратиться к библиографии. Я сделал все возможное, чтобы отдать должное этим людям за их вклад, и приношу свои извинения, если кого-то забыл.

Еще мы склонны игнорировать дела, которые не кажутся нам важными в данный момент.

Проблема не в том, что вы не знаете, как делать, а в том, что вы забываете это сделать, хотя знаете, что должны.

Эта проблема касается не только программирования. От этого страдают и, например, пилоты. Последние придумали простое решение проблемы: *чек-листы*.

Я понимаю, что это звучит невероятно скучно, но советую вам ознакомиться с историей происхождения чек-листа. Согласно Атулу Гаванде [40], идея чек-листов появилась в 1935 году, когда во время испытаний разбился прототип бомбардировщика B-17, что едва не привело к закрытию проекта. По сравнению с предыдущими самолетами B-17 оказался слишком сложным и дорогим в производстве. Управлять бомбардировщиком было непросто, что привело к трагедии, в результате которой погибли два члена экипажа, включая пилота.

Расследование авиакатастрофы показало, что причиной крушения стала ошибка пилота. Учитывая, что пилот был одним из самых опытных летчиков-испытателей армейской авиации, вряд ли это можно было бы списать на непрофессионализм. Как позже написали в одной из газет, самолет был слишком сложен, чтобы на нем мог летать один человек [40].

В результате группой летчиков-испытателей было придумано простое решение: создать чек-лист *основных* операций выполнения этапов взлета и посадки.

Простые чек-листы значительно расширяют возможности квалифицированных специалистов, таких как пилоты. При решении сложной задачи вы неизбежно забываете учитывать одно или несколько действий. Контрольный список поможет вам сосредоточиться на сложных частях вашей задачи, отвлекая внимание от разных мелочей. Вам не нужно будет прилагать усилия для выполнения всех простых действий — вы просто будете следовать *пунктам*.

Важно понимать, что чек-листы должны помогать, поддерживать и освобождать специалиста. Они не предназначены для мониторинга

или аудита. Сила таких списков в том, что работа с ними всегда выполняется качественно, единообразно и результативно. Хороший чек-лист поможет добиться необходимого результата и избежать ошибок. Это могут быть просто списки на плакате, в буфере обмена, в скоросшивателе и т. д.

Чек-листы не должны ограничивать вас в действиях. Они предназначены для улучшения организации процесса и повышения качества достигаемого результата.

Американский хирург и общественный деятель Атул Гаванде разработал чек-лист для врачей и медперсонала, который напоминает мыть руки перед операцией, проверять, давно ли введен антибиотик, и проводить рабочие совещания. Результаты оказались впечатляющими [40].

Если пилоты и хирурги смогли использовать чек-листы, то сможете и вы. Суть в том, чтобы *улучшить результат, не улучшая навыки*.

Далее я периодически буду приводить примеры чек-листов. Это не единственный «инженерный подход», который вы изучите, но он самый простой. Пусть это будет хорошим началом!

Контрольный список просто помогает вам не забыть о чем-то. Он не должен ограничивать вас — он существует, чтобы помочь вам не забывать выполнять тривиальные, но важные действия, такие как мытье рук перед операцией.

2.2. ЧЕК-ЛИСТ ДЛЯ НОВОЙ КОДОВОЙ БАЗЫ

Чек-листы из этой книги основаны на моем подходе к программированию и носят рекомендательный характер. Ваши ситуации могут отличаться, поэтому мои рекомендации могут подойти вам только частично. Так же как чек-лист по проверке взлета Airbus A380 отличается от чек-листа для B-17.

Так что просто ознакомьтесь с предложенными примерами и используйте наиболее подходящие для вашей конкретной ситуации либо создавайте свои наподобие.

Вот чек-лист для запуска новой кодовой базы.

1. Использовать Git.
2. Автоматизировать сборку.
3. Включить все сообщения об ошибках.

Я преднамеренно создал такой короткий список. Чек-лист — это не сложная блок-схема с подробными инструкциями. Это простой перечень, который вы можете охватить за несколько минут.

Чек-листы бывают двух видов: *«прочитал — сделал»* и *«сделал — отметил»* [40]. Цель первого — читать каждый пункт и выполнять действия строго друг за другом. Во втором вы выполняете все действия, а затем проверяете и подтверждаете завершенные отметкой.

Я намеренно оставил приведенный выше список расплывчатым и абстрактным, но, так как в нем использовано повелительное наклонение, он будет считаться чек-листом *«прочитал — сделал»*. Но вы можете легко сделать из него чек-лист *«сделал — отметил»,* и я настоятельно советую, чтобы его просмотрел другой человек. Именно так делают пилоты: один читает чек-лист, а другой подтверждает. В одиночку очень легко пропустить важный пункт, но товарищ всегда может вас подстраховать.

Как именно *использовать Git, автоматизировать сборку* и *включать все сообщения об ошибках,* зависит от вас. Далее мы разберем пример создания более конкретного чек-листа на основе списка, представленного выше.

2.2.1. Использовать Git

Git — стандартная система контроля версий для проектов с открытым исходным кодом, поэтому я рекомендую использовать именно его[1].

[1] Несмотря на то что Git превосходит большинство альтернативных систем, у него есть и ряд недостатков. Самый серьезный из них — это сложный и непоследовательный интерфейс командной строки. Если в будущем появится улучшенная распределенная система управления версиями, то лучше будет выбрать ее. Но на момент написания моей книги такой альтернативы не было.

По сравнению с централизованными системами управления версиями, такими как CVS или Subversion, распределенная система (Git) дает вам огромное преимущество. Но Git может быть полезна, только если вы знаете, как ее использовать.

Это не самая удобная система в мире, но вы — программист. Вам удалось выучить как минимум один язык программирования, и на фоне этого опыта изучить основы Git будет для вас проще простого. Но обязательно потратьте на это время. Не на изучение графического пользовательского интерфейса, а на изучение основного функционала Git, того, как система работает на самом деле.

Git дает возможность смело *экспериментировать* с кодом. Вы можете написать код и, если он не сработает, просто отменить изменения. Именно способность работать в качестве системы контроля версий на *вашем жестком диске* ставит Git выше централизованных систем.

У Git еще есть несколько графических пользовательских интерфейсов (GUI), но в этой книге я остановлюсь на командной строке. Это не только основа Git, но и способ, которым я обычно предпочитаю с ним работать. И так как моя ОС — Windows, я работаю в командной строке Git Bash.

Первое, что нужно сделать в новой кодовой базе, — инициализировать локальный репозиторий Git[1]. Откройте окно командной строки в папке, в которую хотите поместить код. Сейчас вам не нужно беспокоиться об онлайн-сервисах Git вроде GitHub. Вы всегда можете подключить репозиторий позже. Затем пропишите следующую команду[2]:

```
$ git init
```

[1] Я бы придерживался этого правила для любой кодовой базы, которая должна прожить больше недели. Иногда я не беспокоюсь об инициализации репозитория Git для действительно эфемерного кода, но мой порог для создания репозитория Git довольно низок. Вы всегда можете все отменить, удалив папку .git.

[2] Символ $ указывает на приглашение командной строки, его не нужно писать перед командой. Я буду использовать его в примерах в этой книге.

Вот и все. Вы можете последовать совету моего друга Энрико Компидоглио [17] и добавить пустой коммит:

```
$ git commit --allow-empty -m "Initial commit"
```

Обычно я тоже так делаю, поскольку это позволяет мне переписать историю моего репозитория, прежде чем я опубликую его в онлайн-сервисе Git. Но так делать вам совсем не обязательно.

2.2.2. Автоматизировать сборку

Когда у вас почти нет кода, легко автоматизировать компиляцию, тестирование и развертывание. Попытка же приспособить непрерывную доставку (continuous delivery, CD, CDE) [49] к существующей кодовой базе может показаться сложной задачей. Поэтому мы с вами займемся этим прямо сейчас.

Сейчас у вас нет кода, есть только репозиторий Git. Вам нужно сделать небольшое приложение, чтобы было что компилировать. Создайте минимальный объем кода, который сможете развернуть. Эта концепция похожа на «ходячий скелет» [36], но на один шаг раньше в процессе разработки (рис. 2.1).

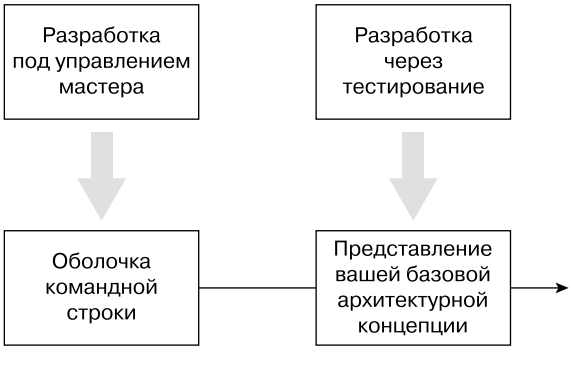


Рис. 2.1. Чтобы создать оболочку приложения, закоммитить ее и развернуть, используйте мастер или программу формирования шаблонов. Далее используйте автоматизированный тест для создания «ходячего скелета» [36], с которым вы будете совершать эти же действия

«Ходячий скелет» (Walking Skeleton) — это реализация наименьшей возможной функциональности, которую вы можете автоматически создавать, развертывать и тестировать от начала до конца [36]. Вы можете сделать это следующим шагом, но я думаю, что лучше сначала установить конвейер развертывания [49].

Распространенные проблемы, связанные с созданием конвейера развертывания

Что делать, если вы не можете настроить конвейер развертывания? Возможно, у вас нет сервера непрерывной интеграции. Если это так, то его нужно приобрести. Вовсе не обязательно покупать реальный сервер — сейчас есть множество облачных сервисов непрерывной доставки.

Возможно, у вас нет продакшен-среды (production environment, PROD)? Решить эту проблему можно, настроив конвейер развертывания так, чтобы вы могли выпускать его в тестовой среде. Желательно тот, который максимально похож на продакшен-среду. Даже если вы не можете получить аппаратное обеспечение, похожее на продакшен-среду, попробуйте смоделировать топологию сети продакшен-системы. Вы можете использовать машины поменьше, виртуальные машины или контейнеры.

Большинство методов, которые я предлагаю здесь, бесплатны. Самые большие суммы обычно идут на оплату работы серверов, ПО или облачных сервисов. Эти затраты обычно составляют лишь малую часть от зарплаты программиста, поэтому, по сравнению с общей стоимостью разработки ПО, эти деньги будут потрачены не зря.

Перед настройкой конвейера развертывания важно убедиться, что вы можете легко скомпилировать код и запустить тесты для разработчиков. В этом случае вам понадобится код.

Эта книга построена вокруг одного основного примера: разработки простой системы онлайн-бронирования ресторанов на языке C#. И прямо сейчас нам нужен веб-сервис, который будет обрабатывать HTTP-запросы.

Самый простой способ начать двигаться в этом направлении — создать веб-проект на ASP.NET Core. Для этого я буду использовать среду разработки Visual Studio[1]. Хотя мне нравится использовать интерфейсы командной строки для часто выполняемых взаимодействий, я считаю полезным и руководство по IDE, которое помогает мне в решении моих задач. По желанию вы можете использовать вместо этого инструмент командной строки, но результат должен быть тем же: несколько файлов и работающий веб-сайт. В листингах 2.1 и 2.2 показаны[2] примеры файлов, созданных Visual Studio.

После запуска сайта вы увидите, что он содержит следующий текстовый файл:

```
Hello World!
```

Сейчас нам этого достаточно, поэтому сделайте коммит своего кода в Git.

Листинг 2.1. Точка входа веб-сервиса ASP.NET Core по умолчанию, созданная Visual Studio (Restaurant/f729ed9/Restaurant.RestApi/Program.cs)

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Цель этого шага — автоматизация сборки. Вы можете открыть свою IDE и использовать ее для компиляции кода, но это нельзя будет

---

[1] Я не буду вдаваться в подробности процесса, так как эта информация, скорее всего, уже устареет на момент выхода книги. Но это простой метод, включающий всего один-два шага.

[2] C# — относительно многословный язык, поэтому я буду показывать только основные моменты. Я не использовал директивы и объявления пространств имен.

автоматизировать. Создайте файл сценария, выполняющего сборку, и тоже отправьте коммит в Git. На начальном этапе все так же просто, как в листинге 2.3.

Несмотря на то что я работаю в Windows, я все время использую командную строку в Bash, поэтому я определил сценарий оболочки. Вместо этого вы можете создать файл `.bat` или скрипт PowerShell[1]. Важно то, что прямо сейчас должна быть вызвана команда `dotnet build`. Обратите внимание: я настраиваю релизную сборку. Автоматизированная сборка должна отражать то, что в конечном итоге будет запущено в производство.

По мере добавления шагов сборки их следует добавлять и в скрипт. Он должен стать простым инструментом, который разработчики могут запускать на своей машине. Если скрипт сборки выполняется на компьютере разработчика, можно отправлять изменения на сервер непрерывной интеграции.

Листинг 2.2. Файл по умолчанию Startup, сгенерированный Visual Studio. Чтобы строки с комментариями выглядели более эстетично и компактно, я их отредактировал (Restaurant/f729ed9/Restaurant.RestApi/Startup.cs)

```
public class Startup
{
    // This method gets called by the runtime. Use this method to add
    // services to the container.
    // For more information on how to configure your application,
    // visit https://go.microsoft.com/fwlink/?LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
    }

    // This method gets called by the runtime. Use this method
    // to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
```

[1] Если вам нужно сделать что-то более сложное, например собрать документацию, скомпилировать многоразовые пакеты для менеджеров пакетов и т. д., можете рассмотреть полноценный инструмент сборки. Но всегда легче двигаться от простого к сложному.

```
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", async context =>
            {
                await context.Response.WriteAsync("Hello World!");
            });
        });
    }
}
```

Листинг 2.3. Скрипт сборки (Restaurant/f729ed9/build.sh)

```
#!/usr/bin/env bash
dotnet build --configuration Release
```

Следующий шаг — создание конвейера развертывания. При добавлении новых коммитов в *мастер-ветку* запустится процесс, который (при успешном выполнении) будет развертывать изменения в вашей продакшен-среде или как минимум подготовит все, чтобы этот процесс можно было выполнить вручную.

В этой книге я не буду описывать все детали. Они зависят от сервера или сервиса непрерывной интеграции, который вы используете, и от его версии — все это со временем может измениться. Я мог бы показать вам, как включить это в Azure DevOps Services, Jenkins, TeamCity и т. д., но тогда эта книга стала бы книгой о конкретной технологии.

2.2.3. Включить все сообщения об ошибках

Однажды, когда я обучал другого программиста добавлять модульные тесты в существующую кодовую базу, мы столкнулись с проблемой. Код скомпилировался, но работал не должным образом. Он лихорадочно перемещался по своей базе, хаотично меняя разные строчки кода. Тогда я поинтересовался у своего коллеги: «Можем ли мы посмотреть, не было ли каких-либо предупреждений компилятора?»

Я понимал, в чем проблема, но предоставил своему ученику возможность найти решение самостоятельно — так обучение будет более продуктивным.

«Это бесполезно, — ответил он. — В этой кодовой базе сотни предупреждений компилятора».

Это было правдой, но я настоял, чтобы мы просмотрели список, и быстро нашел предупреждение, которое, как я знал, там будет. Оно верно обозначило проблему.

Старайтесь использовать предупреждения компилятора и другие автоматические инструменты — они способны находить разные проблемы с кодом. Помимо применения Git, это один из самых простых способов, которые вы можете выбрать.

Меня беспокоит, что так мало людей используют легкодоступные инструменты.

Большинство языков и сред программирования сопровождаются разными инструментами проверки вашего кода, такими как компиляторы, линтеры, инструменты анализа кода, средства защиты стиля и форматирования. Старайтесь использовать как можно больше из них — они очень полезны и редко ошибаются.

Все примеры в этой книге написаны на языке C#. Это компилируемый язык, и компиляторы обычно всегда выдают предупреждения при обнаружении компилируемого, но, скорее всего, неправильного кода. Как правило, эти предупреждения верны, поэтому обращайте на них внимание.

Как говорится, бывает сложно обнаружить новое предупреждение компилятора, если у вас уже есть 124 предыдущих. Поэтому старайтесь не копить предупреждения — их не должно быть вообще.

По сути, вы должны относиться к предупреждениям как к ошибкам.

Все компилируемые языки, с которыми я работал, могут преобразовывать предупреждения компилятора в ошибки, чтобы избежать их дальнейшего накопления.

Быстро устранить сотню существующих предупреждений может быть сложно. Гораздо проще реагировать на предупреждение в момент его появления. Поэтому первое, что вам нужно сделать в новой кодовой базе, — включить параметр «Представлять предупреждения как ошибки». Это поможет вам избежать накопления любых предупреждений компилятора.

Когда я делаю это в кодовой базе из подраздела 2.2.2, код все равно компилируется. Тот небольшой код, который Visual Studio сгенерировал для меня, к счастью, не выдает никаких предупреждений[1].

У многих языков и сред программирования есть дополнительные автоматизированные инструменты, которые вы тоже можете использовать. Например — линтер, который предупредит вас, если фрагмент кода окажется подозрительным, или даже проверит текст на наличие орфографических ошибок. Существуют линтеры для разных языков, например для таких, как JavaScript и Haskell.

В C# есть аналогичный набор инструментов, которые называются *анализаторами*. В отличие от превращения предупреждений в ошибки, которые включаются простой постановкой галочки, добавление анализаторов требует немного больше усилий. Но, начиная с последней версии Visual Studio, это стало сделать гораздо проще[2].

На сегодняшний день анализаторы отражают накопленные за десятилетия знания о том, как писать код .NET. Изначально они представляли собой встроенный инструмент UrtCop, который использовался во время ранней разработки самой платформы .NET, поэтому он предшествует .NET 1.0. Позже инструмент был переименован в FxCop [23] и недавно был повторно реализован в цепочке инструментов компилятора Roslyn.

---

[1] В Visual Studio параметры «Предоставлять предупреждения как ошибки» связаны с конфигурацией сборки. Вы обязательно должны рассматривать предупреждения как ошибки в режиме Release, но я делаю это и в режиме Debug. Если хотите изменить этот параметр для обеих конфигураций, не забудьте сделать это дважды. Возможно, вам стоит внести этот пункт в свой чек-лист.

[2] Я не описываю фактические шаги. Детальное описание может устареть еще до момента выхода книги.

Это расширяемая структура, содержащая множество рекомендаций и правил. Анализатор определяет нарушения соглашений об именах, потенциальные проблемы безопасности, неправильное использование API известных библиотек, проблемы с производительностью и многое другое.

При активации в примере кода из листингов 2.1 и 2.2 набор правил по умолчанию выдает не менее семи предупреждений! Поскольку теперь компилятор обрабатывает предупреждения как ошибки, код больше не компилируется. Может показаться, что это мешает выполнению работы, но единственное, что должно быть таким образом разрушено, — это иллюзия того, что код можно поддерживать без тщательного обдумывания.

Легче устранить семь предупреждений сегодня, чем сотни в будущем. Позже вы поймете, что большинство исправлений связано с удалением кода. Вам нужно внести только одно изменение в класс `Program`. Результат можно увидеть в листинге 2.4. Можете заметить изменения?

Листинг 2.4. Точка входа веб-сервиса ASP.NET Core после устранения предупреждений анализатора (Restaurant/caafdf1/Restaurant.RestApi/Program.cs)

```
public static class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Изменение в том, что теперь класс `Program` помечен ключевым словом `static`. У класса нет причин поддерживать создание экземпляров, если у него есть только общие члены. Это правило анализа кода. Здесь

оно вряд ли имеет большое значение, но, с другой стороны, исправить это так же просто, как добавить одно ключевое слово в объявление класса. Так почему бы это не сделать? В других случаях это правило может помочь вам упростить вашу кодовую базу.

Большинство изменений, которые мне пришлось внести, затрагивают класс `Startup`. Они содержат удаление кода, поэтому я думаю, что результат будет значительно лучше (листинг 2.5).

Листинг 2.5. Файл Startup после устранения предупреждений анализатора. Сравните с листингом 2.2 (Restaurant/caafdf1/Restaurant.RestApi/Startup.cs)

```
public sealed class Startup
{
    // This method gets called by the runtime. Use this method
    // to configure the HTTP request pipeline.
    public static void Configure(
        IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", async context =>
            {
                await context.Response.WriteAsync("Hello World!")
                    .ConfigureAwait(false);
            });
        });
    }
}
```

Что здесь изменилось? Я удалил метод `ConfigureServices`, так как в нем не было никакого смысла, запечатал класс с помощью модификатора `sealed` и добавил вызов `ConfigureAwait`.

К каждому правилу анализа кода прилагается онлайн-документация. Вы можете прочитать о мотивах для правила и о том, как реагировать на предупреждения.

Ссылочные типы, допускающие значение NULL

В C# 8 есть необязательная функция, известная как *ссылочные типы, допускающие значение NULL*[1]. Она позволяет использовать статическую систему типов, чтобы объявить, может ли объект принимать значение `null`. Когда эта функция включена, предполагается, что объекты *не могут* принимать значение `null`, то есть они не могут быть нулевыми.

Если вам все-таки нужно, чтобы объект *принимал* значение `null`, вы можете дополнить объявление типа символом `?` (знак вопроса), например: `IApplicationBuilder? app`.

Способность отличать объекты, которые не должны принимать значение `null`, от объектов, которые могут принимать его, помогает уменьшить объем защитного кода. Эта функция поможет уменьшить количество ошибок кода во время его выполнения. Добавляйте ее в новую кодовую базу, чтобы исключить большое количество ошибок компилятора.

Когда я включаю эту функцию для примера кода в этой главе, код все еще компилируется.

Статический анализ кода похож на автоматизированный код-ревью. Когда со мной связывается организация разработчиков, которая хочет, чтобы я провел код-ревью их проекта на C#, я сначала прошу их запустить анализаторы. Так можно сэкономить время и деньги.

[1] То, как Microsoft называет концепции и функции, может сбивать с толку. Как и в других основных языках на основе C, ссылочные типы всегда допускали, что объекты могут принимать нулевое значение. На деле эта функция должна называться необнуляемыми ссылочными типами.

Как правило, с жалобами ко мне после этого больше не обращаются[1]. Запуская такие анализаторы в существующей кодовой базе, вы можете легко получить тысячи предупреждений, на проверку которых придется потратить много времени. Чтобы предотвратить это, начните использовать эти инструменты прямо сейчас.

Вопреки предупреждениям компилятора, такие инструменты статического анализа кода, как линтеры или анализаторы .NET Roslyn, часто дают ложные срабатывания[2]. Автоматизированные инструменты обычно предлагают разные варианты подавления ложных срабатываний, так что вряд ли стоит их игнорировать.

Предупреждения компилятора, линтера и статического анализа следует рассматривать как ошибки. Поначалу это будет раздражать, но поможет вам значительно улучшить код и поспособствует вашему профессиональному росту.

Такой подход можно считать *инженерным*, не так ли? Это лишь малая часть из того, что вы можете сделать, но для начала уже хорошо. Инженерия в широком смысле — это применение всех эвристических и детерминированных механизмов для повышения своих шансов на успех. Эти инструменты похожи на автоматизированные чек-листы: каждый раз, когда вы запускаете их, они проверяют тысячи потенциальных проблем.

Некоторые из них существуют уже давно, но, по моему опыту, мало кто ими пользуется. «Будущее уже наступило. Просто оно еще неравномерно распределено». Включите элементы управления. *Улучшайте свой результат, не задействуя навыки.*

В самом начале проще всего рассматривать предупреждения как ошибки — их можно проверить, не прибегая ни к каким инструментам.

---

[1] Я ужасный бизнесмен... или нет?

[2] Понимаю, что это может сбить с толку, но положительное значение означает предупреждение, то есть код выглядит неправильно. Звучит не очень положительно, но в терминологии бинарной классификации положительный результат указывает на наличие сигнала, а отрицательный — на его отсутствие. Это также используется в тестировании программного обеспечения и медицине. Только подумайте, что означает положительный результат теста на COVID-19!

2.3. ДОБАВЛЕНИЕ ПРОВЕРОК В СУЩЕСТВУЮЩИЕ КОДОВЫЕ БАЗЫ

В действительности редко возникает возможность написать свою новую кодовую базу самостоятельно. Бóльшая часть профессиональной разработки ПО предполагает работу с уже существующим кодом. В новой кодовой базе обработка предупреждений как ошибок менее обязательна, а в существующей это возможно.

2.3.1. Постепенное улучшение

Очень важно постепенно добавлять в код дополнительные проверки или условия. Большинство существующих кодовых баз содержат несколько библиотек[1] (рис. 2.2). Включите дополнительные проверки по одной библиотеке за раз.

Рис. 2.2. Кодовая база, состоящая из пакетов.
Здесь это HTTP API, доменная модель и доступ к данным

Часто вы можете включить один тип предупреждений за раз. В существующей кодовой базе у вас уже могут быть сотни предупреждений

[1] Библиотеки известны и как пакеты. Разработчики Visual Studio часто называют библиотеки проектами внутри решения.

компилятора. Извлеките список и сгруппируйте его по типам. Затем выберите конкретный тип предупреждения, который может иметь около дюжины экземпляров. Исправьте их, пока они еще являются предупреждениями компилятора, чтобы можно было продолжать работу с кодом. Проверяйте свои изменения в Git каждый раз, когда что-то улучшаете. Объедините весь улучшенный код с *мастер-веткой*.

Как только вы устраните последнее предупреждение данного типа (в этой части кодовой базы), превратите эти предупреждения в ошибки. Затем перейдите к другому типу предупреждений или обратитесь к тому же в другой части кодовой базы.

То же можно сделать с линтерами и анализаторами. Например, с помощью анализаторов .NET вы можете указать, какие правила следует включить. Обращайтесь к одному правилу за раз и, как только вы устраните все предупреждения, создаваемые этим правилом, включите его, чтобы оно предотвращало все повторения в будущем.

Точно так же можно постепенно включать функцию ссылочных типов C#, *допускающих значение NULL*.

Ключевой момент — правило бойскаута [61]: помещать код в лучшем состоянии, чем вы его нашли.

2.3.2. «Взломайте» свою организацию

Когда я выступаю на конференциях, ко мне часто подходят люди. Обычно они воодушевлены, но чувствуют, что их руководитель не позволяет им сосредоточиваться на *внутреннем качестве*.

Преимущество обработки предупреждений как ошибок в том, что вы переходите на новый уровень качества. При рассмотрении предупреждений как ошибок и включении статического анализа кода вы теряете часть контроля. Это звучит неприятно, но может стать вашим преимуществом.

Когда вы сталкиваетесь с необходимостью «просто сделать», так как «у нас нет времени делать это как положено», представьте, что вы отвечаете: «Извините, но, если я это сделаю, код не скомпилируется».

Такой ответ может пресечь попытки заинтересованных сторон настаивать на игнорировании требований и правил. Это не совсем тот случай, когда вы не можете обойти ни одну из этих автоматических проверок, но вы не обязаны всем об этом говорить. Хитрость в том, что то, что раньше вы делали вручную, сейчас можно сделать с помощью специальных программ.

Уместно ли это? Решайте сами. Как профессиональный разработчик, вы являетесь техническим экспертом. Это ваша работа — принимать технические решения. Вы можете сообщить обо всех деталях своему руководству, но для простых менеджеров бóльшая часть информации будет бессмысленной. Техническая экспертиза включает в себя предоставление только той информации, которую заинтересованные стороны *могут легко понять и использовать*.

В здоровой организации лучшая стратегия — быть открытым и честным в том, что вы делаете. В нездоровой же, например в той, где плохо налажены процессы, принятие контрстратегии может быть более подходящим методом. Здесь вы как раз можете использовать автоматические проверки. Даже если это будет неприемлемо, вы всегда сможете утверждать, что конечная цель — это качественное ПО. Это должно быть выгодно для всей организации.

Используйте свои правила и принципы во благо организации, а не только для продвижения личных целей.

2.4. ЗАКЛЮЧЕНИЕ

Чек-листы значительно улучшают результаты вашей работы, не требуя совершенствования навыков [40]. Используйте их. Чек-листы помогают не забыть о принятии правильных решений, выполнять сложные задачи и не упускать важные детали. Они поддерживают вас, но не контролируют.

В этой главе приведен пример простого чек-листа, который можно использовать при запуске новой кодовой базы. Чек-лист может быть простым, но максимально эффективным.

Вы изучили принцип работы с Git. Это самый простой из трех пунктов в чек-листе, но эти небольшие усилия окупаются многократно.

Вы научились автоматизировать сборку. Это будет легко, если сделать это сразу. Создайте скрипт сборки и используйте его.

Наконец, вы разобрались, как преобразовать предупреждения компилятора в ошибки. Можете использовать и дополнительные автоматические проверки, такие как линтеры или статический анализ кода. Эти функции легко добавить, нет особых причин игнорировать их.

В оставшейся части книги вы изучите влияние этих ранних решений на кодовую базу по мере добавления новых функций.

Инженерия — это больше, чем следование чек-листам или автоматизация. Но если вы применяете эти технологии, то движетесь в верном направлении. Это небольшие улучшения, которые вы можете сделать уже сегодня.

3 ПРЕОДОЛЕНИЕ ТРУДНОСТЕЙ

Прислушайтесь к своей интуиции и попробуйте решить следующую простую задачу. Не пытайтесь прибегнуть к математике или вычислениям.

Бейсбольная бита и мяч стоят 1 доллар 10 центов. Бита стоит на 1 доллар дороже, чем мяч. Сколько стоит мяч?

Обратите внимание на свой ответ. Кажется, что это слишком простой вопрос для такой книги, поэтому вы начинаете искать здесь подвох.

Чуть позднее мы вернемся к этой задаче.

Читая эту главу, вы делаете шаг назад и пытаетесь ответить на фундаментальный вопрос: «Почему разрабатывать ПО так сложно?»

Ответ на него столь же фундаментален. Он связан с принципом работы человеческого мозга. Это главная тема всей книги. Прежде чем обсуждать, как писать код, который умещается в вашей голове, мы должны обсудить, что еще укладывается в ней.

В последующих главах мы всё будем рассматривать на примерах.

3.1. ЦЕЛЬ

Скорее всего, после прочтения первых двух глав вы будете озадачены. Возможно, вы думали, что программная инженерия — интеллектуально сложная и таинственная дисциплина. Так и есть, но нужно с чего-то начать. И почему бы не начать с чего-то простого? Каждый успешный человек начинал свой путь, делая небольшие шаги в выбранном направлении. Помните, что и восхождение в гору всегда начинается у ее подножия (рис. 3.1).

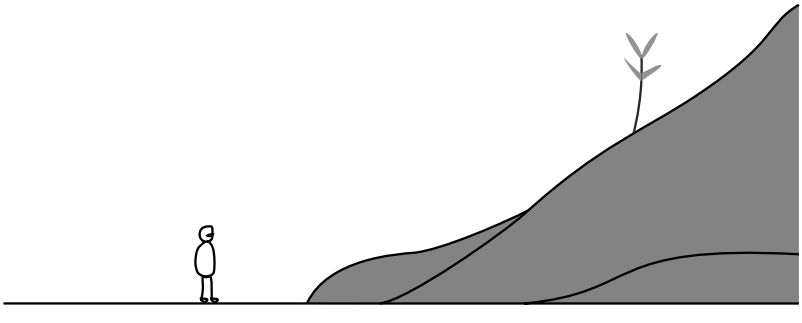


Рис. 3.1. Восхождение на гору всегда начинается у ее подножия

Прежде чем мы продолжим, думаю, нам следует сделать паузу и обсудить проблему, которую мы пытаемся решить. Что это за проблема?

Основная тема, которая поднимается в этой книге, связана с надежностью программного обеспечения, которое впоследствии должно стать важной и полезной составляющей вашей организации или вашего бизнеса.

3.1.1. Надежность

Организация создает ПО по разным причинам. Как правило, чтобы заработать или сэкономить. Иногда государство запускает программные проекты для развития цифровой инфраструктуры граждан. Само программное обеспечение не предполагает прямой прибыли или экономии, но перед ним всегда стоит задача, которую нужно выполнить.

Часто на разработку сложного ПО уходят месяцы или даже годы.

Многие программы могут существовать десятилетиями. В течение этого времени могут выпускаться обновления, внедряться новые функции, исправляться ошибки и т. д. Процесс разработки ПО — это постоянная, непрерывная работа.

Программное обеспечение предназначено для помощи в организации рабочих процессов компании. Обновляя функционал или исправляя ошибки, вы оказываете существенную поддержку вашей фирме. И хорошо, когда такое сопровождение ПО осуществляется постоянно, как сейчас, так и через полгода.

Разработка, обновление и сопровождение ПО требуют постоянных усилий по его *поддержанию*.

Как говорит Мартин Фаулер, не обращая внимания на внутреннее качество, вы очень быстро потеряете способность вносить улучшения в свою кодовую базу.

> *«Вот что происходит, когда качество оставляет желать лучшего. Сначала мы видим быстрый прогресс, но со временем добавлять новые функции становится все сложнее. Даже незначительные изменения требуют от программистов понимания больших объемов сложного для восприятия кода. После внесения изменений начинают возникать неожиданные проблемы, что приводит к длительному тестированию и появлению ошибок, которые нужно исправлять» [32].*

Это та ситуация, которую должна решать программная инженерия. Важно сделать процесс разработки ПО более регулярным, что

положительно повлияет на поддержку вашей организации на месяцы, годы и десятилетия.

> Разработка, тестирование и сопровождение программного обеспечения — это *регулярный* и непрерывный процесс. Качественное и надежное ПО — это залог успеха организации.

3.1.2. Ценность

Цель программного обеспечения — приносить *пользу* организации. С его помощью можно повысить ценность вашего бизнеса. Часто профессионалы в области разработки ПО думают, что если их код не представляет ценности, то его было бессмысленно создавать.

Определенный акцент на создании ценности может быть оправдан. Я знаком с некоторыми разработчиками, которые, если их предоставить самим себе, будут проводить часы, бездельничая с какой-нибудь умной структурой собственного изобретения.

Такие ситуации свойственны и для коммерческих компаний. Ричард П. Габриэль рассказал о взлете и падении Lucid [38] так: пока они возились с идеальной коммерческой реализацией языка Common Lisp, появился C++ и захватил рынок кросс-платформенных языков разработки ПО.

Сотрудники Lucid считали, что C++ хуже Common Lisp, но Габриэль в конце концов понял, почему клиенты выбрали именно его. Язык C++, возможно, был менее последовательным и более сложным, но он работал и был доступен клиентам. Это побудило Габриэля сформулировать афоризм: «Хуже значит лучше». Так компания Lucid ушла с рынка.

В правой части рис. 3.2 изображены специалисты, которые сосредотачиваются только на одной какой-то технологии, не обращая внимания на ее назначение.

Рис. 3.2. Одни разработчики никогда не обращают внимания на ценность кода, который пишут, а другие всегда оценивают свои прошлые результаты. Верное решение находится где-то посередине

Акцент на ценности будет возможной реакцией на аналогичный тип мышления. Важно узнать, служит ли код какой-то определенной цели. Термин *«ценность»* часто используется в качестве *условного обозначения*, несмотря на то что его нельзя измерить. Есть школа управления проектами, которая базируется [88] на следующих принципах.

1. Сформулировать гипотезу о влиянии планируемых изменений.
2. Внести изменения.
3. Измерить влияние и сравнить его с прогнозом.

Мое пособие вовсе не об управлении проектами, но такой подход кажется мне разумным, о чем я уже писал ранее [29].

Представление о том, что код должен быть полезным, к сожалению, ведет к логической ошибке, согласно которой код, не несущий ценности, запрещен. Идея «чем хуже, тем лучше» становится все ближе.

Это в корне неверно, так как не каждый код сможет дать *немедленно измеримое значение*. С другой стороны, можно измерить ценность его отсутствия. Простой пример — безопасность. Возможно, у вас не получится измерить ценность включения аутентификации в онлайн-систему, но вы точно сможете измерить ее отсутствие.

То же самое относится и к аргументу Фаулера о внутреннем качестве [32]. Отсутствие архитектуры можно измерить, но, когда появится такая возможность, будет уже слишком поздно. В своей жизни я уже наблюдал, как из-за низкого внутреннего качества разорялись целые компании.

На рис. 3.2 среднее место занимает надежность. Здесь не поощряются технологии ради технологий, но и рекомендуется не слишком фокусироваться на ценности.

Разработка ПО должна способствовать надежности. Следуя чек-листам, рассматривая предупреждения как ошибки и т. д., вы сможете предотвратить появление нежелательного кода [32]. Ни одна из методологий и эвристик в этой книге не сможет гарантировать идеального результата, но их знание поможет вам двигаться в верном направлении. Так или иначе вам придется использовать свой опыт и навыки. В конце концов, это и есть искусство разработки программного обеспечения.

3.2. ПОЧЕМУ ПРОГРАММИРОВАТЬ ТАК СЛОЖНО?

Почему программировать сложно? Причин много. Во-первых, как обсуждалось в разделе 1.1, мы проводим неверные аналогии, которые заводят нас в тупик. Но это не единственная причина.

Еще одна причина в том, что компьютер — это не мозг. Кстати, это еще одно неподходящее сравнение.

3.2.1. Аналогия с мозгом

Сходство компьютера и мозга кажется очевидным (рис. 3.3). Между ними определенно есть поверхностное сходство. Оба могут выполнять вычисления, вспоминать прошлые события, хранить и извлекать информацию.

Компьютер похож на мозг? Я думаю, что различий между ними больше, чем сходств. Компьютер не может делать интуитивные выводы, плохо интерпретирует изображение и звук[1]. В конце концов, у него нет внутренней мотивации.

[1] Искусственный интеллект в последние годы добился больших успехов, но проблемы, с которыми борются исследователи, все еще находятся на уровне, который может легко решить даже малыш. Покажите компьютеру детскую книжку с рисунками животных и спросите, что изображено на каждой картинке.

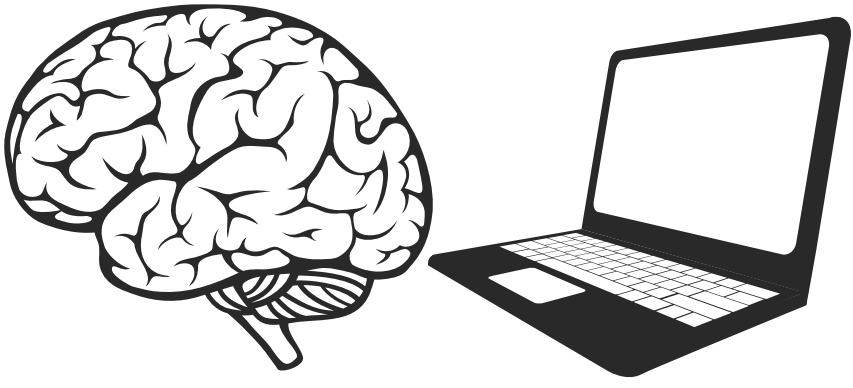


Рис. 3.3. Мозг похож на компьютер? Пусть вас не вводит в заблуждение очевидное сходство

Мозг похож на компьютер? По сравнению с компьютером наша способность к вычислениям невероятно медленная, а память слишком ненадежна: воспоминания могут исчезнуть или быть искажены, ими можно манипулировать [109], а вы даже не будете подозревать об этом. Например, вы уверены, что 20 лет назад были на вечеринке со своей лучшей подругой, тогда как она утверждает, что никогда там не была. Получается, что у кого-то из вас ложные воспоминания.

А что насчет *оперативной памяти*? Благодаря ей компьютер может отслеживать миллионы действий, тогда как кратковременная память человека может хранить от четырех до семи[1] единиц информации [80; 109].

Все вышеперечисленное играет в программировании важную роль. Даже скромная подпрограмма может легко создать десятки переменных и команд. Пытаясь понять, что делает исходный код, вы, по сути, запускаете в уме эмулятор языка программирования. И если происходит слишком много всего, вы никогда не сможете это проконтролировать.

[1] Возможно, вы тоже сталкивались с этим волшебным числом плюс-минус два, точное число не так уж и важно. Важно лишь то, что это число на несколько порядков меньше размера оперативной памяти компьютера.

Слишком много — это сколько?

В этой книге число *семь* используется как символ предела кратковременной памяти мозга. Иногда вы можете отслеживать и девять действий, но семь из них всегда будут под контролем.

3.2.2. Код больше читается, чем пишется

Постепенно мы приближаемся к фундаментальной проблеме программирования.

> Вы тратите больше времени на чтение кода, чем на его написание.

Вы пишете строку кода один раз, а читаете ее по нескольку раз [61]. С нетронутой кодовой базой редко приходится иметь дело, а прежде, чем приступить к работе с уже существующей, нужно в коде разобраться. Чтобы добавить новую функцию, вы читаете данный код, чтобы понять, какой можно использовать повторно, а куда лучше добавить новый. Чтобы исправить ошибку, важно сначала понять причину ее выведения. Обычно вы тратите бóльшую часть своего рабочего времени на чтение существующего кода.

> Оптимизируйте свой код для удобства чтения.

Вы постоянно слышите о новых языках программирования, библиотеках, платформах или функциях IDE, позволяющих быстрее создавать больше кода. Как показывает история компании Lucid, вряд ли это будет хорошей стратегией для устойчивого развития ПО. Быстрое создание большого количества кода скорее будет означать больше кода, который нужно будет прочесть, — чем больше вы пишете, тем больше вам приходится читать. Автоматическая генерация кода может только усугубить ситуацию.

Мартин Фаулер сказал о низком качестве кода следующее:

> *«Даже небольшие изменения требуют от программистов понимания больших объемов сложного для восприятия кода» [32].*

Трудный для понимания код замедляет работу, а каждая минута, которую вы тратите на то, чтобы этот код упростить, окупается десятикратно.

3.2.3. Удобочитаемость

Легко сказать, что вы должны отдавать предпочтение читабельному коду, а не тому, который легко написать, но что же такое этот читабельный код?

Бывало ли такое, что, читая код, вы спрашивали себя: «Кто написал эту ерунду?!» — а потом оказывалось, что именно вы и написали?[1]

Такое бывает со всеми: когда вы пишете код, вы понимаете, о чем пишете, но когда читаете его, все понимание куда-то исчезает.

В конце концов, важность кода неоспорима. Документация может быть устаревшей или ее может не быть вовсе. Автор кода может быть в отпуске или уволиться.

Вдобавок ко всему, мозг плохо работает при чтении и оценке формальных утверждений. И все-таки как вы ответили на вопрос о бейсбольной бите и мяче в начале этой главы?

Число, которое сразу же пришло вам в голову, — 10. Этот ответ дает большинство людей [51]. Но он неверный. Если мяч стоит 10 центов, то бита должна стоить 1 доллар 10 центов, а общая стоимость составит 1,20 доллара. Правильный ответ — 5 центов.

Дело в том, что мы постоянно ошибаемся. Особенно, когда решаем тривиальные математические задачи или читаем код.

---

[1] git blame — отличный инструмент для такого случая.

Чтобы написать удобочитаемый код, вам не следует доверять своей интуиции. Нужно что-то более эффективное: эвристика, чек-листы... программная инженерия. Мы будем возвращаться к этой теме на протяжении всей книги.

3.2.4. Интеллектуальный труд

Бывало ли, что вы едете на машине и через десять минут вдруг «просыпаетесь», в ужасе спрашивая себя: «Как я сюда попал?»

У меня однозначно было. Не то чтобы я в прямом смысле заснул за рулем, но я настолько погрузился в свои мысли, что забыл, что я за рулем. Однажды я проехал на велосипеде мимо собственного дома и попытался попасть в соседнюю квартиру вместо своей.

Я понимаю, что после прочитанного вы едва ли захотите садиться ко мне в машину. Но дело не в том, что меня легко отвлечь, а в том, что мозг работает, даже когда вы этого не осознаёте.

Вы знаете, что ваш мозг контролирует ваше дыхание, даже когда вы об этом не думаете. От его работы зависят ваши двигательные функции, и это тоже происходит неосознанно.

Я был очень потрясен, когда вдруг оказался за рулем своей машины, не помня, как я там оказался. Я ехал по своему родному Копенгагену и, возможно, выполнил ряд сложных маневров. Останавливаясь на красный, поворачивая налево, затем направо, при этом остерегаясь велосипедистов, я прибыл к месту назначения. Но я не помнил, что я делал и как добрался.

Для сложной интеллектуальной работы неважна такая составляющая, как осознанность.

Вы когда-нибудь были настолько *погружены* в работу, что, оторвавшись от монитора, поняли, что на улице уже стемнело и вы просидели так несколько часов? В психологии это называется оказаться в состоянии *потока* [51]. Находясь в нем, вы настолько полностью поглощены какой-либо деятельностью, что теряете контроль на собой и временем.

Вы можете программировать, даже абсолютно не задумываясь над этим, или же писать код, осознавая весь процесс. Дело в том, что в вашей голове происходит многое, о чем вы явно не подозреваете. Ваш мозг выполняет работу, тогда как сознание — это пассивный зритель.

Вам может казаться, что умственный труд — это преднамеренное мышление, но это не так. Психолог и лауреат Нобелевской премии Даниэль Канеман предлагает модель мышления, состоящую из двух систем: *системы 1* и *системы 2*.

> *«Система 1 интуитивная. Она отвечает за быстрое мышление, практически без усилий и без чувства произвольного контроля.*
>
> *Система 2 тяжелая, медленная. Она направляет внимание на требующие усилий умственные действия, включая сложные вычисления. Действия системы 2 часто связаны с субъективным опытом свободы действий, выбора и концентрации» [51].*

Вы, наверное, думаете, что программирование основано только на системе 2, но это не совсем так. Когда вы пытаетесь разобраться в коде, система 1 всегда работает в фоновом режиме. Проблема в том, что она работает быстро, но не очень точно, что может привести к ошибочным выводам. Это и происходит, когда 10 становится первым числом, приходящим вам на ум во время поиска решения задачи о бейсбольной бите и мяче.

Чтобы организовать исходный код так, чтобы наш мозг смог его понять, вы должны научиться контролировать свою систему 1. Канеман пишет:

> *«Важная особенность системы 1 в том, что она обрабатывает только активную информацию, тогда как не извлеченная (даже бессознательно) из памяти может вообще для нее не существовать. Система 1 отлично справляется с выстраиванием лучшей возможной истории, включающей активированные в данный момент идеи, но не учитывает (и не может учитывать) информацию, которой у нее нет.*
>
> *Система 1 ловко придумывает связную историю причин и следствий, объединяющую доступные ей кусочки информации. В случае*

> *неуверенности она делает ставку на тот или иной ответ, исходя из опыта. Система 1 постоянно отслеживает, что происходит внутри и снаружи разума, и генерирует оценку разных аспектов ситуации без конкретного намерения и почти, или совсем, без усилий. Система 1 — это эмоции» [51].*

Ваш мозг всегда склонен делать *поспешные выводы*[1], что может повлиять на ваш код. Лучше организовать код так, чтобы активировалась актуальная информация. Как выразился Канеман, то, что вы видите — это все, что здесь есть (WYSIATI, акроним от англ. What You See Is All There Is) [51].

Становится ясно, почему глобальные переменные и скрытые зависимости делают код непонятным. Когда вы смотрите на фрагмент кода, глобальная переменная обычно не видна. Даже если ваша система 2 знает об этом, это знание неактивно, поэтому система 1 не учитывает его.

Сделайте так, чтобы связанный код было видно полностью. Все зависимости, переменные и необходимые решения должны быть видны одновременно. Позже я приведу много примеров на эту тему, особенно в главе 7.

3.3. НАВСТРЕЧУ ПРОГРАММНОЙ ИНЖЕНЕРИИ

Целью разработки программного обеспечения должна быть поддержка организации, которой оно принадлежит. Вы всегда должны быть готовы вносить изменения в устойчивом темпе.

Но писать код сложно. Вы тратите больше времени на его чтение, чем на написание, поэтому мозг легко ввести в заблуждение — даже такими тривиальными задачами, вроде той, что была в начале главы.

Программная инженерия призвана решить эту проблему.

---

[1] Почему система 1 работает все время, а система 2 может не работать? Одна из причин может заключаться в том, что процесс мышления, требующий усилий, сжигает больше глюкозы [51]. Это подразумевает под собой то, что система 1 — энергосберегающий механизм.

3.3.1. Связь с computer science

Может ли информатика помочь? Почему бы и нет? Но computer science — это не программная инженерия, точно так же, как физика — это не машиностроение.

Такие дисциплины могут взаимодействовать, но это не одно и то же. Ученые могут получить от специалистов-практиков важные данные, а результаты исследований можно применять в инженерии (рис. 3.4).

Рис. 3.4. Наука и инженерия взаимодействуют, но это не одно и то же

Например, результаты computer science можно сравнить с многоразовыми пакетами.

У меня было несколько лет профессионального опыта разработки ПО, прежде чем я узнал об алгоритмах сортировки. Образования в области computer science у меня нет — я научился программированию самостоятельно. И если бы мне нужно было отсортировать массив в C++, Visual Basic или VBScript, я бы вызвал метод.

Для сортировки коллекций вам не обязательно иметь возможность реализовать быструю сортировку или сортировку слиянием. Чтобы выполнять запросы к базе данных, вам не нужно знать о хеш-индексах, таблицах SST, LSM- и B-деревьях[1].

Computer science помогает индустрии разработки программного обеспечения развиваться, но полученные в ней знания часто можно упаковать в многоразовое ПО. Разбираться в computer science никогда не помешает, но это совсем не обязательно, чтобы быть программным инженером.

---

[1] Вот некоторые из структур данных, на которых основаны БД [55].

3.3.2. Гуманный код

Алгоритмы сортировки можно инкапсулировать и распространять в виде повторно используемых библиотек. Сложные структуры хранения и извлечения данных могут быть упакованы в виде программного обеспечения БД общего назначения или предлагаться в виде облачной инфраструктуры.

Но вам все равно придется писать код.

Вы должны структурировать его так, чтобы он «умещался в вашей голове».

Как говорил Мартин Фаулер,

> *«любой дурак может написать код, который будет понятен компьютеру. Хороший программист пишет код, который будет понятен людям» [34].*

Когнитивные ограничения мозга полностью отличаются от ограничений компьютера. Последний может отслеживать миллионы действий в оперативной памяти, тогда как ваш мозг — только семь.

Компьютер будет принимать решения только на основе информации, к которой ему приказано обратиться, а ваш мозг склонен делать поспешные выводы. То, что вы видите, — это все, что здесь есть.

Очевидно, что код нужно писать так, чтобы итоговое ПО работало так, как вы хотите. Это больше не главная проблема программной инженерии. Теперь основная задача: организовать код так, чтобы он «умещался в вашей голове».

Код должен быть *гуманным*, что подразумевает написание небольших автономных функций. В этой книге я использую число семь в качестве показателя пределов кратковременной памяти человека. Исходя из этого гуманный код подразумевает менее семи зависимостей.

Но дьявол кроется в деталях, поэтому готовьтесь к большому количеству примеров.

3.4. ЗАКЛЮЧЕНИЕ

Основная проблема программной инженерии в том, что она настолько сложна, что не укладывается в голове. Фредерик Брукс в 1986 году написал в своем исследовании:

> *«Многие классические проблемы разработки программных продуктов пристекают из этой существенной сложности и ее нелинейного увеличения [...] Из-за сложности возникает трудность перечисления, а тем более понимания всех возможных состояний программы» [14].*

Я использую термин *«сложность»* в том же смысле, в каком его использует Рич Хикки [45]: как антоним простоте. *«Сложный»* означает «собранный из частей», в отличие от *«простого»*, подразумевающего «единство».

Человеческий мозг способен справиться с ограниченной сложностью, а наша кратковременная память может запомнить около семи действий. Из-за своей невнимательности мы можем легко написать код, который обрабатывает более семи действий одновременно. Компьютеру все равно, поэтому он нас не остановит.

Цель программной инженерии — быть преднамеренным процессом предотвращения разрастания *сложности*.

Возможно, вы абстрагируетесь от всего этого, так как думаете, что это вас затормозит.

Но в этом и суть. Перефразируя Дж. Б. Райнсбергера [86], вам нужно замедлиться. Чем быстрее вы печатаете, тем больше делаете кода, который все должны поддерживать. Код — это не актив, это ответственность [77].

Как утверждает Мартин Фаулер, именно применяя хорошую архитектуру, вы можете поддерживать стабильный темп [32]. Программная инженерия служит средством для достижения этой цели. Это попытка превратить разработку ПО из чистого искусства в *методологию*.

4 ВЕРТИКАЛЬНЫЙ СРЕЗ

Однажды мой постоянный клиент попросил меня помочь с проектом. Когда я приехал к нему, я узнал, что команда работала над задачей около полугода и ничего не добилась.

Их задача действительно была сложной, но они впали в аналитический паралич [15]. Требований было настолько много, что команда никак не могла найти решение, чтобы все их удовлетворить. Я не раз видел, как подобное случалось с разными командами.

Иногда лучшая стратегия — просто начать. Вы все еще должны думать и планировать. Нет причин быть преднамеренно безразличными или скептически настроенными. Но недостаток планирования может так же плохо сказаться на вашей работе, как и его переизбыток. Если вы уже установили свой конвейер развертывания [49], то чем раньше вы сможете развернуть часть работающего ПО, независимо от его сложности, тем раньше вы начнете собирать отзывы заинтересованных сторон [29].

Для начала создайте вертикальный срез приложения и запустите процесс его развертывания.

4.1. НАЧНИТЕ С РАБОЧЕГО ПО

Как узнать, что ПО работает? Вы не будете в этом уверены, пока не отправите его в продакшен. Только после того, как программа будет развернута или установлена и будет применяться реальными пользователями, вы сможете проверить, работает ли она. Но это еще не все. Ваше программное обеспечение может работать так, как вы предполагали, но не решать реальные проблемы пользователей. Эту часть я не рассматриваю в своей книге, поэтому остановлюсь здесь[1]. Для меня *программная инженерия* — это методология, позволяющая убедиться в том, что программа работает так, как задумано.

Идея вертикального среза в том, чтобы как можно скорее получить работающее ПО. Для этого вы реализуете самую простую функциональность, которую только можно представить, — на всем пути от пользовательского интерфейса до хранилища данных.

4.1.1. От поступления данных до их сохранения

Бóльшая часть программного обеспечения сопровождается двумя типами внешних границ. Вы, наверное, уже знакомы с диаграммами, подобными рис. 4.1. Данные приходят на верхний уровень. Приложение может подвергнуть ввод разным изменениям и в конечном счете решить сохранить их.

Даже операцию *чтения* можно считать входными данными, хотя она и не приводит к их сохранению. Запрос обычно содержит параметры, которые идентифицируют запрашиваемые данные. ПО по-прежнему преобразует эти входные значения во взаимодействие со своим хранилищем.

Иногда хранилище данных представляет собой выделенную базу, а иногда это просто другая система. Это может быть HTTP-сервис где-то в интернете, очередь сообщений, файловая система или даже стандартный поток вывода локального компьютера.

---

[1] Книги «Бизнес с нуля» [88] и «Ускоряйся!» [29] — хорошее начало для изучения этой темы.

Это могут быть системы только для записи (например, стандартный поток вывода), только для чтения (например, сторонний HTTP API) или для чтения-записи (например, файловая система или базы данных).

Диаграмма (достаточно высокого уровня абстракции) на рис. 4.1 описывает бóльшую часть программного обеспечения, от веб-сайтов до утилит командной строки.

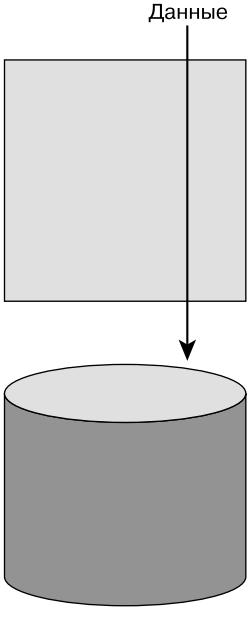


Рис. 4.1. Типовая архитектурная схема. Данные поступают с верхнего уровня, проходят через приложение (блок) и сохраняются на нижнем уровне (в цилиндре)

4.1.2. Минимальный вертикальный срез

Вы можете организовать код по-разному. Классическая архитектура предполагает организацию составляющих элементов в слои [26; 33; 50; 60]. Вам не обязательно делать это именно так, но, ссылаясь на многоуровневую архитектуру приложений, можно объяснить, почему это называется *вертикальным* срезом.

> Вам не нужно организовывать свой код слоями. Многоуровневая архитектура описана в этом разделе только для того, чтобы объяснить, почему это называется *вертикальным* срезом.

Как показано на рис. 4.2, обычно рассматриваются горизонтальные слои, в которых данные поступают наверх, а сохраняются внизу. Чтобы реализовать целостную функциональность, вам придется перемещать данные от точки входа до уровня сохранения или наоборот. Если слои расположены горизонтально, тогда одна отдельная функциональность представляет собой вертикальный срез.

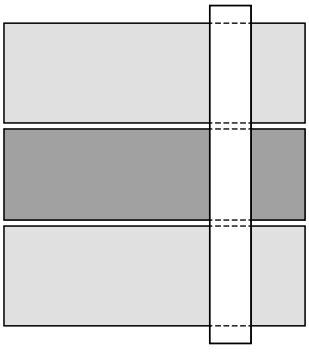


Рис. 4.2. Вертикальный срез горизонтальных уровней стереотипной архитектуры приложения

Независимо от того, организуете вы свой код слоями или как-то иначе, у реализации сквозной функциональности как минимум два преимущества.

1. Это дает вам быструю обратную связь обо всем жизненном цикле вашего процесса разработки программного обеспечения.
2. Это рабочее ПО, которое сможет кому-то пригодиться.

Я встречал разработчиков, которые месяцами совершенствовали самодельный фреймворк доступа к данным, прежде чем хотя бы попробовать использовать его для реализации функциональности.

Как правило, они узнают, что их предположения об использовании не соответствуют действительности.

Вам следует избегать спекулятивной общности [34], тенденции добавлять функциональность в код, так как она «может понадобиться позже». Вместо этого реализуйте функциональность с максимально простым кодом, но следите за наличием дублей по мере их добавления.

Реализация вертикального среза — эффективный способ узнать, какой код вам нужен, а без чего можно обойтись.

4.2. «ХОДЯЧИЙ СКЕЛЕТ»

Найдите мотивацию для внесения изменений в код, ту, которая будет подталкивать вас к ним.

Примеры таких «двигателей изменений» вам уже знакомы: когда вы относитесь к предупреждениям как к ошибкам, когда включаете линтеры и другие статические анализаторы, вы мотивируете код на изменения. Это может быть полезно, так как добавляет объективности в вашу работу.

Использование таких стимулов порождает целое семейство методологий *x-ориентированной* разработки ПО, основанных на:

- разработке через тестирование [9] (TDD);
- разработке через поведение (BDD);
- предметно-ориентированном проектировании [26] (DDD);
- типоориентированной разработке;
- тестировании на основе свойств[1].

Вспомните задачу о бейсбольной бите и мяче — ошибиться очень легко. Использование внешнего стимула несколько похоже на двойную бухгалтерию [63]. Вы как-то взаимодействуете с ним, и он побуждает вас вносить изменения в код.

---

[1] Пример тестирования на основе свойств приведен в подразделе 15.3.1.

Таким стимулом может быть линтер или же код в виде автоматизированных тестов. Обычно я следую внешнему стилю разработки через тестирование — методу, при котором первые тесты, которые вы пишете, проверяют высокоуровневую границу тестируемой системы. Затем вы можете углубляться, добавляя по мере необходимости тесты для более мелких деталей реализации. Подробнее об этом я расскажу в разделе 4.3.

Вам понадобится набор тестов.

4.2.1. Характеризационные тесты

В оставшейся части этой главы я объясню вам, как добавить вертикальный срез в HTTP API для резервирования столиков в ресторане, код которого мы уже начали писать в подразделе 2.2.2. Сейчас результат работы кода — это вывод простого текста `Hello World!`.

Если добавить простое автоматизированное тестирование системы, то это уже будет началом разработки через тестирование. У вас в коде будет некоторая функциональность, которую вы сможете автоматически тестировать и разворачивать: «ходячий скелет» [36].

При добавлении проекта модульного тестирования в решение Visual Studio следуйте *чек-листу новой кодовой базы* из раздела 2.2: добавьте в Git новый тестовый проект, обработайте предупреждения как ошибки и убедитесь, что автоматизированная сборка запускает тест.

Теперь добавьте свой первый тест-кейс, как в листинге 4.1.

Листинг 4.1. Интеграционный тест домашнего ресурса HTTP (Restaurant/3ee0733/Restaurant.RestApi.Tests/HomeTests.cs)

```
[Fact]
public async Task HomeIsOk()
{
    using var factory = new WebApplicationFactory<Startup>();
    var client = factory.CreateClient();

    var response = await client
        .GetAsync(new Uri("", UriKind.Relative))
        .ConfigureAwait(false);
```

```
    Assert.True(
        response.IsSuccessStatusCode,
        $"Actual status code: {response.StatusCode}.");
}
```

Чтобы внести ясность: я написал этот тест постфактум, поэтому не придерживался разработки через тестирование. Скорее всего такой тип можно назвать характеризационным тестом [27], так как он характеризует *(описывает)* поведение существующего ПО.

Я сделал так, потому что программа уже существует. Как вы помните, в главе 2 я использовал *мастер-ветку* для создания исходного кода. Сейчас он работает как задумано. Но откуда мы знаем, что он продолжит работать?

Считаю, что для защиты от регрессии лучше добавить автоматические тесты.

В тесте из листинга 4.1 используется среда модульного тестирования xUnit.net. Ее я и буду использовать в примерах на протяжении всей книги. Даже если вы не знакомы с этой средой, на основе примеров вы сможете легко все изучить [66].

Для создания автономного экземпляра HTTP-приложения используется специфический класс `WebApplicationFactory<T>`. Класс `Startup` (см. листинг 2.5) определяет и запускает само приложение.

Обратите внимание, что утверждение рассматривает только самое поверхностное свойство системы: будет ли получен ответ HTTP в категории `200` (например, `200 OK` или `201 Created`)? Я решил не проверять что-то более сложное, ведь сейчас это только плейсхолдер (возвращает `Hello World!`). В будущем это должно измениться.

Если логическое выражение истинно, единственное сообщение, которое вы получите от библиотеки утверждений, покажет, что ожидалось `true`, но фактическое значение было `false`. Это едва ли прояснит ситуацию, поэтому нужно добавить немного контекста. Для этого я использовал перегрузку `Assert.True`, принимающую дополнительное сообщение в качестве второго аргумента.

Тест получился слишком многословным, но компилируется и выполняется успешно. Сейчас мы улучшим тестовый код, но всегда учитывайте *чек-лист для новой кодовой базы*. Сделал ли я что-нибудь, что скрипт сборки должен автоматизировать?

Да, я добавил набор тестов. Измените скрипт сборки, чтобы запустить тесты, как показано в листинге 4.2.

Листинг 4.2. Скрипт сборки с тестами (Restaurant/3ee0733/build.sh)

```
#!/usr/bin/env bash
dotnet test --configuration Release
```

Единственное различие с листингом 2.3 здесь в том, что в листинге 4.2 вызывается команда `dotnet test` вместо `dotnet build`.

Помните о своем чек-листе. Сделайте коммит изменений в Git.

4.2.2. Паттерн AAA (Arrange-Act-Assert)

В листинге 4.1 представлена структура теста. Код начинается с двух строк, за которыми следует пустая строка, затем один оператор, занимающий еще три строки, за которым следует пустая строка, и, наконец, еще один оператор, занимающий три строки.

Бóльшая часть этой структуры — результат продуманной методологии. Сейчас я не буду объяснять, почему некоторые операторы занимают несколько строк. Вы можете прочитать об этом в разделе 7.1.3.

Пустые же строки есть потому, что код следует шаблону Arrange-Act-Assert [9], также известному как шаблон AAA. Идея здесь в том, чтобы разделить модульный тест на три этапа.

1. На этапе *arrange* (подготовка) настраивается тестовое окружение.
2. На этапе *act* (действие) происходит выполнение или вызов тестируемого сценария.
3. На этапе *assert* (утверждение) проверяется, соответствует ли фактический результат ожидаемому.

Вы можете преобразовать этот паттерн в эвристику. Я обычно указываю три этапа, разделяя их пустой строкой (см. листинг 4.1).

Это работает, только если вы можете избежать дополнительных пустых строк в тесте. Часто этап *arrange* становится настолько длительным, что возникает необходимость в форматировании путем добавления пустых строк. В итоге в вашем тесте будет более двух пустых строк и не будет ясно, какая из них разграничивает три этапа.

Когда тестовый код становится слишком большим, его можно считать проблемным [34]. Удобнее, когда три этапа сбалансированы. Этап *act* обычно самый маленький, но, если представить, что вы поворачиваете код на 90° (рис. 4.3), вы сможете приблизительно сбалансировать код на нем.

Если тестовый код настолько большой, что вам пришлось добавить дополнительные пустые строки, придется использовать комментарии, чтобы обозначить три этапа [92]. Но постарайтесь этого избегать.

С другой стороны, вы можете написать маленький тестовый код. Если у вас только три строки кода, и каждая из них относится к каждому из этапов AAA, вы можете обойтись без пустых строк. Аналогично, если у вас есть только одна или две строки кода. Цель шаблона AAA — сделать тест более читабельным за счет добавления хорошо известной структуры. Если у вас всего две-три строки кода, скорее всего, тест настолько мал, что его уже можно читать как есть.

4.2.3. Модерация статического анализа

Хотя в листинге 4.1 всего несколько строк кода, я все же считаю его слишком большим.

Например, этап *act* мог бы быть более читабельным, но есть две проблемы.

1. Вызов метода `ConfigureAwait` добавляет что-то похожее на шум.
2. Довольно запутанный способ передачи пустой строки в качестве аргумента.

Рассмотрим это на примере.

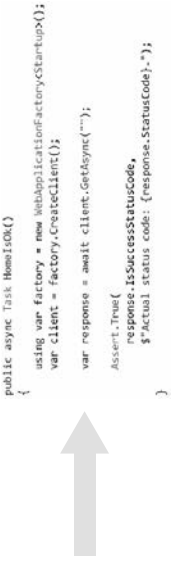


Рис. 4.3. Представьте, что вы повернули тестовый код на 90° (это может быть любой блок кода модульного тестирования). Если точка его опоры будет примерно на этапе act, то он будет считаться сбалансированным

Если метод `ConfigureAwait` избыточен, то зачем он там? Потому что иначе код не скомпилируется. Я настроил тестовый проект в соответствии с *чек-листом новой кодовой базы*, включающим в себя добавление статического анализа кода и преобразование всех предупреждений в ошибки.

В одном из этих правил[1] рекомендуется вызывать метод `ConfigureAwait` для ожидаемых задач. К правилу прилагается документация, которая объясняет мотивацию. В общем, по умолчанию задача возобновляется в потоке, который ее первоначально создал. Вызывая `ConfigureAwait(false)`, вы указываете, что вместо этого задача может

[1] CA2007: Do not directly await a task («Не ждать задачи напрямую»).

возобновиться в любом потоке. Это позволит избежать взаимоблокировок и некоторых проблем с производительностью. Правило настоятельно рекомендует вызывать этот метод в коде, реализующем повторно используемую библиотеку.

Но тестовую библиотеку нельзя использовать повторно. Клиенты известны заранее: две-три стандартные системы выполнения тестов, включая встроенный инструмент запуска Visual Studio и тот, который используется вашим сервером непрерывной интеграции.

В документации есть и раздел о том, когда безопасно правило деактивировать. Библиотека модульного тестирования подходит под описание, так что вы можете отключить ее, чтобы убрать шум из ваших тестов.

Помните, что, хотя это правило можно отключить для модульных тестов, оно должно оставаться в силе при разработке окончательной версии кода. В листинге 4.3 показан пример характеризационного теста после очистки.

Еще одна проблема листинга 4.1 в том, что метод `GetAsync` включает перегрузку, которая принимает строку (`string`) вместо объекта `Uri`. Можно упростить тест, добавив `""` вместо `new Uri("", UriKind.Relative)`. К сожалению, другое правило статического анализа кода[1] не рекомендует использовать такую перегрузку.

Вам следует избегать использования строкотипизированного (stringly typed) [3] кода[2]. Вместо передачи строк используйте объекты с хорошей инкапсуляцией. Что касается меня, то я приветствую такой принцип разработки, поэтому не намерен деактивировать правило, как я сделал с правилом, касающимся `ConfigureAwait`.

Тем не менее я считаю, что мы можем сделать принципиальное исключение из правил. Итак, вам нужно заполнить объект `Uri` строкой (`string`).

Преимущество `Uri` перед `string` в том, что принимающая сторона знает, что инкапсулированный объект обеспечивает более надежные

---

[1] CA2234: Pass System.Uri objects instead of strings («Передавать объекты System.Uri вместо строк»).

[2] Также известен как одержимость примитивами [34].

гарантии, чем строка[1]. На площадке, где вы создаете объект, разница будет незаметна, поэтому я считаю справедливым подавить предупреждение, так как код содержит строковый *литерал*, а не переменную.

Листинг 4.3. Тест с облегченными правилами анализа кода [Fact]

```
[Fact]
[SuppressMessage(
    "Usage", "CA2234:Pass system uri objects instead of strings",
    Justification = "URL isn't passed as variable, but as literal.")]
public async Task HomeIsOk()
{
    using var factory = new WebApplicationFactory<Startup>();
    var client = factory.CreateClient();

    var response = await client.GetAsync("");

    Assert.True(
        response.IsSuccessStatusCode,
        $"Actual status code: {response.StatusCode}.");
}
```

Листинг 4.3 — пример результата подавления правила `ConfigureAwait` для всех тестов и правила `Uri` для конкретного. Обратите внимание, что этап *act* сократился с трех строк кода до одной. Главная цель — упростить код. Код, который я удалил, был (в этом контексте) ненужным.

Как вы видите, я скрыл рекомендацию `Uri` с помощью атрибута метода тестирования. Обратите внимание, что я предоставил письменное обоснование (`Justification`) своего решения. Как я утверждал в главе 3, код — это единственное, что действительно важно. Будущим разработчикам может понадобиться понять, почему код организован именно так[2].

> Документация должна в первую очередь объяснять, *почему* было принято решение, а не *какое* решение было принято.

Каким бы полезным ни был статический анализ кода, всегда будут ложные срабатывания. Можно отключить правила или подавить

---

[1] Подробнее о гарантиях и инкапсуляции читайте в главе 5.

[2] Вы можете восстановить все изменения из вашей истории Git. Гораздо сложнее восстановить причину этих изменений.

конкретные предупреждения, но нужно делать это осознанно. По крайней мере, задокументируйте причину своего решения и по возможности получите обратную связь о нем.

4.3. МОДЕЛЬ ТЕСТИРОВАНИЯ «ОТ ОБЩЕГО К ЧАСТНОМУ» (OUTSIDE-IN)

Теперь мы готовы ускориться. Есть система, отвечающая на HTTP-запросы (хотя и мало что делает), и есть автоматический тест. Это наш «ходячий скелет» [36].

Система должна делать что-то полезное. Цель этой главы — реализация вертикального среза системы от границы HTTP до хранилища данных. Как уже было описано в подразделе 2.2.2, наша программа должна быть простой системой онлайн-бронирования столиков в ресторане. Я думаю, что хорошим кандидатом для среза является возможность получать действительный запрос на резервирование и сохранять информацию в базе данных (рис. 4.4).

Система должна представлять собой HTTP API, который получает документы JSON и отвечает на них — так будет происходить взаимодействие системы с остальным миром. Это связь с внешними клиентами, поэтому важно ее соблюдать.

Как предотвращать регрессии? Один из способов — написать набор автоматических тестов для границы HTTP. Если вы можете писать тесты перед реализацией, значит, у вас есть для этого *драйвер*.

Такой тест может выполнять двойную функцию автоматизированного приемочного тестирования [49], поэтому вы можете назвать этот процесс *разработкой, управляемой приемочным тестированием*. Я предпочитаю называть это *разработкой, ориентированной на тестирование «от общего к частному»*[1], ведь, хотя вы и начинаете с границы, вам нужно двигаться вперед. Скоро мы рассмотрим это на примере.

[1] Не я придумал этот термин, но где я его услышал — не помню. Но с этой идеей я впервые столкнулся в книге Growing Object-Oriented Software, Guided by Tests [36].

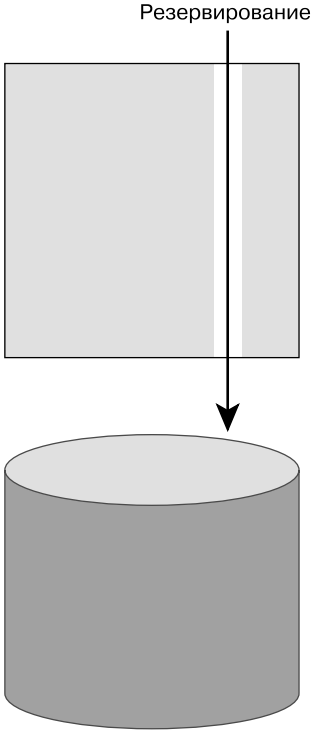


Рис. 4.4. План такой: создать вертикальный срез системы, который получает действительный запрос на резервирование и сохраняет его в базе данных

4.3.1. Получение данных JSON

Когда вы начинаете работу над новой кодовой базой, будьте готовы потратить большое количество времени. Двигаться маленькими шагами может быть трудно, но вам придется постараться. Начнем изучение примера резервирования столика в ресторане с проверки того, что ответ от API — это документ JSON.

Сейчас мы знаем, что это не так: веб-приложение просто возвращает строку `Hello World!` в формате обычного текстового документа. Качественный ориентированный на тесты подход предполагает написание

нового теста, утверждающего, что ответ должен быть в формате JSON, но бóльшая его часть будет повторять существующий тест из листинга 4.3. Вместо того чтобы дублировать тестовый код, попробуйте доработать существующий тест. В листинге 4.4 приведен пример расширенного теста.

Листинг 4.4. Тест, подтверждающий возврат в формате JSON (Restaurant/316beab/Restaurant.RestApi.Tests/HomeTests.cs)

```
[Fact]
[SuppressMessage(
    "Usage", "CA2234:Pass system uri objects instead of strings",
    Justification = "URL isn't passed as variable, but as literal.")]
public async Task HomeReturnsJson()
{
    using var factory = new WebApplicationFactory<Startup>();
    var client = factory.CreateClient();

    using var request = new HttpRequestMessage(HttpMethod.Get, "");
    request.Headers.Accept.ParseAdd("application/json");
    var response = await client.SendAsync(request);

    Assert.True(
        response.IsSuccessStatusCode,
        $"Actual status code: {response.StatusCode}.");
    Assert.Equal(
        "application/json",
        response.Content.Headers.ContentType?.MediaType);
}
```

Что изменилось?

1. Я конкретизировал название теста.
2. Тест теперь явно устанавливает заголовок `Accept` запроса в `application/json`.
3. Я добавил второе утверждение.

Установив заголовок `Accept`, клиент задействует протокол согласования контента HTTP [2]. Если сервер может обслужить ответ JSON, он должен это сделать.

Чтобы убедиться в этом, я добавил второе утверждение, которое проверяет `Content-Type`[1] ответа.

На втором утверждении тест терпит неудачу. Ожидается, что заголовок `Content-Type` будет `application/json`, но фактически он принимает значение `null`. Это больше похоже на разработку через тестирование (TDD): написать провальный тест, а затем сделать так, чтобы он прошел.

При работе с ASP.NET нужно следовать шаблону Model-View-Controller [33] (MVC). В листинге 4.5 показана простейшая реализация контроллера, которую я смог реализовать.

Листинг 4.5. Первая реализация HomeController (Restaurant/316beab/Restaurant.RestApi/HomeController.cs)

```
[Route("")]
public class HomeController : ControllerBase
{

    public IActionResult Get()
    {
        return Ok(new { message = "Hello, World!" });
    }
}
```

Но этого недостаточно. Вы должны указать ASP.NET использовать свою структуру MVC. Это можно сделать с помощью класса `Startup` (листинг 4.6).

---

[1] Возможно, вы слышали, что в тесте должно быть только одно утверждение и что наличие нескольких утверждений называется Assertion Roulette (рулеткой утверждений), что приводит к коду «с запашком». Assertion Roulette — это действительно проблемный код, но наличие нескольких утверждений не обязательно является его примером. Рулетка утверждений — это когда вы неоднократно чередуете этапы assert с дополнительным кодом этапов assert и act или когда в утверждении отсутствует информативное сообщение [66].

Листинг 4.6. Настройка ASP.NET для MVC (Restaurant/316beab/Restaurant.RestApi/Startup.cs)

```
public sealed class Startup
{

    public static void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
    }

    public static void Configure(
        IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
            app.UseDeveloperExceptionPage();

        app.UseRouting();
        app.UseEndpoints(endpoints => { endpoints.MapControllers(); });
    }
}
```

По сравнению с листингом 2.5 код в листинге 4.6 выглядит гораздо проще. Я считаю это прогрессом.

Благодаря этим изменениям тест из листинга 4.4 проходит успешно. Сделайте коммит изменений в Git и рассмотрите их пуск через конвейер развертывания [49].

4.3.2. Размещение бронирования

Напомню, что цель вертикального среза — показать, что система работает. Мы потратили некоторое время на то, чтобы разобраться, с чего начать. Это нормально при работе с новой кодовой базой, но теперь она готова.

Выбирая функциональность для первого вертикального среза, я обращаю внимание на несколько моментов.

1. Функциональность должна быть простой в реализации.
2. По возможности отдавайте предпочтение вводу данных.

При разработке систем с постоянными данными у вас возникнет необходимость в данных для тестирования других вещей. Если начать с функциональности, добавляющей данные в систему, можно аккуратно решить эту проблему.

В нашем примере будет полезно разрешить веб-приложению получать и сохранять резервирование столика в ресторане. Используя outside-in-разработку через тестирование, вы можете написать тест как в листинге 4.7.

Следуя правилам вертикального среза, вы будете двигаться в верном направлении [66]. Сейчас старайтесь игнорировать все, что может пойти не так[1]. Ваша цель — продемонстрировать, что у системы есть определенные возможности. В нашем примере ожидаемым результатом будет получение и сохранение информации о резервировании.

Код листинга 4.7 отправляет *валидные* данные о брони в сервис. Эти данные должны включать фактическую дату, адрес электронной почты, имя и количество человек. В тесте для эмуляции объекта JSON используется анонимный тип. При сериализации результирующий JSON имеет ту же структуру и те же имена полей.

Листинг 4.7. Проверка того, что запрос на резервирование может быть передан в HTTP API. Метод PostReservation приведен в листинге 4.8 (Restaurant/90e4869/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Fact]
public async Task PostValidReservation()
{
    var response = await PostReservation(new {
        date = "2023-03-10 19:00",
        email = "katinka@example.com",
        name = "Katinka Ingabogovinanana",
        quantity = 2 });

    Assert.True(
        response.IsSuccessStatusCode,
        $"Actual status code: {response.StatusCode}.");
}
```

[1] В любом случае, чтобы не забыть, запишите все, что может пойти не так [9].

Высокоуровневые тесты должны легко справляться с утверждениями. В процессе разработки многие данные будут меняться, и если вы сделаете утверждения слишком конкретными, вам придется их часто исправлять. Лучше не переусердствовать. Тест в листинге 4.7 проверяет только, соответствует ли код состояния HTTP успешному выполнению, как обсуждалось в подразделе 4.2.1. По мере добавления тестового кода вы будете все более подробно описывать ожидаемое поведение системы, повторяя эту же операцию.

Вы также могли заметить, что тест делегирует все действия методу `PostReservation`. Это вспомогательный метод теста (test utility method) [66] из листинга 4.8.

Бóльшая часть кода аналогична листингу 4.4. Я мог бы написать это в самом тесте. Почему я этого не сделал? На то есть несколько причин, но именно здесь я хотел показать, что программная инженерия — больше искусство, чем наука.

Одна из причин в том, что это упростит код и будет видно только самое необходимое: вы отправляете некоторые значения в службу, а полученный ответ указывает на успех. По словам Роберта Мартина, это отличный пример абстракции:

> *«Абстракция — это устранение неважного и усиление существенного» [60].*

Листинг 4.8. Вспомогательный метод PostReservation. Данный метод определен в кодовой базе теста (Restaurant/90e4869/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[SuppressMessage(
    "Usage",
    "CA2234:Pass system uri objects instead of strings",
    Justification = "URL isn't passed as variable, but as literal.")]
private async Task<HttpResponseMessage> PostReservation(
    object reservation)
{
    using var factory = new WebApplicationFactory<Startup>();
    var client = factory.CreateClient();
```

```
    string json = JsonSerializer.Serialize(reservation);
    using var content = new StringContent(json);
    content.Headers.ContentType.MediaType = "application/json";
    return await client.PostAsync("reservations", content);
}
```

Еще одна причина: мне бы хотелось оставить за собой право изменять код. Обратите внимание, что последняя строка кода вызывает `PostAsync` с жестко заданным относительным путем `reservations`. Это значит, что ресурс бронирования существует по URL-адресу, например https://api.example.com/reservations. Это может быть так, но вам это может быть совсем не нужно.

Вы можете написать HTTP API с опубликованными шаблонами URL, но это не будет REST, так как сложно изменить API, не нарушая контракт [2]. API, которые предполагают, что клиенты будут использовать задокументированные шаблоны URL-адресов, используют команды HTTP, но не элементы управления Hypermedia[1].

Слишком сложно, чтобы настаивать на элементах управления Hypermedia (на *ссылках*) прямо сейчас, поэтому вы можете использовать метод инкапсуляции SUT[2] [66].

Хочу сделать одно замечание по поводу листинга 4.8: я решил подавить правило анализа кода, предлагающее объекты `Uri`, по той же причине, что и в подразделе 4.2.3.

После запуска тест, как и ожидалось, терпит неудачу. Assertion Message (сообщение подтверждения) [66] имеет *фактический код состояния: NotFound* (Actual status code: NotFound). Значит, на сервере нет ресурса `/reservations`. Неудивительно, ведь мы еще не реализовали его.

Сделать это легко. В листинге 4.9 представлен пример минимальной реализации, которая проходит все существующие тесты.

---

[1] Модель зрелости Ричардсона для сервиса REST различает три уровня: 1) ресурсы; 2) команды HTTP; 3) управление элементами Hypermedia [114].

[2] System Under Test — тестируемая система.

Листинг 4.9. Минимальная реализация ReservationsController (Restaurant/90e4869/Restaurant.RestApi/ReservationsController.cs)

```
[Route("[controller]")]
public class ReservationsController
{
#pragma warning disable CA1822 // Помечаем члены как статические
    public void Post() { }
#pragma warning restore CA1822 // Помечаем члены как статические
}
```

Первое, что вы видите, — это безобразные инструкции `#pragma`. Из их комментариев следует, что они подавляют правило статического анализа кода, настаивающее на том, чтобы сделать метод `Post` статическим (`static`). Но вы не можете так сделать. Если вы вызовете метод `static`, тест завершится неудачно. Платформа ASP.NET MVC сопоставляет HTTP-запросы с методами контроллера по соглашению, а методы должны быть методами экземпляра (то есть не `static`).

Есть несколько способов подавить предупреждения от анализаторов `.NET`, и я намеренно выбрал самый плохой из них вместо того, чтобы оставить комментарий `//TODO`. Надеюсь, что инструкции `#pragma` имеют такой же эффект.

Метод `Post` сейчас не работает, и очевидно, что так быть не должно. Но для успешной компиляции кода вы должны временно подавить предупреждение. Рассматривать предупреждения как ошибки непросто. Этот процесс требует времени, но я считаю, что он необходим. Помните: цель не в том, чтобы как можно быстрее написать как можно больше строк кода, а в том, чтобы создать надежное ПО.

> Цель не в быстром написании кода. Главная цель — надежное ПО.

Теперь, когда все тесты выполнены успешно, сделайте коммит изменений в Git и рассмотрите их пуск через конвейер развертывания [49].

4.3.3. Модульное тестирование

В листинге 4.9 мы видим, что веб-сервис не обрабатывает отправленную информацию о резервировании.

Для дальнейшего продвижения можно использовать другой тест (листинг 4.10).

Листинг 4.10. Модульный тест передачи валидного резервирования (Restaurant/bc1079a/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Fact]
public async Task PostValidReservationWhenDatabaseIsEmpty()
{
    var db = new FakeDatabase();
    var sut = new ReservationsController(db);

    var dto = new ReservationDto
    {
        At = "2023-11-24 19:00",
        Email = "juliad@example.net",
        Name = "Julia Domna",
        Quantity = 5
    };
    await sut.Post(dto);

    var expected = new Reservation(
        new DateTime(2023, 11, 24, 19, 0, 0),
        dto.Email,
        dto.Name,
        dto.Quantity);
    Assert.
}
```

В отличие от прошлых тестов это не проверка системного HTTP API. Это модульный тест[1]. Он иллюстрирует ключевую идею, лежащую

[1] Термин «модульный тест» определен неясно. Не существует единого мнения по поводу его определения. Я склоняюсь к тому, чтобы определить его как автоматизированный тест, проверяющий модуль отдельно от его зависимостей. Обратите внимание, что это определение все еще расплывчато. Как правило, я думаю о модуле как о небольшом фрагменте кода, но он может быть настолько мал, что также может быть плохо определен.

в основе *outside-in-разработки через тестирование*: находясь на границе системы, вы должны проложить себе путь внутрь.

«Но граница системы — это место, где она взаимодействует с внешним миром, — возразите вы. — Разве мы не должны проверить его *поведение*?» Звучит уместно, но, к сожалению, нецелесообразно.

Попытка охватить все поведение и пограничные случаи с помощью граничных тестов приводит к комбинаторному взрыву[1]. Для этого вам нужно написать десятки тысяч тестов [85]. Переход от внешнего тестирования к изолированному тестированию отдельных модулей решает эту проблему.

Хотя модульный тест в листинге 4.10 и выглядит простым, на самом деле это не так. Это еще один пример абстракции: усилить существенное и исключить ненужное. Очевидно, что никакой код не имеет значения. Суть в том, что для того, чтобы понять общую цель теста, вам (пока) не нужно разбираться во всех деталях `ReservationDto`, `Reservation` или `FakeDatabase`.

Тест построен в соответствии с шаблоном AAA [9] [92]. Каждый этап разделяется пустой строкой. На этапе *arrange* создаются `FakeDatabase` и тестируемая система (SUT) [66].

На этапе *act* создается объект передачи данных (DTO) [33] и передается методу `Post`. Можно создать и `dto` как часть этапа *arrange*. Вы можете использовать обе альтернативы, но я предпочитаю более сбалансированную, как описал в подразделе 4.2.2. В нашем случае на каждом этапе есть два оператора. Я думаю, что структура 2-2-2 лучше сбалансирована, чем 3-1-2, получаемая, если вы перенесете инициализацию `dto` на этап *arrange*.

Наконец, на этапе *assert* проверяется, что БД содержит ожидаемую информацию о резервировании.

---

[1] Комбинаторный взрыв — термин, используемый для описания эффекта резкого (взрывного) роста временной сложности алгоритма при увеличении размера входных данных задачи. — *Примеч. ред.*

Это описывает общий ход теста и причину, по которой он структурирован именно так. Надеюсь, что представленные здесь абстракции были вам полезны, даже если вы еще не видели новые классы. Прежде чем рассмотреть пример из листинга 4.11, представьте, как выглядит `ReservationDto`.

4.3.4. DTO и модель предметной области (доменная модель)

Вы удивлены? Это совершенно обычный C# DTO, единственная цель которого — отражать структуру входящего документа JSON и фиксировать его составляющие значения.

Листинг 4.11. Резервирование DTO. Это часть окончательной версии кода (Restaurant/bc1079a/Restaurant.RestApi/ReservationDto.cs)

```
public class ReservationDto
{
    public string? At { get; set; }
    public string? Email { get; set; }
    public string? Name { get; set; }
    public int Quantity { get; set; }
}
```

Как, по-вашему, выглядит код примера с `Reservation`? Почему код вообще содержит два класса с одинаковыми именами? Но на то есть причина. Да, они оба представляют собой *резервирование*, но перед ними поставлены разные *цели*.

Цель DTO — записывать входящие данные в структуру данных или помогать преобразовывать ее в выходные данные. Вы не можете использовать его ни для чего другого, так как этот класс не предлагает инкапсуляцию. Мартин Фаулер говорит об этом так:

> *«Если бы я был заботливой мамой, то обязательно сказал бы своему ребенку: “Никогда не пиши объекты переноса данных!”». [33]*

С другой стороны, цель класса `Reservation` — инкапсулировать бизнес-правила, применимые к резервированию. Это часть модели предметной области кода [33; 26]. В листинге 4.12 показана его начальная версия. Он кажется более сложным[1], чем листинг 4.11, но на самом деле это не так: листинг 4.12 состоит из такого же количества составных частей.

Листинг 4.12. Класс Reservation как часть модели предметной области (Restaurant/bc1079a/Restaurant.RestApi/Reservation.cs)

```
public sealed class Reservation
{
    public Reservation(
        DateTime at,
        string email,
        string name,
        int quantity)
    {
        At = at;
        Email = email;
        Name = name;
        Quantity = quantity;
    }
    public DateTime At { get; }
    public string Email { get; }
    public string Name { get; }
    public int Quantity { get; }

    public override bool Equals(object? obj)
    {
        return obj is Reservation reservation &&
            At == reservation.At &&
            Email == reservation.Email &&
            Name == reservation.Name &&
            Quantity == reservation.Quantity;
    }
    public override int GetHashCode()
    {
        return HashCode.Combine(At, Email, Name, Quantity);
    }
}
```

[1] Я использую слово complex (англ. «сложный»), имея в виду, что код собран из частей [45]. Это не синоним английского слова complicated.

У вас могут возникнуть сомнения вроде: «Но там так много кода! Ты нас не обманул? Где тесты, которые привели тебя к этой реализации?»

Я не писал тестов класса `Reservation` (кроме листинга 4.10) и никогда не говорил, что буду строго придерживаться разработки через тестирование.

Ранее в этой главе я говорил о том, что не доверю сам себе написание правильного кода. Опять же вспомните задачу о бите и мяче, если вам нужно напоминание о том, как легко обмануть мозг. Но я доверяю инструменту для написания кода за меня. Хотя я не поклонник автоматически генерируемого кода, Visual Studio сделала бóльшую часть работы в листинге 4.12.

Я написал четыре свойства только для чтения, а затем использовал инструмент Visual Studio, чтобы добавить конструктор, и инструмент создания Equals и GetHashCode для всего остального. Я верю, что Microsoft тестирует функции, которые включает в свои продукты.

Разве `Reservation` не лучше инкапсулирует бизнес-правила резервирования? Сейчас это не так. Основное отличие от DTO в том, что доменный объект требует присутствия всех четырех составляющих значений[1]. Кроме того, `Date` объявляется как `DateTime`. Это гарантирует, что значение — это правильная дата, а не просто произвольная строка. Если вы все еще не убеждены, можете обратиться к разделу 5.3 и подразделу 7.2.5, касающимся класса `Reservation`.

Почему `Reservation` похож на Value Object[2] (объект-значение)? Потому что это дает ряд преимуществ. Для вашей модели предметной

---

[1] Напомню, что функция ссылочных типов, допускающих нулевое значение, включена. Отсутствие вопросительных знаков в объявлениях свойств указывает на то, что ни одно из них не должно принимать нулевое значение. В отличие от листинга 4.11, в котором все строковые свойства содержат вопросительные знаки, указывающие на возможность принимать это значение.

[2] Value Object (объект-значение) [33] — это неизменяемый объект, представляющий простую сущность, равенство которой не основано на идентичности, то есть два объекта значения равны, когда они имеют одинаковое значение, не обязательно являясь одним и тем же объектом. Типичный пример — класс Money, содержащий данные о валюте и сумме [33].

области нужно использовать объекты-значения [26] — это значительно упростит тестирование [104].

Рассмотрим утверждение из листинга 4.10. В `db` оно будет определено как `expected`. Как `expected` оказалось в `db`? Никак. Это просто объект, который выглядит *точно так же*.

Для сравнения ожидаемых и фактических значений утверждения используют собственные определения равенства объектов, а `Reservation` переопределяет `Equals`. Безопасно реализовать такое *структурное равенство* вы можете, только когда класс неизменяем. Иначе есть риск сравнить два изменяемых объекта и подумать, что они одинаковы, а позже увидеть, что они различаются.

Структурное равенство делает возможными «элегантные» утверждения [104]. Просто создайте в тесте объект, представляющий ожидаемый результат, и сравните его с фактическим.

4.3.5. Fake Object, или фиктивный объект

Последний новый класс — это `FakeDatabase` из листинга 4.13. Как следует из названия, это Fake Object, или фиктивный объект [66], разновидность Test Double[1] (тестового двойника) [66]. Он притворяется базой данных.

Листинг 4.13. Фиктивная база данных. Это часть тестового кода (Restaurant/bc1079a/Restaurant.RestApi.Tests/FakeDatabase.cs)

```
[SuppressMessage(
    "Naming",
    "CA1710:Identifiers should have correct suffix",
    Justification = "The role of the class is a Test Double.")]
public class FakeDatabase :
    Collection<Reservation>, IReservationsRepository
```

[1] Вы можете знать Test Double как макетный объект и заглушку. Как и в случае со словом unit test, нет единого мнения о том, что на самом деле означают эти слова, поэтому я стараюсь их избегать. Как бы то ни было, в книге «Шаблоны тестирования xUnit» [66] описаны четкие определения этих терминов, которыми, к сожалению, никто не пользуется.

```
{
    public Task Create(Reservation reservation)
    {
        Add(reservation);
        return Task.CompletedTask;
    }
}
```

Это обычная in-memory-коллекция, реализующая `IReservationsRepository`. Поскольку это производный от `Collection<Reservation>` интерфейс, он сопровождается разными методами сбора данных, включая `Add`. По этой же причине он работает с `Assert.Contains` в листинге 4.10.

Fake Object [66] — это специфичный для теста объект, который тем не менее ведет себя корректно. Когда вы используете его в качестве замены реальной БД, вы можете работать с ним как с in-memory-базой данных. Он хорошо работает с тестированием на основе состояний [100] (листинг 4.10). На этапе *assert* вы проверяете, соответствует ли фактическое состояние ожидаемому. Этот конкретный тест рассматривает состояние `db`.

4.3.6. Интерфейс Repository

Класс FakeDatabase реализует интерфейс `IReservationsRepository` (листинг 4.14). На столь раннем этапе жизни кодовой базы интерфейс определяет только один метод.

Я решил назвать интерфейс в честь паттерна Repository [33], хотя он едва ли имеет сходство с его исходным описанием. Я сделал так, поскольку большинство разработчиков знакомы с этим названием и понимают, что оно как-то моделирует доступ к данным. Возможно, позже переименую его.

Листинг 4.14. Интерфейс Repository. Это часть модели предметной области (Restaurant/bc1079a/Restaurant.RestApi/IReservationsRepository.cs)

```
public interface IReservationsRepository
{
    Task Create(Reservation reservation);
}
```

4.3.7. Работа с интерфейсом Repository

Из текста на с. 101–107 можно понять, что этот единственный тест привел к созданию нескольких новых типов. В начале жизни кодовой базы это нормально. Там почти нет существующего кода, поэтому даже простой тест, скорее всего, вызовет добавление нового кода.

Чтобы поддерживать взаимодействие, управляемое тестом, вам нужно изменить конструктор `ReservationsController` и метод `Post`. Конструктор должен принимать параметр `IReservationsRepository`, а метод — `ReservationDto`. После внесения этих изменений тест скомпилируется и вы сможете его запустить.

Как и предполагалось, при выполнении тест завершается неудачно.

Чтобы тест прошел успешно, вы должны в репозиторий в методе `Post` добавить объект `Reservation`, как в листинге 4.15.

Чтобы получить внедренный репозиторий и сохранить его как свойство только для чтения для последующего использования, конструктор `ReservationsController` использует `Constructor Injection` (внедрение конструктора) [25]. Это означает, что метод `Post` может его использовать в любом корректно инициализированном экземпляре класса. В нашем случае вызывается `Create` с жестко заданным `Reservation`. Хотя это, очевидно, неправильно, тест выполняется успешно. Это самый простой рабочий код[1] [22].

Листинг 4.15. Сохранение информации о резервировании во внедренном репозитории (Restaurant/bc1079a/Restaurant.RestApi/ReservationsController.cs)

```
[ApiController, Route("[controller]")]
public class ReservationsController
```

[1] Вы можете возразить, что было бы так же просто скопировать значения из dto. Действительно, это будет иметь ту же цикломатическую сложность и такое же количество строк кода, но в рамках предпосылки приоритета трансформации (The Transformation Priority Premise, TPP) [64], я считаю, использование констант проще, чем переменных. Более подробную информацию, касающуюся TPP, вы найдете в подразделе 5.1.1.

```
{
    public ReservationsController(IReservationsRepository repository)
    {
        Repository = repository;
    }

    public IReservationsRepository Repository { get; }
    public async Task Post(ReservationDto dto)
    {
        if (dto is null)
            throw new ArgumentNullException(nameof(dto));

        await Repository
            .Create(
                new Reservation(
                    new DateTime(2023, 11, 24, 19, 0, 0),
                    "juliad@example.net",
                    "Julia Domna",
                    5))
            .ConfigureAwait(false);
    }
}
```

Если вам интересно, что побудило меня использовать Guard Clause [7] против существования `null`-значений, то это было продиктовано статическим правилом анализа кода. Опять же имейте в виду, что вы можете использовать более одного *драйвера* одновременно: разработку через тестирование *и* анализаторы или линтеры. Есть много инструментов, которые могут управлять созданием кода. Например, я использовал инструмент *добавления проверки на null* в Visual Studio для добавления защиты.

Теперь код листинга 4.15 успешно проходит тест из листинга 4.10, но другой тест завершается неудачно!

4.3.8. Настройка зависимостей

В то время как новый тест проходит успешно, граничный тест из листинга 4.7 завершается неудачей, так как в `ReservationsController` больше нет конструктора без параметров. Платформе ASP.NET нужна помощь в создании экземпляров класса, особенно потому, что ни один

из классов в продакшен-коде не реализует требуемый интерфейс `IReservationsRepository`.

Самый простой способ пройти все тесты успешно — это добавить реализацию интерфейса `Null Object` [118]. В листинге 4.16 показан временный класс, вложенный в класс `Startup`. Это реализация `IReservationsRepository`, которая ничего не делает.

Листинг 4.16. Реализация Null Object, временного вложенного класса (Restaurant/bc1079a/Restaurant.RestApi/Startup.cs)

```
private class NullRepository : IReservationsRepository
{
    public Task Create(Reservation reservation)
    {
        return Task.CompletedTask;
    }
}
```

Регистрация этого класса во встроенном в ASP.NET контейнере внедрения зависимостей [25] решит проблему. Как это сделать, показано в листинге 4.17. Поскольку у `NullRepository` нет состояния, вы можете зарегистрировать один объект с временем жизни `Singleton` [25]. Это значит, что один и тот же объект будет совместно использоваться всеми потоками на протяжении всей жизни веб-сервиса.

Листинг 4.17. Регистрация NullRepository во встроенном контейнере внедрения зависимостей ASP.NET (Restaurant/bc1079a/Restaurant.RestApi/Startup.cs)

```
public static void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddSingleton<IReservationsRepository>(
        new NullRepository());
}
```

Теперь, когда все тесты проходят успешно, сделайте коммит изменений в Git и рассмотрите их пуск через конвейер развертывания.

4.4. ЗАВЕРШЕНИЕ СРЕЗА

При рассмотрении вертикального среза на рис. 4.5 кажется, что чего-то не хватает. Чтобы сохранить информацию о резервировании в постоянном хранилище, нужна правильная реализация `IReservationsRepository`. Как только вы это сделаете, вы завершите срез.

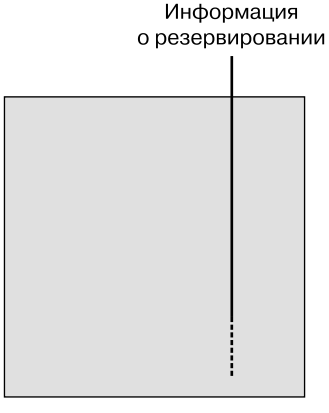


Рис. 4.5. Прогресс на данный момент. Сравните с планом на рис. 4.4

«Подождите, — скажете вы, — это совсем не работает! Это просто сохранит жестко запрограммированную информацию о резервировании! А как насчет проверки вводных данных, ведения лога или безопасности?»

Мы обязательно вернемся к этому вопросу. Прямо сейчас меня устроит, если мой код вызовет постоянное изменение состояния, даже если это будет жестко запрограммированная информация о резервировании. Это по-прежнему будет означать, что внешнее событие (HTTP POST) может изменить состояние приложения.

4.4.1. Схема

Как сохранить бронирование? В реляционной базе данных? Графовой [89]? Документной?

Если вы хотите следовать рекомендациям GOOS, или *«Развития объектно-ориентированного ПО, управляемого тестами»* [36], выберите технологию, которая лучше всего поддержит разработку через тестирование. Желательно ту, что вы можете разместить в своих автоматических тестах. Для этого подойдет документная база данных.

Но, несмотря на это, я все же выберу реляционную БД, а именно SQL Server. Я делаю это в образовательных целях. Во-первых, GOOS [36] — уже отличный ресурс для того, чтобы научиться принципиальной outside-in-разработке через тестирование. Во-вторых, в действительности реляционные базы данных очень популярны. Наличие таковых часто даже не обсуждается. Ваша организация может иметь соглашение о поддержке с определенным провайдером. Ваша операционная группа может выбрать определенную систему, так как они знают, как ее обслуживать и как работать с ней. Наконец, вашим коллегам может быть удобнее работать с определенной базой.

Несмотря на развитие *NoSQL*, реляционные базы данных остаются неотъемлемой частью разработки корпоративного ПО. Надеюсь, что мой пример с использованием одной из таких баз сделает эту книгу более полезной. Я выбрал SQL Server, так как это идиоматическая часть стандартного стека Microsoft, но методы, которые вам придется применять, не сильно изменятся, если вы решите выбрать другую базу.

В листинге 4.18 приведен пример исходной структуры таблицы `Reservations`.

Листинг 4.18. Схема базы данных для таблицы Reservations (Restaurant/c82d82c/Restaurant.RestApi/RestaurantDbSchema.sql)

```
CREATE TABLE [dbo].[Reservations] (
    [Id] INT NOT NULL IDENTITY,
    [At] DATETIME2 NOT NULL,
    [Name] NVARCHAR (50) NOT NULL,
    [Email] NVARCHAR (50) NOT NULL,
    [Quantity] INT NOT NULL
    PRIMARY KEY CLUSTERED ([Id] ASC)
)
```

Мне нравится структура БД на языке SQL, потому что это родной язык базы данных. Вместо этого можно использовать объектно-реля-

ционный преобразователь или предметно-ориентированный язык, это тоже будет нормально. Основная идея в том, что вы делаете коммит структуры базы данных в том же репозитории Git, в котором хранится остальной исходный код.

> Сделайте коммит схемы базы данных в репозиторий Git.

4.4.2. Репозиторий SQL

Теперь, зная, как выглядит структура БД, вы можете реализовать интерфейс `IReservationsRepository` для этой базы. В листинге 4.19 я привел пример своей реализации. Как вы понимаете, я не фанат объектно-реляционного преобразования (object-relational mapper, ORM).

Вы можете возразить, что использование фундаментального API ADO.NET слишком сложно по сравнению с, например, Entity Framework, но помните, что вам не следует оптимизировать скорость написания. При оптимизации для удобочитаемости вы все равно можете утверждать, что использование объектно-реляционного преобразователя было бы более читабельным.

Если вы все-таки захотите использовать объектно-реляционное отображение, сделайте это. Это не так важно. Важно, чтобы ваша модель предметной области [33] не была загрязнена деталями реализации[1].

Что мне нравится в реализации из листинга 4.19, так это то, что в ней содержатся простые инварианты. Это потокобезопасный объект без состояния: вы можете создать один его экземпляр и повторно использовать его в течение всего срока службы вашего приложения.

«Но, Марк, — возразите вы, — вы снова пытаетесь нас обмануть! Вы не проводили тестирование этого класса».

---

[1] Это принцип инверсии зависимостей: абстракции не должны зависеть от деталей — детали должны зависеть от абстракций [60]. Абстракция в этом контексте — доменная модель, то есть резервирование.

Листинг 4.19. Реализация интерфейса репозитория в SQL Server (Restaurant/c82d82c/Restaurant.RestApi/SqlReservationsRepository.cs)

```
public class SqlReservationsRepository : IReservationsRepository
{
    public SqlReservationsRepository(string connectionString)
    {
        ConnectionString = connectionString;
    }
    public string ConnectionString { get; }

    public async Task Create(Reservation reservation)
    {
        if (reservation is null)
            throw new ArgumentNullException(nameof(reservation));
        using var conn = new SqlConnection(ConnectionString);
        using var cmd = new SqlCommand(createReservationSql, conn);
        cmd.Parameters.Add(new SqlParameter("@At", reservation.At));
        cmd.Parameters.Add(new SqlParameter("@Name", reservation.Name));
        cmd.Parameters.Add(new SqlParameter("@Email",
            reservation.Email));
        cmd.Parameters.Add(
            new SqlParameter("@Quantity", reservation.Quantity));
        await conn.OpenAsync().ConfigureAwait(false);
        await cmd.ExecuteNonQueryAsync().ConfigureAwait(false);
    }
    private const string createReservationSql = @"
        INSERT INTO
            [dbo].[Reservations] ([At], [Name], [Email], [Quantity])
        VALUES (@At, @Name, @Email, @Quantity)";
}
```

Я этого не сделал, так как считаю `SqlReservationsRepository` «скромным объектом» [66]. Это реализация, которую трудно модульно тестировать, так как она зависит от подсистемы, которую нельзя легко автоматизировать. Вместо этого вы истощаете объект логики ветвления и других видов поведения, которые обычно вызывают дефекты.

Единственная ветвь в `SqlReservationsRepository` — это защита от значений `null`, созданная в Visual Studio на основе статического анализа кода.

При этом в разделе 12.2 вы увидите, как добавлять автоматические тесты, в которых задействована база данных.

4.4.3. Конфигурация базы данных

Теперь, когда у вас есть правильная реализация `IReservationsRepository`, вы должны сообщить об этом ASP.NET. В листинге 4.20 приведен пример изменений, которые нужно внести в класс `Startup`.

Листинг 4.20. Фрагменты файла Startup, с помощью которых можно настроить приложение для работы с SQL Server (Restaurant/c82d82c/Restaurant.RestApi/Startup.cs)

```
public IConfiguration Configuration { get; }

public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    var connStr = Configuration.GetConnectionString("Restaurant");
    services.AddSingleton<IReservationsRepository>(
        new SqlReservationsRepository(connStr));
}
```

Вы вызываете `AddSingleton` с новым классом `SqlReservationsRepository` вместо `NullRepository` из листинга 4.16, который можно удалить.

Вы не можете создать экземпляр `SqlReservationsRepository` без предоставления строки подключения, поэтому должны получить ее из конфигурации ASP.NET.

При добавлении конструктора в `Startup`, как в листинге 4.20, платформа автоматически предоставляет экземпляр `IConfiguration`.

Вам придется настроить приложение с правильной строкой подключения. Среди многих доступных опций вы можете выбрать файл конфигурации. В листинге 4.21 вы увидите, что я закоммитил в Git. Даже если вашим коллегам будет полезно коммитить структуру необходимой конфигурации, не включайте фактические строки подключения.

Они будут различаться в зависимости от среды разработки и могут содержать идентификационные данные, которых не должно быть в вашей системе контроля версий.

Листинг 4.21. Структура конфигурации строки подключения. Это то, что вы должны закоммитить в Git. Избегайте наличия и ввода идентификационных данных (Restaurant/c82d82c/Restaurant.RestApi/appsettings.json)

```
{
    "ConnectionStrings": {
        "Restaurant": ""
    }
}
```

Приложение должно работать после помещения этой строки подключения в файл конфигурации.

4.4.4. Дымовой тест, или smoke-тестирование

Как узнать, что программа работает, если мы не добавили автоматизированный системный тест?

Хотя предпочтительнее пользоваться автоматическими тестами, не стоит забывать и о ручном тестировании. Время от времени включайте систему и смотрите, не «загорится» ли она, — это и есть smoke-тест, или дымовое тестирование.

Если вы поместите правильную строку подключения в файл конфигурации и запустите систему на своем компьютере для разработки, вы можете попытаться отправить на нее информацию о резервировании через POST. Есть много инструментов для взаимодействия с HTTP API. Разработчики .NET обычно используют инструменты с графическим интерфейсом, такие как Postman или Fiddler, но сделайте себе одолжение и изучите что-то, что легче автоматизировать. Я часто пользуюсь cURL. Рассмотрим следующий пример (для компактности я разбил его на несколько строк):

```
$ curl -v http://localhost:53568/reservations
    -H "Content-Type: application/json"
```

```
-d "{ \"at\": \"2022-10-21 19:00\",
    \"email\": \"caravan@example.com\",
    \"name\": \"Cara van Palace\",
    \"quantity\": 3 }"
```

Здесь информация о резервировании в формате JSON отправляется на соответствующий URL-адрес. Если вы посмотрите в базу, которую настроили для использования приложения, то увидите строку с данными о резервировании... для *Julia Domna*!

Напомню, что система по-прежнему сохраняет жестко запрограммированные данные, но теперь вы знаете, что при внесении изменений обязательно что-то произойдет.

4.4.5. Граничный тест с фиктивной базой данных

Теперь осталась единственная проблема: граничный тест из листинга 4.7 не проходит успешно. Класс `Startup` настраивает службу `SqlReservationsRepository` со строкой подключения, но в тестовом контексте такой строки нет, как и базы данных.

Можно автоматизировать настройку и удаление базы данных для автоматизированного тестирования, но это слишком громоздко и сильно замедлит процесс тестирования. Может быть, я сделаю это позже[1], но не сейчас.

Вместо этого вы можете запустить граничный тест для `FakeDatabase`, как в листинге 4.13. Для этого нужно изменить поведение теста `WebApplicationFactory`, переопределив его метод `ConfigureWebHost`, как в листинге 4.22.

Код в блоке `ConfigureServices` запускается после выполнения метода `ConfigureServices` класса `Startup`. Он находит все службы, реализующие интерфейс `IReservationsRepository` (есть только одна), и удаляет их. Затем он добавляет экземпляр `FakeDatabase` в качестве замены.

---

[1] На самом деле в разделе 12.2.

Листинг 4.22. Как заменить реальную зависимость фиктивной для тестирования (Restaurant/c82d82c/Restaurant.RestApi.Tests/RestaurantApiFactory.cs)

```
public class RestaurantApiFactory : WebApplicationFactory<Startup>
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        if (builder is null)
            throw new ArgumentNullException(nameof(builder));

        builder.ConfigureServices(services =>
        {
            services.RemoveAll<IReservationsRepository>();
            services.AddSingleton<IReservationsRepository>(
                new FakeDatabase());
        });
    }
}
```

В своем модульном тесте используйте новый класс `RestaurantApiFactory`, но это всего лишь изменение одной строки во вспомогательном методе `PostReservation`. Сравните код в листинге 4.23 и листинге 4.8.

Листинг 4.23. Вспомогательный метод теста с обновленной фабрикой веб-приложений. Выделенная строка, которая инициализирует фабрику, изменилась по сравнению с листингом 4.8 (Restaurant/c82d82c/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[SuppressMessage(
    "Usage",
    "CA2234:Pass system uri objects instead of strings",
    Justification = "URL isn't passed as variable, but as literal.")]
private async Task<HttpResponseMessage> PostReservation(
    object reservation)
{
    using var factory = new RestaurantApiFactory();
    var client = factory.CreateClient();

    string json = JsonSerializer.Serialize(reservation);
    using var content = new StringContent(json);
    content.Headers.ContentType.MediaType = "application/json";
    return await client.PostAsync("reservations", content);
}
```

И снова все тесты проходят. Отправьте коммит изменений в Git и рассмотрите их пуск через конвейер развертывания. После того как изменения будут включены в продакшен, выполните в рабочей системе еще один ручной smoke-тест.

4.5. ЗАКЛЮЧЕНИЕ

Создание тонкого вертикального среза — эффективный способ продемонстрировать, что программа действительно может работать. В сочетании с непрерывной доставкой [49] вы можете быстро сделать релиз своего рабочего ПО.

Возможно, вы подумали, что первый вертикальный срез настолько «тонкий», что в нем нет смысла. В примере показано, как сохранить информацию о резервировании в базе, но сохраняемые значения не являются предоставленными системе. Это как-то повышает ценность кода?

Вряд ли, но это поможет установить рабочую систему и конвейер развертывания [49]. Теперь вы можете внести небольшие изменения, которые будут приближать вас к цели получения полезной системы. Другие заинтересованные стороны всегда смогут определить, когда система станет полезной, ваша же задача — позволить им сделать эту оценку. Выполняйте развертывание как можно чаще, и пусть другие заинтересованные лица сообщат вам, когда работа будет считаться завершенной.

5 ИНКАПСУЛЯЦИЯ

Вы когда-нибудь покупали что-нибудь дорогостоящее, например дом, участок земли, компанию или автомобиль? Если да, то, скорее всего, вы подписывали контракт.

Контракт содержит список прав и обязанностей обеих сторон. Продавец обязуется передать имущество, а покупатель — оплатить его в установленный срок. Продавец может дать некоторые гарантии относительно состояния имущества, а покупатель — не возлагать на продавца ответственность за ущерб после завершения сделки и т. д.

Контракт обеспечивает тот уровень доверия, которого в противном случае не было бы. С чего бы вам доверять незнакомому человеку? Это рискованно. Но вы можете заключить с ним контракт. Это и есть *инкапсуляция*. Как можно принимать на веру то, что объект будет вести себя разумно? Заставляя его заключить контракт.

5.1. СОХРАНЕНИЕ ДАННЫХ

Глава 4 изучена, но вопросы еще остались. В листинге 4.15 показано, как метод `Post` сохраняет жестко закодированные сведения о резервировании, игнорируя при этом полученные данные.

Это баг, для исправления которого нужно добавить код. С этого мы и начнем изучение инкапсуляции. Так мы сможем убить сразу двух зайцев, поэтому давайте первым делом расправимся с этой задачей.

5.1.1. Предпосылки приоритета трансформации (TPP)

По возможности используйте драйвер. Жестко закодированные значения в коде листинга 4.15 управлялись одним тест-кейсом. Как улучшить ситуацию? Возможность просто исправить код выглядит очень заманчиво. Во время проведения обучения мне постоянно приходится напоминать разработчикам немного притормозить. Пишите продакшен-код как ответы на такие драйверы, как тесты или анализаторы. Движение вперед небольшими шагами снижает риск ошибок.

При редактировании вы *преобразуете* код из одного рабочего состояния в другое. Это не происходит атомарно. При модификации код может не компилироваться. Старайтесь максимально сократить время, когда код нерабочий (рис. 5.1), — это уменьшит количество действий, которые нужно отследить.

Рис. 5.1. Редактирование кода — это процесс перехода из одного рабочего состояния в другое. Старайтесь, чтобы период нахождения кода в переходном состоянии был как можно короче

В 2013 году Роберт Мартин опубликовал следующие рекомендации для улучшения кода [64]. Это было задумано только как предварительное предложение, но я нахожу их очень полезными:

- (**{} → nil**) нет кода → код, использующий значения `nil`;
- (**nil → constant**);
- (**constant → constant+**) простая константа → более сложная константа;

- **(constant → scalar)** замена константы переменной или аргументом;
- **(statement → statements)** добавление дополнительных безусловных операторов;
- **(unconditional → if)** разделение пути выполнения;
- **(scalar → array)**;
- **(array → container)**;
- **(statement → recursion)**;
- **(if → while)**;
- **(expression → function)** замена выражения функцией или алгоритмом;
- **(variable → assignment)** замена значения переменной.

Список построен по принципу «от простого к сложному».

Не переживайте, если чего-то не поняли. Почти все рекомендации в этой книге — лишь основа для размышлений, а не жесткие правила. Самое важное — двигаться небольшими шагами, например используя жестко закодированную константу вместо `null`[1] или превращая единичное значение в массив.

Сейчас метод `Post` сохраняет *константу*, но он должен сохранять данные из `dto`: набор *скалярных значений*. Это *константно-скалярное* преобразование (или их набор).

Суть предпосылки приоритета трансформации в том, что мы должны стремиться вносить изменения в наш код, используя небольшие преобразования из списка.

Поскольку мы определили изменение, к которому стремимся, как одно из гарантированных изменений, давайте осуществим его.

---

[1] В своей статье [64] Роберт Мартин вызывает неопределенное значение nil, но, предполагаю, что он имеет в виду null. В некоторых языках (например, Ruby) null именуется как nil.

5.1.2. Параметризованные тесты

В основе предпосылки приоритета трансформации заложена идея о том, что после определения трансформации, к которой нужно стремиться, вы должны написать тест, мотивирующий это изменение.

Можно написать новый тестовый метод, но это будет дубликат кода из листинга 4.10, только с некоторыми другими значениями свойств для `dto`. Вместо этого сделайте существующий тест параметризованным [66].

Листинг 5.1. Параметризованный тест на примере резервирования. По сравнению с листингом 4.10 новым является только выделенный тест-кейс (Restaurant/4617450/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Theory]
[InlineData(
    "2023-11-24 19:00", "juliad@example.net", "Julia Domna", 5)]
[InlineData("2024-02-13 18:15", "x@example.com", "Xenia Ng", 9)]
public async Task PostValidReservationWhenDatabaseIsEmpty(
    string at,
    string email,
    string name,
    int quantity)
{
    var db = new FakeDatabase();
    var sut = new ReservationsController(db);

    var dto = new ReservationDto
    {
        At = at,
        Email = email,
        Name = name,
        Quantity = quantity
    };
    await sut.Post(dto);

    var expected = new Reservation(
        DateTime.Parse(dto.At, CultureInfo.InvariantCulture),
        dto.Email,
        dto.Name,
        dto.Quantity);
    Assert.Contains(expected, db);
}
```

Сравните листинги 4.10 и 5.1. Вместо атрибута `[Fact]` для указания параметризованного теста используется `[Theory]`[1], еще добавлены два атрибута `[InlineData]`, предоставляющие данные. Обратите внимание, что верхний атрибут `[InlineData]` предоставляет те же тестовые значения, что и в листинге 4.10, а второй содержит новый тест-кейс.

Но сейчас вас должно беспокоить, что этап утверждения теста теперь дублирует то, что, по сути, было бы окончательной версией кода. А это не есть хорошо. Вы не должны доверять себе написание продакшен-кода, не просчитав все риски, но это работает, только если представления различаются. Это не тот случай.

Как мы знаем, лучшее — враг хорошего. Хотя это изменение создает проблему в коде теста, его цель — показать, что метод `Post` не работает. И действительно, когда вы запускаете набор тестов, новый тест-кейс дает сбой.

5.1.3. Копирование данных dto в модель предметной области

В листинге 5.2 показан пример простейшего преобразования, которое вы можете применить к методу `Post`, чтобы все тесты прошли успешно.

Листинг 5.2. Метод Post теперь сохраняет данные dto (Restaurant/4617450/Restaurant.RestApi/ReservationsController.cs)

```
public async Task Post(ReservationDto dto)
{
```

[1] Это API xUnit.net для параметризованных тестов. Другие фреймворки предоставляют эту функцию похожими или не очень похожими способами. Некоторые фреймворки модульного тестирования вообще не поддерживают это. На мой взгляд, это достаточная причина, чтобы найти другой фреймворк. Возможность писать параметризованные тесты — одна из наиболее важных особенностей фреймворка модульного тестирования.

```
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));

    var r = new Reservation(
        DateTime.Parse(dto.At!, CultureInfo.InvariantCulture),
        dto.Email!,
        dto.Name!,
        dto.Quantity);
    await Repository.Create(r).ConfigureAwait(false);
}
```

Этот код считается улучшением по сравнению с тем, что в листинге 4.15, но все еще есть проблемы, требующие решения. Прямо сейчас сдержитесь и не делайте никаких дальнейших улучшений. Добавив тест-кейс из листинга 5.1, вы внесли небольшие изменения. Хотя код все еще не идеален, он *улучшен*. Все тесты проходят успешно. Сделайте коммит изменений в Git и рассмотрите их пуск через конвейер развертывания.

Восклицательные знаки после `dto.At`, `dto.Email` и `dto.Name` — это лишь некоторые из оставшихся недостатков.

В нашей кодовой базе используется функционал *ссылочных типов C#, допускающих значение NULL*, и большинство свойств `dto` объявлены как допускающие значение `NULL`. Без восклицательного знака компилятор будет возмущаться, что код обращается к значению, допускающему значение `NULL`, без проверки. Символ `!` блокирует ошибки, и код компилируется.

Это ужасный способ. Хоть код и компилируется, он может легко вызвать исключение `NullReferenceException`. Обмен ошибки времени компиляции на исключение времени выполнения — плохая затея. Но мы должны решить эту проблему.

Еще одно потенциальное исключение во время выполнения заключается в том, что нет никакой гарантии, что вызов метода `DateTime.Parse` будет успешным. И эту проблему нам тоже нужно решить.

5.2. ВАЛИДАЦИЯ

Что произойдет с кодом в листинге 5.2, если клиент отправит документ JSON без свойства `at`?

Вы наверняка решите, что `Post` выдаст исключение `NullReferenceException`, но на самом деле `DateTime.Parse` вместо этого выдает исключение `ArgumentNullException`. По крайней мере, этот метод выполняет проверку ввода. Вы должны сделать то же самое.

Почему ArgumentNullException лучше, чем NullReferenceException?

Так ли важно, какое исключение генерирует метод? В конце концов, если вы не обработаете его, ваша программа просто не будет работать.

Типы исключений — это очень важно. Зная, что вы можете обработать исключение определенного типа, можно написать блок `try/catch`. Проблема во всех исключениях, с которыми вы не можете справиться.

Обычно `NullReferenceException` возникает, когда нет требуемого объекта (`null`). Если объект важен, но недоступен, вы с этим ничего не сделаете. Это актуально как для `NullReferenceException`, так и для `ArgumentNullException`. Так зачем проверять значение `null` только для того, чтобы создать исключение?

Разница в том, что в `NullReferenceException` нет никакой полезной информации. Оно указывает, что какой-то объект был `null`, но неизвестно, какой именно. Исключение же `ArgumentNullException` как раз содержит информацию о том, какой аргумент был `null`.

Столкнувшись с сообщением об исключении в логе или в отчете об ошибке, что бы вы хотели увидеть? `NullReferenceException`, не содержащее какой-либо информации, или `ArgumentNullException` с именем аргумента, который был `null`?

Думаю, второе.

Платформа ASP.NET преобразует необработанное исключение в ответ `500 Internal Server Error`. Но это не то, что нам нужно.

5.2.1. Невалидные данные

При вводе невалидных данных HTTP API должен вернуть код ответа `400 Bad Request` [2]. Но так не происходит. Добавьте тест, который воспроизводит проблему.

В листинге 5.3 приведен пример проверки в случае отсутствия даты и времени резервирования. Почему я написал это как `[Theory]` лишь с одним тест-кейсом? Почему не `[Fact]`?

Признаю: я немного вас обманул. И снова я внес немного *творчества* в разработку ПО. Мой выбор основан на «зыбучих песках индивидуального опыта» [4] — я знаю, что скоро добавлю больше тест-кейсов, поэтому мне легче начать с `[Theory]`.

Листинг 5.3. Проверка отправки dto резервирования с отсутствующим значением at (Restaurant/9e49134/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Theory]
[InlineData(null, "j@example.net", "Jay Xerxes", 1)]
public async Task PostInvalidReservation(
    string at,
    string email,
    string name,
    int quantity)
{
    var response =
        await PostReservation(new { at, email, name, quantity });
    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}
```

Тест провален, поскольку код ответа — `500 Internal Server Error`.

Но вы можете легко пройти тест с кодом из листинга 5.4. Основное его отличие от кода выше — это добавление Null Guard, или проверки на `null`.

Листинг 5.4. Проверка на null свойства At (Restaurant/9e49134/Restaurant.RestApi/ReservationsController.cs)

```
public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));
    if (dto.At is null)
        return new BadRequestResult();

    var r = new Reservation(
        DateTime.Parse(dto.At, CultureInfo.InvariantCulture),
        dto.Email!,
        dto.Name!,
        dto.Quantity);
    await Repository.Create(r).ConfigureAwait(false);

    return new NoContentResult();
}
```

Компилятор C# достаточно умен, чтобы обнаружить Guard Clause. Это значит, что вы можете удалить восклицательный знак после `dto.At`.

Вы можете добавить еще один тест-кейс, в котором нет свойства `email`, но давайте заглянем вперед еще на один шаг. Листинг 5.5 содержит два новых тест-кейса.

Листинг 5.5. Дополнительные тест-кейсы с вводом невалидных данных резервирования (Restaurant/3fac4a3/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Theory]
[InlineData(null, "j@example.net", "Jay Xerxes", 1)]
[InlineData("not a date", "w@example.edu", "Wk Hd", 8)]
[InlineData("2023-11-30 20:01", null, "Thora", 19)]
public async Task PostInvalidReservation(
    string at,
    string email,
    string name,
    int quantity)
{
    var response =
        await PostReservation(new { at, email, name, quantity });
    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}
```

Нижний атрибут `[InlineData]` содержит тестовый случай с отсутствующим свойством `email`, а средний предоставляет значение `at`, которое не является датой и временем.

Листинг 5.6. Проверка на ввод различных невалидных входных данных (Restaurant/3fac4a3/Restaurant.RestApi/ReservationsController.cs)

```
public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));
    if (dto.At is null)
        return new BadRequestResult();
    if (!DateTime.TryParse(dto.At, out var d))
        return new BadRequestResult();
    if (dto.Email is null)
        return new BadRequestResult();

    var r = new Reservation(d, dto.Email, dto.Name!, dto.Quantity);
    await Repository.Create(r).ConfigureAwait(false);

    return new NoContentResult();
}
```

Код листинга 5.6 проходит все тесты успешно. Обратите внимание, что я мог бы удалить еще один восклицательный знак, добавив проверку электронной почты.

5.2.2. Цикл «красный, зеленый, рефакторинг»

Рассмотрим пример из листинга 5.6. По сравнению с 4.15 он уже гораздо сложнее. Можете ли вы его упростить? Это важный вопрос, который вы должны задавать после каждой итерации теста. Это часть цикла «красный, зеленый, рефакторинг» (Red Green Refactor) [9].

- **Red («красный»).** Написание теста, дающего сбой. Обычно отображается красным.
- **Green («зеленый»).** Минимальные возможные изменения, которые приведут к успешному прохождению теста. Тестировщики часто отображают пройденные тесты зеленым.

- **Refactor («рефакторинг»).** Улучшение кода без изменения его поведения.

Пройдя все три этапа, вы начнете с нового неудачного теста (рис. 5.2).

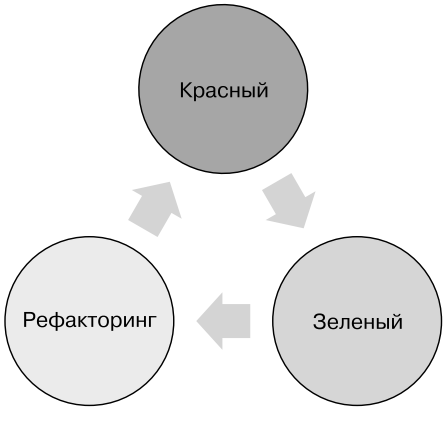


Рис. 5.2. Цикл «красный, зеленый, рефакторинг»

До сих пор в примере вы наблюдали только процессы «красный, зеленый». Пришло время добавить третью фазу.

Наука разработки через тестирование (TDD)

Процесс «красный, зеленый, рефакторинг» — одна из самых научных методологий разработки ПО.

В научном методе вы сначала формируете гипотезу в форме предсказания фальсифицируемого результата. Затем проводите эксперимент и измеряете результат. А после сравниваете фактический результат с прогнозируемым.

Ничего не напоминает?

Это похоже на паттерн AAA [9], где этап *act* — это эксперимент, а *assert* — сравнение ожидаемого и фактического результатов.

«Красная» и «зеленая» фазы в цикле «красный, зеленый, рефакторинг» — это, по сути, небольшие научные эксперименты.

Гипотеза «красной» фазы такова: написанный тест должен дать сбой. Это измеримый эксперимент, который можно провести. У него есть количественный результат: он пройдет либо удачно, либо нет.

Применив «красный, зеленый, рефакторинг» как последовательный процесс, вы будете удивлены, как часто этот этап завершается успешным тестом. Вспомните, как легко ваш мозг делает поспешные выводы [51] или вы ненароком можете написать тавтологические утверждения [105]. Все это может произойти, но вы не обнаружите результат, если не проведете эксперимент.

Точно так же и «зеленая» фаза — это уже готовая гипотеза: написанный тест будет успешным. Опять же эксперимент здесь — это проведение теста, дающего количественный результат.

Если вы хотите двигаться в сторону программной инженерии и если верите в связь между наукой и инженерией, я не могу придумать ничего более подходящего, чем разработка через тестирование.

Проводя рефакторинг, вы рассматриваете код, написанный на этапе «зеленый». Можете ли вы улучшить его? Если да, то это будет рефакторинг.

> *«Рефакторинг — это процесс изменения программной системы так, чтобы не менялось внешнее поведение кода, но улучшалась его внутренняя структура» [34].*

Откуда вы знаете, что внешнее поведение не меняется? Трудно доказать универсальную гипотезу, но опровергнуть ее легко. Если хотя бы один автоматизированный тест не прошел после изменения, вы бы знали, что что-то сломали. Итак, вы как минимум знаете, что при изменении структуры кода все тесты все равно должны завершиться успешно.

Можно ли улучшить код листинга 4.15, сохранив при этом все тесты? Да, получается, что проверка на `null dto.At` избыточна. Ниже приведен пример упрощенного метода `Post` (листинг 5.7).

Листинг 5.7. Нет необходимости проводить проверку на null свойства At — DateTime.TryParse уже делает это (Restaurant/b789ef1/Restaurant.RestApi/ReservationsController.cs)

```
public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));
    if (!DateTime.TryParse(dto.At, out var d))
        return new BadRequestResult();
    if (dto.Email is null)
        return new BadRequestResult();

    var r = new Reservation(d, dto.Email, dto.Name!, dto.Quantity);
    await Repository.Create(r).ConfigureAwait(false);

    return new NoContentResult();
}
```

Почему это все еще работает? Потому что `DateTime.TryParse` уже проверяет значение `null`, и, если входное значение равно `null`, возвращаемое вернет `false`.

Как вы могли это знать? Я не уверен, что смогу дать ответ, который приведет к воспроизводимым результатам. Я подумал об этом рефакторинге, так как знал поведение `DateTime.TryParse`. Это еще один пример программирования, основанного на индивидуальном опыте [4], — *искусство* разработки ПО.

5.2.3. Натуральные числа

Инкапсуляция — это больше, чем просто проверка нулевого значения. Это контракт, описывающий допустимые взаимодействия между объектами и вызывающими объектами. Один из способов определить валидность — указать, что считается *невалидным*, — так все остальное будет считаться валидным.

Запрещая нулевые ссылки, вы неявно разрешаете все ненулевые объекты. Код в листинге 5.7 уже делает это для `dto.At`. Мало того что значение `null` запрещено, но еще и строка должна представлять валидное значение даты и времени.

Дизайн по контракту

Инкапсуляция — это представление о том, что вы должны иметь возможность взаимодействовать с объектом, не зная подробностей его реализации. Это позволяет достичь как минимум двух целей:

- изменить реализацию (то есть провести *рефакторинг*);
- думать об объекте абстрактно.

Второй момент важен, когда дело доходит до разработки ПО. Вспомним о проблеме когнитивных ограничений из главы 3: ваш мозг может отслеживать только семь действий. Инкапсуляция позволяет «заменить» многие детали реализации объекта более простым контрактом.

Вернемся к определению абстракции Роберта Мартина: *«Абстракция — это устранение неважного и усиление существенного»* [60].

Неотъемлемая часть объекта — его контракт. Он обычно гораздо проще, чем низлежащая реализация, поэтому лучше укладывается в вашем мозге.

Идея сделать контракты явной частью объектно-ориентированного программирования (ООП) тесно связана с Бертраном Мейером и языком Eiffel, где контракты — это явная часть языка [67].

Хотя ни один из современных языков не сделал контракты более явными, чем Eiffel, вы все равно можете проектировать с учетом контрактов. Например, Guard Clause [7] может обеспечить выполнение контракта, отклонив неверный ввод.

Проектируйте явно с упором на то, что является и не является допустимым вводом и какие гарантии вы можете дать в отношении вывода.

Что можно сказать о других частях резервирования? Используя статическую систему типов C#, в классе `ReservationDto`, уже показанном в листинге 4.11 (из-за отсутствия символа `?`), мы объявляем, что `Quantity` не может принимать значение `null`. Но будет ли при этом любое целое число, например 2, 0 или –3, валидным?

Число 2 кажется разумным, в отличие от –3. А что насчет 0? Зачем делать бронирование без людей? Логично, что значение количества человек при резервировании будет натуральным числом. По моему опыту, так часто происходит, когда вы развиваете модель предметной области [33; 26]. Модель — это попытка описать реальность[1], а в реальном мире натуральных чисел предостаточно.

Ниже представлен пример того же метода тестирования, что и в листинге 5.5, но включая два новых тест-кейса с невалидными значениями количества (листинг 5.8).

Листинг 5.8. Дополнительные тест-кейсы с невалидными значениями количества. Выделенные тест-кейсы являются новыми, если сравнивать с листингом 5.5 (Restaurant/a6c4ead/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Theory]
[InlineData(null, "j@example.net", "Jay Xerxes", 1)]
[InlineData("not a date", "w@example.edu", "Wk Hd", 8)]
[InlineData("2023-11-30 20:01", null, "Thora", 19)]
[InlineData("2022-01-02 12:10", "3@example.org", "3 Beard", 0)]
[InlineData("2045-12-31 11:45", "git@example.com", "Gil Tan", -1)]
public async Task PostInvalidReservation(
    string at,
    string email,
    string name,
    int quantity)
{
    var response =
        await PostReservation(new { at, email, name, quantity });
    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}
```

[1] Даже когда реальный мир — это всего лишь бизнес-процесс.

Эти новые тест-кейсы привели к повторной проверке метода `Post` из листинга 5.9. Новая Guard Clause [7] принимает только натуральные числа.

Листинг 5.9. Метод Post теперь также предотвращает недопустимое значение количества (Restaurant/a6c4ead/Restaurant.RestApi/ReservationsController.cs)

```
public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));
    if (!DateTime.TryParse(dto.At, out var d))
        return new BadRequestResult();
    if (dto.Email is null)
        return new BadRequestResult();
    if (dto.Quantity < 1)
        return new BadRequestResult();

    var r = new Reservation(d, dto.Email, dto.Name!, dto.Quantity);
    await Repository.Create(r).ConfigureAwait(false);

    return new NoContentResult();
}
```

У большинства языков программирования встроенные типы данных. Есть несколько типов целочисленных данных: 8-, 16-битные целые числа и т. д.

Но обычные целые числа могут быть как отрицательными, так и положительными. Обойти эту проблему можно, используя целые числа без знака, но здесь это не сработает, так как целое число без знака все еще может быть 0.

Чтобы исключить резервирование с 0 человек, вам все равно понадобится пункт Guard Clause.

Код в листинге 5.9 компилируется, и все тесты проходят успешно. Сделайте коммит изменений в Git и рассмотрите их пуск через конвейер развертывания.

5.2.4. Закон Постела (принцип надежности)

Итак, давайте повторим. Что такое валидное резервирование? Дата должна быть правильной, а количество должно быть натуральным числом. Еще нам нужно, чтобы `Email` не был пустым, но так ли это?

Разве нет необходимости вводить *реальный* адрес электронной почты? А что насчет имени?

Адреса электронной почты трудно проверить [41]. И даже если бы у вас была полная реализация спецификации SMTP, что бы это вам дало?

Пользователи могут легко дать вам неверный, но соответствующий спецификации адрес электронной почты. Единственный способ действительно проверить его — отправить письмо с подтверждением (например, с прикрепленной ссылкой, по которой пользователь может перейти для проверки). Это длительный асинхронный процесс, поэтому, даже если вы захотите так сделать, вы не сможете сделать это как вызов блокирующего метода.

По правде говоря, нет особо важной причины подтверждать почту, кроме проверки того, что она не является `null`. Поэтому я считаю, что сделал достаточно, и больше не буду предпринимать что-то сверх того, что уже сделал.

А что насчет имени? Это прежде всего для удобства. Когда вы придете в ресторан, администратор спросит ваше имя, а не адрес электронной почты или номер брони. Если же вы не назвали своего имени при резервировании, ресторан, возможно, сможет найти вас по адресу электронной почты.

Вместо того чтобы отклонять пустое имя, вы можете преобразовать его в пустую строку. Это проектное решение следует закону Постела: будьте либеральны в том, что принимаете, и консервативны в том, что отправляете.

Закон Постела

Проектирование взаимодействия объектов в соответствии с контрактом означает четкое обдумывание пред- и постусловий. Какие условия должен выполнить клиент перед взаимодействием с объектом? Какие гарантии дает объект относительно условий после взаимодействия? Эти вопросы тесно связаны с правилами ввода и вывода.

Чтобы подробно разобраться в пред- и постусловиях, вы можете использовать закон Постела. Я перефразирую его так: *«Будьте консервативны в том, что отправляете, и либеральны в том, что принимаете»*.

Джон Постел первоначально сформулировал этот закон как часть спецификации TCP (Transmission Control Protocol, протокол управления передачей), но я считаю его полезным в более широком контексте разработки API.

Чем надежнее ваши гарантии и чем меньше вы требуете от другой стороны при заключении контракта, тем более привлекательным становится этот контракт.

Когда дело доходит до проектирования API, я обычно интерпретирую закон Постела как разрешение ввода, но не более того. Как следствие, хотя вы должны быть либеральны в том, что принимаете, все равно будут входные данные, которые вы не сможете принять. Как только вы с этим столкнетесь, быстро потерпите неудачу и отклоните входные данные.

У вас по-прежнему должен быть драйвер для этого изменения, поэтому добавьте еще один тест-кейс, как показано в листинге 5.10. Самое значительное изменение по сравнению с листингом 5.1 — это новый тест-кейс, который задается третьим атрибутом `[InlineData]`. Изначально этот тест провальный, как и должно быть в соответствии с циклом «красный, зеленый, рефакторинг».

Листинг 5.10. Еще один тестовый пример ввода пустого имени. Выделенный тест-кейс новый по сравнению с листингом 5.1 (Restaurant/c31e671/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Theory]
[InlineData(
    "2023-11-24 19:00", "juliad@example.net", "Julia Domna", 5)]
[InlineData("2024-02-13 18:15", "x@example.com", "Xenia Ng", 9)]
[InlineData("2023-08-23 16:55", "kite@example.edu", null, 2)]
public async Task PostValidReservationWhenDatabaseIsEmpty(
    string at,
    string email,
    string name,
    int quantity)
{
    var db = new FakeDatabase();
    var sut = new ReservationsController(db);

    var dto = new ReservationDto
    {
        At = at,
        Email = email,
        Name = name,
        Quantity = quantity
    };
    await sut.Post(dto);

    var expected = new Reservation(
        DateTime.Parse(dto.At, CultureInfo.InvariantCulture),
        dto.Email,
        dto.Name ?? "",
        dto.Quantity);
    Assert.Contains(expected, db);
}
```

В «зеленой» фазе сделайте тест проходным, как, например, в листинге 5.11. Вы могли бы использовать стандартный тернарный оператор, но оператор нулевого слияния в C# (`??`) — более компактная альтернатива. В некотором смысле он заменяет оператор `!`, но это хорошее решение, так как оператор `??` не подавляет механизм проверки компилятора на `null`.

На этапе *рефакторинга* вы должны подумать о возможности внести какие-либо улучшения в код. Думаю, это можно сделать. Нет правила, запрещающего проводить небольшую проверку между фазами «крас-

ный» и «рефакторинг». А пока сделайте коммит текущих изменений в Git и рассмотрите их пуск через конвейер развертывания.

Листинг 5.11. Метод Post преобразует пустые имена в пустую строку (Restaurant/c31e671/Restaurant.RestApi/ReservationsController.cs)

```
public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));
    if (!DateTime.TryParse(dto.At, out var d))
        return new BadRequestResult();
    if (dto.Email is null)
        return new BadRequestResult();
    if (dto.Quantity < 1)
        return new BadRequestResult();

    var r =
        new Reservation(d, dto.Email, dto.Name ?? "", dto.Quantity);
    await Repository.Create(r).ConfigureAwait(false);

    return new NoContentResult();
}
```

5.3. ЗАЩИТА ИНВАРИАНТОВ

Вас ничего не смущает в листинге 5.11?

Вас не беспокоит сложность кода? Хотя выглядит он не так уж и плохо. Visual Studio содержит встроенный калькулятор простых метрик кода, таких как цикломатическая сложность, глубина наследования, подсчет количества строк кода и т. д. Метрика, на которую я обращаю больше всего внимания, — это цикломатическая сложность. Если она превышает 7, нужно уменьшить число[1]. В нашем случае это 6.

С другой стороны, если вы рассматриваете всю систему, происходит нечто большее. Пока метод `Post` проверяет предварительные условия того, что представляет собой фактическое резервирование, вся информация теряется. Он вызывает метод `Create` в своем

[1] Помните, что я использую число 7 как символ ограничения кратковременной памяти мозга (см. подраздел 3.2.1).

репозитории (`Repository`). Напомню, что он реализован классом `SqlReservationsRepository` в коде листинга 4.19.

Если вы занимаетесь техническим обслуживанием, и первое знакомство с кодовой базой происходит на примере листинга 4.19, у вас могут возникнуть вопросы по поводу параметра `reservation`.

Соответствует ли `At` фактической дате? Есть ли гарантия того, что значение `Email` не будет пустым? Отображает ли `Quantity` натуральное число?

Вы можете проанализировать класс `Reservation` в листинге 4.12 и убедиться, что `Email` действительно гарантированно не будет нулевым, поскольку вы использовали систему типов, чтобы объявить его не допускающим значение `null`. То же самое относится и к дате. Но что насчет количества? Вы уверены, что оно не отрицательное или не равно 0?

Единственный способ ответить на этот вопрос — провести анализ. Какой еще код вызывает метод `Create`? Сейчас есть только одно место вызова, но в будущем это может измениться. Что делать, если вызовов было несколько? Слишком много информации, чтобы держать все в голове.

Не было бы проще, если бы был какой-то способ, гарантирующий, что объект уже проверен?

5.3.1. Постоянная валидность

Инкапсуляция должна гарантировать, что объект никогда не будет находиться в невалидном состоянии. У этого понятия есть два аспекта: валидность и состояние.

Вы уже знакомы с законом Постела, который помогает решить, что валидно, а что — нет. А как быть с состоянием?

Состояние объекта — это совокупность составляющих его значений, которая всегда должна быть валидной. Если объект поддерживает изменение, то каждая операция, изменяющая его состояние, должна гарантировать, что не приведет к невалидному состоянию.

Одно из многих положительных качеств неизменяемых объектов — вам нужно учитывать валидность только в одном месте: в конструкторе. В случае успешной инициализации объект должен быть в валидном состоянии. Сейчас для класса `Reservation` (см. листинг 4.12) это не так.

Мы все еще далеки до идеала. Вы должны убедиться, что не можете создать объект `Reservation` с отрицательным значением количества. Для управления этим изменением используйте параметризованный тест [66], как в листинге 5.12.

Листинг 5.12. Параметризованный тест, который подтверждает, что вы не можете создавать объекты Reservation с невалидным значением количества (Restaurant/b3ca85e/Restaurant.RestApi.Tests/ReservationTests.cs)

```
[Theory]
[InlineData( 0)]
[InlineData(-1)]
public void QuantityMustBePositive(int invalidQantity)
{
    Assert.Throws<ArgumentOutOfRangeException>(
        () => new Reservation(
            new DateTime(2024, 8, 19, 11, 30, 0),
            "mail@example.com",
            "Marie Ilswe",
            invalidQantity));
}
```

Я решил параметризовать этот метод тестирования, потому что считаю, что нулевое значение фундаментально отличается от отрицательных чисел. Возможно, вы думаете, что 0 — натуральное число, возможно, нет. Как и с множеством других вещей[1], единого мнения на этот счет не существует. Тем не менее тест дает понять, что 0 — невалидная величина. В качестве примера отрицательного числа используется и число –1.

Тест утверждает, что при попытке инициализировать объект `Reservation` с невалидным значением количества он должен генерировать

[1] Что такое модуль (юнит)? Что такое mock-объект?

исключение. Обратите внимание, что он не утверждает сообщение об исключении, так как текст сообщения не является частью *поведения* объекта. Это не значит, что сообщение неважно, но значит, что нет необходимости проводить лишние тесты, и если вы позже захотите изменить сообщение об исключении, вам придется отредактировать как тестируемую систему, так и сам тест. Не повторяйтесь [50].

В фазе «красный» этот тест не проходит. Перейдите к фазе «зеленый», сделав его успешным. В листинге 5.13 показан получившийся в результате конструктор.

Класс `Reservation` неизменяем. Это гарантирует, что он никогда не окажется невалидным[1], и весь код, который обрабатывает объекты `Reservation` может обходиться без защитного программирования. Свойства `At`, `Email`, `Name` и `Quantity` гарантированно будут заполнены, а `Quantity` будет положительным числом. В подразделе 7.2.5 мы снова вернемся к классу `Reservation`, чтобы воспользоваться этими гарантиями.

Листинг 5.13. Конструктор Reservation, проверка на отрицательное значение количества (Restaurant/b3ca85e/Restaurant.RestApi/Reservation.cs)

```
public Reservation(
    DateTime at,
    string email,
    string name,
    int quantity)
{
    if (quantity < 1)
        throw new ArgumentOutOfRangeException(
            nameof(quantity),
            "The value must be a positive (non-zero) number.");

    At = at;
    Email = email;
    Name = name;
    Quantity = quantity;
}
```

[1] Я делаю вид, что FormatterServices.GetUninitializedObject не существует. Не используйте этот метод.

5.4. ЗАКЛЮЧЕНИЕ

Инкапсуляция — одна из наиболее неправильно понимаемых концепций ООП. Многие программисты считают, что это запрет на прямое раскрытие полей классов — поля классов должны быть «инкапсулированы» за геттерами и сеттерами. Но к инкапсуляции это имеет мало отношения.

Наиболее важно то, что объект должен гарантировать, что никогда не окажется в невалидном состоянии.

Взаимодействие между объектом и вызывающей стороной должно подчиняться контракту — набору пред- и постусловий. Предварительные условия описывают обязанности вызывающего кода, и если он выполняет их, постусловия описывают гарантии, предоставляемые объектом.

Пред- и постусловия вместе образуют *инварианты*. Вы можете использовать закон Постела для разработки качественного контракта: чем меньше вы опрашиваете вызывающий код, тем легче ему взаимодействовать с объектом — чем больше гарантий вы можете дать, тем меньше защитного кода будет написано.

6 ТРИАНГУЛЯЦИЯ

Однажды я помогал своему клиенту с унаследованной (легаси) кодовой базой. У меня была возможность пообщаться с некоторыми разработчиками, и я спросил нового члена команды, сколько времени прошло, прежде чем он почувствовал, что может внести свой вклад самостоятельно.

«Три месяца», — ответил он.

Столько ему понадобилось, чтобы запомнить кодовую базу до такой степени, чтобы он мог уверенно ее редактировать. Конечно, некоторые из процессов были действительно сложными. Иногда происходило более семи событий одновременно, а точнее, более семидесяти.

Нужно время, чтобы научиться ориентироваться в такой кодовой базе, но это не невозможно. Вы могли подумать, что это опровергает тезис о том, что человеческий мозг может отслеживать только семь вещей. Я думаю, что это предположение по-прежнему в силе, и сейчас объясню почему.

6.1. КРАТКОВРЕМЕННАЯ И ДОЛГОВРЕМЕННАЯ ПАМЯТЬ

В подразделе 3.2.1 упоминалось, что число *семь* относится к кратковременной памяти. Но, помимо нее, есть еще и долговременная [80].

В том же разделе я упоминал и о сомнительности аналогии «мозг как компьютер». Тем не менее очевидно, что у нас есть своего рода хранилище памяти с большим объемом, хоть и ненадежное. Оно отличается от кратковременной памяти, хотя между ними есть и некоторая связь (рис. 6.1).

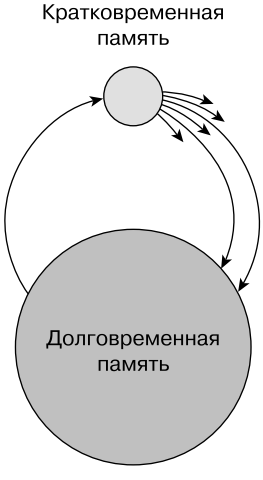


Рис. 6.1. Кратковременная память намного меньше долговременной. Большинство фрагментов первой исчезают, когда «выходят за рамки», но некоторые иногда попадают в долговременную память, где могут оставаться довольно долго. Информация из долговременной памяти тоже может быть извлечена и «загружена» в кратковременную. Здесь сразу напрашивается аналогия с оперативной памятью и жестким диском, но не относитесь к ней слишком буквально

Когда вы просыпаетесь, вы можете вспомнить лишь некоторые фрагменты сна, так как эта информация быстро стирается из памяти.

Когда-то давно людям приходилось запоминать номера телефонов. Сейчас, возможно, нужно помнить только одноразовый код двухфакторной аутентификации, чтобы ввести его. И его мы забудем в течение следующих нескольких минут.

Некоторые фрагменты информации, однако, могут сначала появиться в кратковременной памяти, но затем вы решите, что они достаточно важны и вам нужно закрепить их в долговременной. Когда в 1995 году я встретил свою будущую жену, я решил запомнить ее номер телефона.

И наоборот, вы можете вызывать информацию из долговременной памяти и работать с ней в кратковременной. Например, вы могли запомнить разные API. При написании кода вы извлекаете соответствующие методы из своей кратковременной памяти и комбинируете их.

6.1.1. Легаси-код и память

При работе с унаследованным кодом структура кодовой базы медленно и тщательно фиксируется в долговременной вашей памяти. Вы можете работать с легаси-кодом, но есть (как минимум) две проблемы:

- нужно время для изучения кодовой базы;
- изменения даются тяжело.

Один только первый пункт должен заставить менеджеров по найму задуматься. Если нужно три месяца, прежде чем новый сотрудник сможет продуктивно работать, то разработчики становятся незаменимыми. Если подойти к вопросу довольно цинично, то работа с легаси-кодом гарантирует вам как сотруднику определенную степень безопасности. Но даже так подобный опыт может быть разочаровывающим — как и факт того, что поиск новой работы может быть затруднен, так как ваши навыки будет сложно кому-то передать.

Что еще хуже, так это второй пункт. Информацию, закрепленную в долговременной памяти, сложнее изменить. Что произойдет, если вы попытаетесь улучшить код?

В книге «Ускоряйся! Наука DevOps. Эффективная работа с унаследованным кодом» [27] можно найти много способов улучшения сложного кода. И все они предполагают изменение его структуры.

Что произойдет, если вы измените структуру кода (рис. 6.2)? Информация в вашей долговременной памяти устареет. Работать с кодовой базой будет все труднее, так как ваши приобретенные знания станут неприменимыми.

Помимо того, что с легаси-кодом сложно работать, от него еще и сложно избавиться.

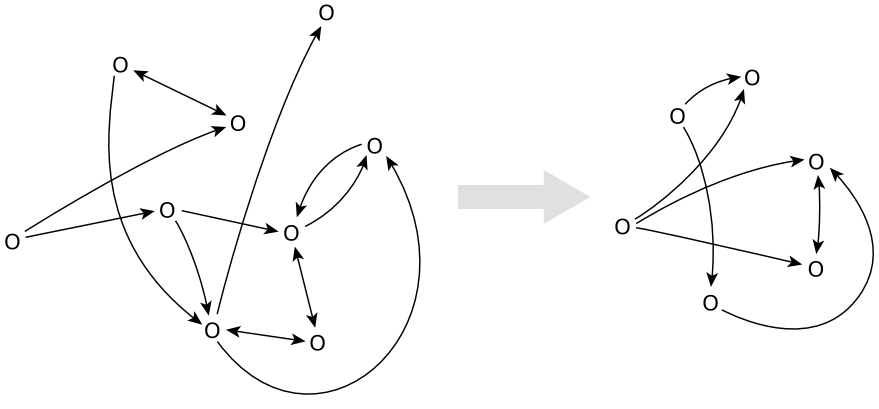


Рис. 6.2. Рефакторинг унаследованного кода может принести ряд проблем. Представьте, что диаграмма слева — это сложная система. Возможно, вы сможете провести рефакторинг и упростить ее. Что произойдет, если система справа, хоть она и проще, все же слишком сложна, чтобы уместиться в вашей голове? Возможно, вы запомнили систему слева, но та, что справа, — новая. Ваши приобретенные знания теперь недействительны — их место заняло нечто неизвестное. Было бы лучше вообще избегать написания устаревшего кода

6.2. ОБЪЕМ ПАМЯТИ

Программная инженерия должна поддерживать организацию, которой принадлежит ПО. Вы разрабатываете надежные кодовые базы, убеждаясь, что код умещается в ваш мозг. Объем вашей рабочей памяти равен семи, поэтому одновременно должно происходить лишь несколько событий.

Любая нетривиальная часть программного обеспечения будет иметь гораздо больше событий, поэтому вам нужно разделить структуру кода на небольшие фрагменты, что поместятся в вашей голове.

Кент Бек выразился на этот счет так:

> *«Цель разработки ПО — создание фрагментов или срезов, которые умещаются в человеческом сознании. Программное обеспечение продолжает расти, но возможности человеческого мозга ограничены, поэтому, чтобы продолжать вносить изменения, мы должны продолжать фрагментировать» [10].*

Как это сделать — фундаментальная тема программной инженерии. К счастью, есть набор эвристик, которые могут помочь.

Я думаю, что лучше всего учиться на примерах. Сейчас пример из книги все еще слишком прост. Нужно создать более сложную кодовую базу, чтобы декомпозиция стала необходимостью.

6.2.1. Переполнение

Текущая система онлайн-бронирования позволяет зарезервировать столик, если в поле для ввода количества гостей указано любое положительное значение. Но возможности самого заведения ограниченны. Кроме того, все столики могут быть уже полностью зарезервированы на определенную дату. Система должна сверить поступающее резервирование с уже существующими и с вместимостью ресторана.

Руководствуясь основной методикой этой книги, используйте тест как драйвер для новой функциональности. Какой тест вам нужно написать?

В листинге 5.11 приведен пример самой последней версии метода `Post`. Если вы придерживаетесь TPP-подхода [64], то логично будет выполнить следующую трансформацию: *unconditional* → *if*. Вам нужно разделить путь выполнения, возвращая `204 No Content`, если

все идет хорошо, но возвращать некоторый код состояния ошибки, если запрос выходит за рамки возможностей ресторана. В листинге 6.1 показан тест, который нужно написать для управления таким поведением.

Сначала тест делает резервирование, а затем пытается сделать еще одно. Обратите внимание, что код структурирован в соответствии с эвристикой паттерна *Arrange-Act-Assert*. Пустые строки четко очерчивают три фазы теста.

Первое резервирование для шести человек — часть этапа *arrange*, а второе — этап *act*.

Листинг 6.1. Проверка невозможности избыточного резервирования. Обратите внимание, что в этом тесте неявно указана вместимость ресторана. Вы должны подумать над тем, как сделать ее более явной (Restaurant/b3694bd/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Fact]
public async Task OverbookAttempt()
{
    using var service = new RestaurantApiFactory();
    await service.PostReservation(new
    {
        at = "2022-03-18 17:30",
        email = "mars@example.edu",
        name = "Marina Seminova",
        quantity = 6
    });

    var response = await service.PostReservation(new
    {
        at = "2022-03-18 17:30",
        email = "shli@example.org",
        name = "Shanghai Li",
        quantity = 5
    });

    Assert.Equal(
        HttpStatusCode.InternalServerError,
        response.StatusCode);
}
```

Наконец, предположение подтверждает, что ответ: `500 Internal Server Error`[1].

Так почему ожидаемый результат ошибочен? По тесту ничего не понятно. Сделайте пометку, чтобы вернуться к нему позже и улучшить его. Этот метод описан Кентом Беком в книге *Test Driven Development: By Example* [9]: в процессе написания тестов вы думаете о других моментах, которые нужно улучшить. Не отвлекайтесь! Запишите свои идеи и двигайтесь дальше.

Скрытая проблема из примера кода листинга 6.1 в том, что оба резервирования относятся к одной и той же дате. Первое резервирование предназначено для шести человек, и, хотя явного утверждения нет, тест предполагает, что оно выполнено успешно. Другими словами, вместимость ресторана должна быть не менее шести человек.

Следующее бронирование на пять человек уже проваливается. Из названия теста ясно, что тестовый пример — это попытка избыточного резервирования. Ресторан не рассчитан на одиннадцать человек. Неявно тест сообщает нам, что вместимость ресторана — от шести до десяти человек.

Код должен быть более явным. Как гласит один из принципов «Дзена питона»,

> *«явное лучше неявного» [79].*

Это правило применяется как к тестовому, так и к продакшен-коду. Тест в листинге 6.1 должен сделать вместимость ресторана более явной. Я мог бы сделать это до того, как покажу вам код, но хочу, чтобы вы увидели, как писать код поэтапно. Этот процесс включает

---

[1] Это спорное проектное решение. Всякий раз, когда я возвращаю этот код состояния, разработчики утверждают, что 500 Internal Server Error зарезервирован для действительно неожиданных ошибок. Тогда у меня возникает вопрос: «Какой код состояния HTTP использовать вместо этого?» Я не нашел в этом отношении полезными ни спецификации HTTP 1.1, ни RESTful Web Services Cookbook [2]. В любом случае от этого конкретного кода состояния ничего не зависит. Если вы хотите использовать другой, просто замените 500 Internal Server Error на тот, который вам больше нравится.

в себя создание пространства для внедрения улучшений. Обращайте внимание на любые недостатки, но не позволяйте им замедлять вас. Помните, что лучшее — враг хорошего. Итак, продолжаем.

Однажды я обедал в хипстерском ресторане в Бруклине. Единственным свободным сидячим местом во всем заведении было место за барной стойкой с видом на кухню (рис. 6.3). Ресторан вмещает двенадцать человек, и если вы заранее не зарезервируете места, ваши друзья будут сидеть с другими людьми. Сервировка начиналась в 18:30, независимо от вашего присутствия. Такие заведения есть. Я учитываю это, так как ресторан соблюдает свои простые правила резервирования. Есть один общий стол и только одна рассадка в день. Это расстановка, на которую мы будем нацелены, по крайней мере пока.

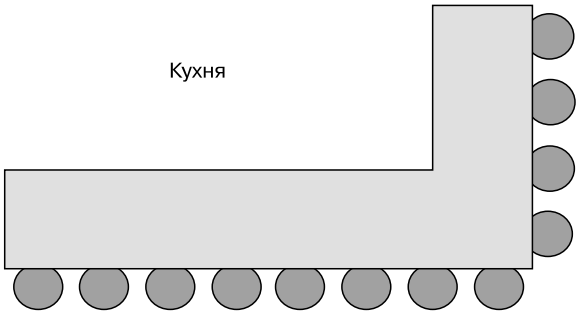


Рис. 6.3. Пример макета расстановки. В этом ресторане есть только сидячие места за барной стойкой с видом на кухню

Какое простейшее решение может сработать [22]? Взгляните на код листинга 6.2.

Листинг 6.2. Несмотря на тестовое покрытие, код, выполненный в этой версии метода Post, не реализует желаемые бизнес-правила (Restaurant/b3694bd/Restaurant.RestApi/ReservationsController.cs)

```
public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));
    if (!DateTime.TryParse(dto.At, out var d))
        return new BadRequestResult();
```

```
    if (dto.Email is null)
        return new BadRequestResult();
    if (dto.Quantity < 1)
        return new BadRequestResult();

    if (dto.Email == "shli@example.org")
        return new StatusCodeResult(
            StatusCodes.Status500InternalServerError);

    var r =
        new Reservation(d, dto.Email, dto.Name ?? "", dto.Quantity);
    await Repository.Create(r).ConfigureAwait(false);

    return new NoContentResult();
}
```

Хотя эта реализация явно неверна, новый тест выполняется успешно, поэтому отправьте коммит изменений в Git.

6.2.2. Метод «Адвокат дьявола»

Вы уже видели, как в коде листинга 4.15 жестко запрограммированы данные, сохраненные в базе. Я называю такое преднамеренное препятствие методом «Адвокат дьявола» [98]. Его не обязательно применять постоянно, но он может быть полезен.

Я периодически преподаю TDD-разработку и заметил, что новичкам часто трудно создавать хорошие тестовые сценарии. Как вы узнаете, что написали достаточно тест-кейсов?

«Адвокат дьявола» — это метод, который поможет вам ответить на этот вопрос. Идея в том, чтобы намеренно попытаться пройти все тесты с заведомо неполной реализацией, как в листинге 6.2.

Это полезно, потому что работает как критика ваших тестов. Если вы можете написать простую, но явно недостаточную реализацию, значит, вам нужно создать больше тест-кейсов для достижения желаемого поведения. Этот процесс можно назвать своеобразной триангуляцией [9], как выразился Роберт Мартин,

> *«чем конкретнее становятся тесты, тем более общим становится код» [64].*

Чтобы предложить правильную реализацию, вам нужно добавить как минимум еще один тест-кейс. К счастью, новый кейс — это обычно просто новая строка данных в параметризованном тесте [66] (листинг 6.3).

Листинг 6.3. Проверка успешного резервирования. Единственное отличие от кода листинга 5.10 — добавление выделенного четвертого набора тестов (Restaurant/5b82c77/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Theory]
[InlineData(
    "2023-11-24 19:00", "juliad@example.net", "Julia Domna", 5)]
[InlineData("2024-02-13 18:15", "x@example.com", "Xenia Ng", 9)]
[InlineData("2023-08-23 16:55", "kite@example.edu", null, 2)]
[InlineData("2022-03-18 17:30", "shli@example.org", "Shanghai Li", 5)]
public async Task PostValidReservationWhenDatabaseIsEmpty(
    string at,
    string email,
    string name,
    int quantity)
{
    var db = new FakeDatabase();
    var sut = new ReservationsController(db);

    var dto = new ReservationDto
    {
        At = at,
        Email = email,
        Name = name,
        Quantity = quantity
    };
    await sut.Post(dto);

    var expected = new Reservation(
        DateTime.Parse(dto.At, CultureInfo.InvariantCulture),
        dto.Email,
        dto.Name ?? "",
        dto.Quantity);
    Assert.Contains(expected, db);
}
```

Возможно, это был не тот метод тестирования, на который вы рассчитывали. Быть может, вы думали, что новый тест-кейс нужно было добавить в метод `OverbookAttempt`, с которым мы сейчас работаем (см. листинг 6.1). Вместо этого мы используем четвертый пример для старого теста (`PostValidReservationWhenDatabaseIsEmpty`). Почему так?

Рассмотрим предпосылку приоритета трансформации [64]. Что не так с листингом 6.2? В коде выполняется разветвление на константу (строка shli@example.org). К какому преобразованию кода нужно стремиться, чтобы улучшить его? Самый подходящий вариант: *constant → scalar*. Нам не нужно, чтобы ветвление переходило к константе, нам нужно, чтобы оно переходило к переменной.

Код в листинге 6.2 подразумевает, что адрес электронной почты shli@example.org почему-то недопустим, что не соответствует действительности. Какой тест-кейс нужно добавить, чтобы исправить ситуацию? Тот, где shli@example.org включен в успешное резервирование. Код листинга 6.3 добавляет точно такое же резервирование, но с другими обстоятельствами. В тестовом методе `PostValidReservationWhenDatabaseIsEmpty` предварительное резервирование не предусмотрено.

К сожалению, метод «Адвокат дьявола» может конфликтовать с реализацией из листинга 6.4.

Листинг 6.4. Тесты провоцируют метод Post учитывать существующее резервирование, чтобы решить, отклонять его или нет, но реализация по-прежнему неверна (Restaurant/5b82c77/Restaurant.RestApi/ReservationsController.cs)

```
public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));
    if (!DateTime.TryParse(dto.At, out var d))
        return new BadRequestResult();
    if (dto.Email is null)
        return new BadRequestResult();
    if (dto.Quantity < 1)
        return new BadRequestResult();

    var reservations =
        await Repository.ReadReservations(d).ConfigureAwait(false);
    if (reservations.Any())
        return new StatusCodeResult(
            StatusCodes.Status500InternalServerError);

    var r =
        new Reservation(d, dto.Email, dto.Name ?? "", dto.Quantity);
```

```
    await Repository.Create(r).ConfigureAwait(false);

    return new NoContentResult();
}
```

Новый тест-кейс в листинге 6.3 не позволяет отклонить резервирование, основанное только на `dto`. Вместо этого метод должен рассматривать более расширенное состояние приложения, чтобы пройти все тесты.

Он правильно делает это, вызывая `ReadReservations` для внедренного `Repository`, но неправильно решает отклонить резервирование, если на эту дату уже есть какие-либо бронирования. Мы на верном пути.

6.2.3. Существующее резервирование

Метод `ReadReservations` — это новый элемент интерфейса `IReservationsRepository` из листинга 6.5. Реализации должны возвращать все резервирования на указанную дату.

Листинг 6.5. Ниже выделено добавление метода ReadReservations в сравнении с листингом 4.14 (Restaurant/5b82c77/Restaurant.RestApi/IReservationsRepository.cs)

```
public interface IReservationsRepository
{
    Task Create(Reservation reservation);

    Task<IReadOnlyCollection<Reservation>> ReadReservations(
        DateTime dateTime);
}
```

Добавляя в интерфейс новый элемент, вы нарушаете существующие реализации. В нашей кодовой базе таких элемента два: `SqlReservationsRepository` и специфичный для теста `FakeDatabase`. Реализация `Fake` [66] очень простая (листинг 6.6). `Fake` использует LINQ для поиска в самом себе всех броней в период между полуночью и до тика[1] полуночи следующего дня.

[1] В .NET тик равен сотне наносекунд и представляет наименьшее разрешение встроенного API даты и времени.

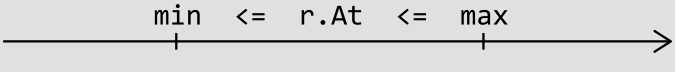
Листинг 6.6. Реализация FakeDatabase метода ReadReservations. Напомню, что FakeDatabase наследуется от базового класса коллекции (см. листинг 4.13), поэтому может использовать LINQ для самофильтрации (Restaurant/5b82c77/Restaurant.RestApi.Tests/FakeDatabase.cs)

```
public Task<IReadOnlyCollection<Reservation>> ReadReservations(
    DateTime dateTime)
{
    var min = dateTime.Date;
    var max = min.AddDays(1).AddTicks(-1);

    return Task.FromResult<IReadOnlyCollection<Reservation>>(
        this.Where(r => min <= r.At && r.At <= max).ToList());
}
```

Запись числовых выражений в порядке следования строк

Обратите внимание, что выражение фильтра в листинге 6.6 записано в *порядке следования строк*. Переменные располагаются в порядке возрастания слева направо. `min` — минимальное значение, поэтому поместите его в крайнее левое положение, как если бы вы делали это на числовой прямой.



С другой стороны, `max` должно быть наибольшим значением, поэтому поместите его в крайнее правое положение. Переменная, с которой связано выражение фильтра, — это `r.At`, поэтому поместите ее между двумя крайними значениями.

Такая организация сравнений облегчает восприятие читающему [65]. Значения в ней размещаются на неявной непрерывной числовой строке. Это значит, что вы будете использовать только операторы «меньше» и «меньше или равно» вместо операторов «больше» и «больше или равно».

Другая реализация интерфейса `IReservationsRepository` — `SqlReservationsRepository`. У него тоже должна быть правильная реализация. Как и раньше, с этим классом можно обращаться как со «скромным объектом» [66], так что автотесты вам не потребуются. Это простой SQL-запрос `SELECT`, поэтому я не буду его описывать здесь. Если вам интересны подробности, обратитесь к репозиторию исходного кода, прилагающемуся к книге.

6.2.4. Метод «Адвокат дьявола» и цикл «красный, зеленый, рефакторинг»

Код в листинге 6.4 еще не завершен. Хотя он запрашивает БД для существующих резервирований, он отклоняет новые, если уже есть хотя бы одно. Тем не менее все тесты успешно выполняются.

Используя процесс триангуляции Роберта Мартина [64], добавляйте больше тест-кейсов, пока не победите «дьявола». Какой же кейс добавить следующим?

Система должна принимать резервирование, до тех пор пока в ней достаточно свободных мест, даже если в ней уже есть одно или несколько резервирований на текущую дату. Это предполагает тест как в листинге 6.7.

Листинг 6.7. Проверка возможности резервирования столика, даже если на ту же дату уже есть зарезервированные столики (Restaurant/bf48e45/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Fact]
public async Task BookTableWhenFreeSeatingIsAvailable()
{
    using var service = new RestaurantApiFactory();
    await service.PostReservation(new
    {
        at = "2023-01-02 18:15",
        email = "net@example.net",
        name = "Ned Tucker",
        quantity = 2
    });
```

```
    var response = await service.PostReservation(new
    {
        at = "2023-01-02 18:30",
        email = "kant@example.edu",
        name = "Katrine Nwhr Troelsen",
        quantity = 4
    });

    Assert.True(
        response.IsSuccessStatusCode,
        $"Actual status code: {response.StatusCode}.");
}
```

Как и в листинге 6.7, этот тест добавляет одно резервирование на этапе *arrange* и еще одно на этапе *act*, но, в отличие от первого, здесь ожидается успешный результат. Так происходит потому, что общее количество равно шести, а мы знаем, что ресторан может вместить по крайней мере шесть гостей.

По силам ли «Адвокату дьявола» победить этот тест? Другими словами, можно ли изменить метод `Post` так, чтобы все тесты были пройдены успешно, но при этом бизнес-правило не было корректно реализовано?

Да, но сделать это становится все труднее. В листинге 6.8 приведен соответствующий фрагмент метода `Post` (то есть не весь метод). В нем используется LINQ для преобразования `reservations` в набор количеств, а затем из них выбирается только первое.

Метод `SingleOrDefault` возвращает значение, если коллекция содержит один элемент, или значение по умолчанию, если коллекция пуста. Значение `int` по умолчанию равно `0`, поэтому, пока резервирований нет или есть только одно, это будет работать.

Листинг 6.8. Фрагмент метода Post, где решается, отклонять резервирование или нет. Метод «Адвокат дьявола» по-прежнему пытается обойти требования, установленные набором тестов. Вместимость ресторана жестко запрограммирована на десять человек (Restaurant/bf48e45/Restaurant.RestApi/ReservationsController.cs)

```
var reservations =
    await Repository.ReadReservations(d).ConfigureAwait(false);
int reservedSeats =
```

```
    reservations.Select(r => r.Quantity).SingleOrDefault();
if (10 < reservedSeats + dto.Quantity)
    return new StatusCodeResult(
        StatusCodes.Status500InternalServerError);
```

Если в коллекции более одного элемента, метод `SingleOrDefault` вызовет исключение, но, поскольку ни один из тестов не реализует этот сценарий, все тесты будут успешно пройдены.

Похоже, в очередной раз «Адвокат дьявола» сорвал наши планы по реализации. Должны ли мы написать еще один тест?

Мы могли бы, но не стоит забывать о цикле «красный, зеленый, рефакторинг». В листинге 6.7 представлена фаза *«красный»*, а в листинге 6.8 — *«зеленый»*. Теперь пришло время последней фазы. Можете ли вы улучшить код в листинге 6.8?

В коде уже используется LINQ. А почему бы не заменить `SingleOrDefault` на `Sum`? В листинге 6.9 приведен пример метода `Post` после рефакторинга. Сравните логику принятия решения в середине метода с кодом листинга 6.8. Все стало гораздо проще!

Код в листинге 6.9 все еще проходит все тесты, но он более общий. Это улучшение, поэтому сверьте изменения с Git.

Как понять, когда проводить рефакторинг? Откуда вы знаете о существовании метода `Sum`? Эти знания можно получить только из опыта. Я не обещал, что *в разработке ПО* все можно будет просчитать. Если бы это было так, нас можно было бы полностью заменить машинами.

Листинг 6.9. Метод Post теперь верно определяет, принять или отклонить бронирование, основываясь на общем количестве резервирований. Вместимость ресторана жестко запрограммирована на десять человек — это еще один недостаток, который нам нужно устранить (Restaurant/9963056/Restaurant.RestApi/ReservationsController.cs)

```
public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));
    if (!DateTime.TryParse(dto.At, out var d))
        return new BadRequestResult();
```

```
    if (dto.Email is null)
        return new BadRequestResult();
    if (dto.Quantity < 1)
        return new BadRequestResult();

    var reservations =
        await Repository.ReadReservations(d).ConfigureAwait(false);
    int reservedSeats = reservations.Sum(r => r.Quantity);
    if (10 < reservedSeats + dto.Quantity)
        return new StatusCodeResult(
            StatusCodes.Status500InternalServerError);
    var r =
        new Reservation(d, dto.Email, dto.Name ?? "", dto.Quantity);
    await Repository.Create(r).ConfigureAwait(false);

    return new NoContentResult();
}
```

6.2.5. Когда тестов будет достаточно?

Оставил ли рефакторинг открытый фланг? Что, если позже другой разработчик вернется к использованию `SingleOrDefault`? Тесты все равно будут успешно пройдены, но реализация будет некорректной.

Это важный вопрос, но я не могу дать вам точного ответа. Обычно я спрашиваю себя: насколько вероятно, что произойдет такая регрессия?

Я стараюсь думать, что у других программистов добрые намерения[1]. Тесты существуют, чтобы не допустить ошибок, которые склонен совершать наш мозг. Итак, насколько вероятно, что программист изменит вызов `Sum` на вызов `SingleOrDefault`?

Я не считаю, что вероятность высока, но если бы это произошло, каковы были бы последствия? Мы получим необработанные исклю-

[1] Это зависит от контекста. Представьте себе проект с открытым исходным кодом, предназначенный для чего-то важного, например для обеспечения безопасности или управления оборудованием. Если бы участник мог пропихнуть внутрь вредоносный код, это могло бы иметь реальные последствия. Возможно, в таких обстоятельствах не помешало бы быть более осторожным.

чения в продакшен-среде. Будем надеяться, что мы быстро обнаружим проблему и исправим ее. И если такое произойдет, обязательно напишите автоматизированный тест, воспроизводящий этот дефект. Любая неисправность в продакшене демонстрирует возможность повторного возникновения *конкретной* ошибки — что случилось однажды, может случиться и во второй раз. Предотвратите регрессию с помощью теста.

В целом решение о том, достаточно ли у вас тестов, — это стандартная оценка рисков. Соотнесите вероятность неблагоприятного исхода с его последствиями. Я не знаю способа количественно оценить ни вероятность, ни влияние, так что вычисление этого по-прежнему остается в основном искусством.

6.3. ЗАКЛЮЧЕНИЕ

В геометрии (и географической разведке) триангуляция — это процесс определения местоположения точки. Для TDD-разработки это абстрактная метафора.

Когда триангуляция используется в геометрии, рассматриваемая точка уже существует, но вы не знаете ее положения (рис. 6.4, *слева*).

При проверке кодовой базы тесты играют роль измерений. Отличие в том, что когда вы добавляете тест, он измеряет то, чего еще нет (рис. 6.4, *справа*).

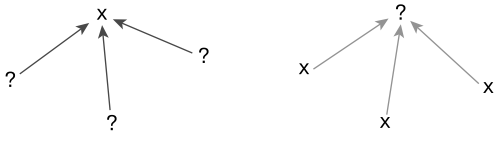


Рис. 6.4. TDD-разработка похожа на триангуляцию — только роли меняются местами: в географической разведке точка уже существует и вы должны измерить точки, от которых вы триангулируете, чтобы вычислить положение точки, а при TDD-разработке тестируемая система изначально не существует, но измерения (в форме тестов) есть

Чем больше измерений вы сделаете в географическом исследовании, тем точнее сможете определить положение точки — чем больше тестов вы добавите, тем лучше опишете систему, которую хотите проверить. Но, чтобы это сработало, важно существенно менять точку зрения между каждым измерением.

Чтобы получить исчерпывающее описание поведения системы без избыточного количества тестов, используйте в совокупности TPP, метод «Адвокат дьявола» и цикл «красный, зеленый, рефакторинг».

7 ДЕКОМПОЗИЦИЯ

Никто и никогда не решит писать устаревший код намеренно. Кодовые базы становятся неактуальными со временем.

Почему так происходит? Все понимают, что файл с тысячами строк кода — плохая идея. Трудно работать с методами, которые занимают сотни строк. Такие кодовые базы сильно усложняют работу программиста.

Но если все это понимают, то почему так происходит?

7.1. ДЕГРАДАЦИЯ КОДА

Код усложняется потому, что каждое вносимое изменение само по себе выглядит незначительно и никто не принимает во внимание общее качество кода. Это происходит постепенно, и однажды вы понимаете, что разработали легаси-кодовую базу и уже слишком поздно что-то менять.

Сначала метод имеет низкую сложность, но по мере внесения исправлений и добавления функционала она возрастает (рис. 7.1). Если вы не учитываете цикломатическую сложность, то проходите заветные семь, а дальше и десять, пятнадцать и двадцать протекающих одновременно событий, не замечая этого.

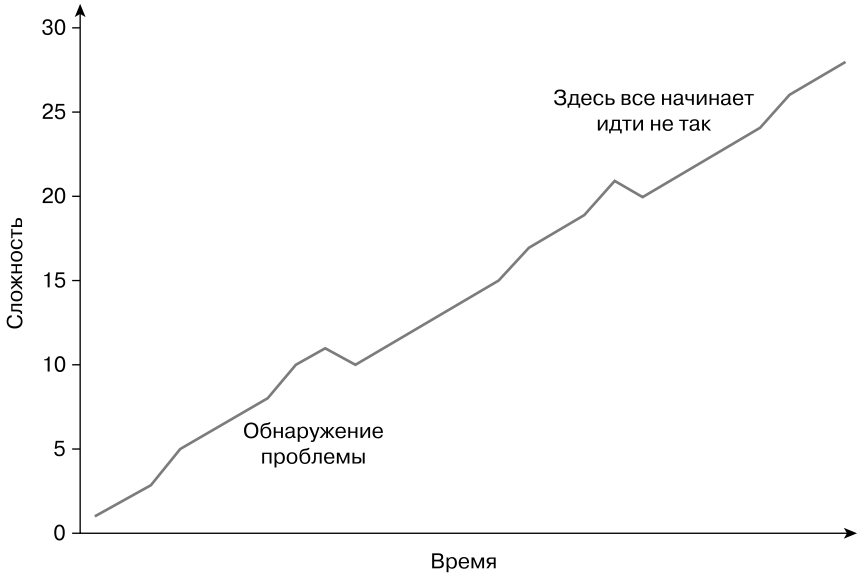


Рис. 7.1. Постепенная деградация кодовой базы. На раннем этапе все начинает идти не так, когда значение сложности пересекает пороговое. Но на метрику никто не обращает внимания, и проблема обнаруживается гораздо позже, когда метрика вырастает настолько, что ее уже может быть невозможно спасти

Однажды вы обнаружите проблему — не потому, что вы наконец обратили внимание на метрику, а потому, что код стал настолько сложным, что все это заметили. Увы, теперь уже поздно что-то менять.

Код деградирует постепенно. Этот процесс можно сравнить с научным анекдотом о лягушке в кипятке.

7.1.1. Пороговые значения

Согласование пороговых значений может помочь предотвратить деградацию кода. Установите норму и отслеживайте показатели. Например, вы можете отслеживать цикломатическую сложность, и если пороговое значение превышает семь, вы отклоняете изменение.

Улучшению качества кода способствует не конкретное число семь, а автоматическая активация правила на основе пороговых значений. Если вы установите десять как граничное значение, это тоже будет иметь место, но я считаю семь подходящим числом, даже если оно определено больше как рекомендация, чем как строгое ограничение. Вспомните из подраздела 3.2.1, что семь — это порог объема рабочей памяти вашего мозга.

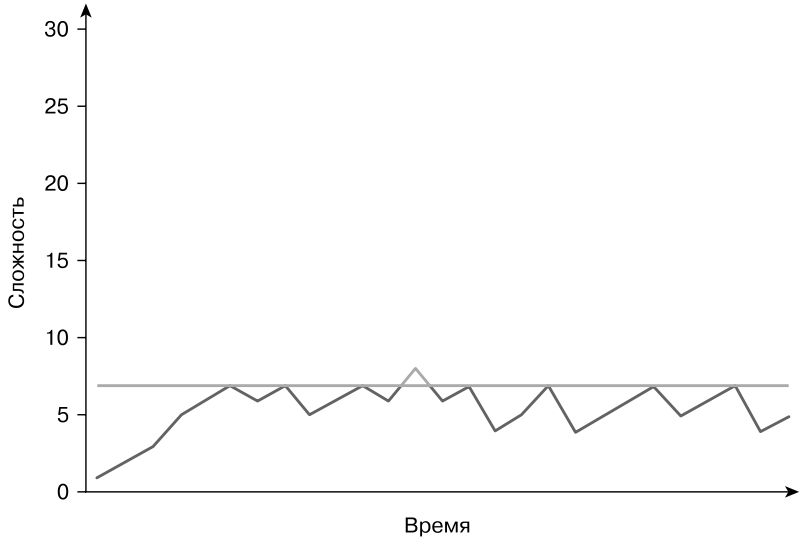


Рис. 7.2. Пороговое значение может помочь контролировать процесс деградации кода

Обратите внимание, что на рис. 7.2 показано, что превышение порогового значения все еще возможно. Правила могут мешать, если вы должны их неукоснительно соблюдать. Бывают ситуации, когда лучший возможный выход — нарушить эти правила. Но после того, как вы отреагируете на ситуацию, найдите способ вернуть нарушающий правило код в состояние, удовлетворяющее норме. Если пороговое значение будет превышено, вы не получите никаких дальнейших предупреждений и есть риск, что этот код будет постепенно деградировать.

Вы можете автоматизировать процесс. Представьте, что анализ цикломатической сложности выполняется как часть построения непрерывной интеграции, и при превышении порогового значения изменения отклоняются. Это в некотором роде преднамеренная попытка повлиять на эффект управления, когда вы получаете то, что измеряете. Делая акцент на цикломатической сложности, вы и ваши коллеги будете уделять ей особое внимание.

Но помните о законе непреднамеренных последствий[1]: будьте внимательны с принятием жестких норм.

Мне удалось ввести нормы для пороговых значений, так как они повышают осведомленность. Это может помочь техническому руководителю уделить внимание качеству, которое нужно улучшить. Как только образ мышления команды изменился, само правило станет избыточным.

7.1.2. Цикломатическая сложность

На протяжении всей книги вы постоянно сталкиваетесь с термином «цикломатическая сложность». Это одна из редких метрик кода, которые я считаю полезными.

Вы могли подумать, что книга по программной инженерии будет полна метрик, но, скорее всего, уже поняли, что это не так. Можно создать мириады метрик кода[2], но большинство из них будут бесполезны. Предварительные исследования показывают, что самая простая метрика — количество строк кода — лучше всего определяет сложность [43]. На этом этапе мне важно убедиться, что все читатели поняли суть.

---

[1] О непреднамеренных последствиях и ложных стимулах вы можете прочитать в «Фрикономике» [57] и «Суперфрикономике» [58]. Хотя эти названия звучат нелепо, я, как дипломированный экономист, могу за них поручиться.

[2] См. Object-Oriented Metrics in Practice [56].

Чем больше строк кода, тем хуже кодовая база. Количество строк кода — показатель производительности только в том случае, если вы измеряете количество *удаленных строк*. Чем больше строк вы добавите, тем больше кода придется прочитать и понять другим специалистам.

Хотя строки кода — довольно практичный показатель сложности, цикломатическая сложность полезна по другим причинам. Это удобный инструмент анализа, поскольку он не только информирует вас о сложности, но и помогает при выполнении модульного тестирования.

Думайте о цикломатической сложности как о количестве путей прохождения фрагмента кода.

Даже простейшая часть кода предоставляет единственный путь, поэтому минимальная цикломатическая сложность равна 1. Вы можете легко «вычислить» цикломатическую сложность метода или функции: начиная с единицы, считать, сколько раз встречается `if` и `for`. Для каждого из этих ключевых слов вы увеличиваете число (от 1).

Специфика подсчета зависит от языка программирования. Идея в том, чтобы подсчитать операторы ветвления и цикла. Например, в языке C# вам придется включать `foreach`, `while`, `do` и каждый `case` в блоке `switch`. В других языках ключевые слова для подсчета будут отличаться.

Какова цикломатическая сложность метода `Post` в системе резервирования столиков в ресторане? Попробуйте сосчитать все операторы ветвления в коде листинга 6.9, начиная с *единицы*.

Что вы получили?

Цикломатическая сложность кода листинга 6.9 равна 7. У вас получилось 6 или 5?

Давайте разберемся, как же получить 7. Не забудьте начать с единицы. Для каждого оператора ветвления увеличьте число на 1. Есть пять операторов `if`. Пять плюс один равно шесть. Последний оператор заметить труднее всего: это `??`, оператор нулевого слияния,

представляющий две альтернативные ветви: одну, где `dto.Name` имеет значение `null`, и другую, где это не так. Это еще один оператор ветвления[1]. Итак, всего метод `Post` содержит семь элементов.

Как я уже упоминал, я использую число семь как предельное значение кратковременной памяти мозга. Если вы определите пороговое значение, равное семи, метод `Post` из листинга 6.9 удовлетворяет ему впритык. Вы могли бы оставить все как есть. Это нормально, но если вам понадобится добавить восьмую ветвь, сначала придется провести рефакторинг. А еще у вас может не быть на это времени в будущем, поэтому если есть время сейчас, то лучше провести профилактический рефакторинг, не откладывая на потом.

Пока не забывайте об этом. В подразделе 7.2.2 мы еще вернемся к методу `Post`, чтобы провести рефакторинг. Но сначала давайте рассмотрим некоторые другие принципы.

7.1.3. Правило 80/24

А как вам идея того, что количество строк кода — это простейший показатель сложности?

Всегда помните об этом. Никогда не пишите длинные методы — пишите небольшие блоки кода.

Как понять, насколько небольшие?

На этот вопрос нет однозначного ответа. Это зависит в том числе и от языка программирования, так как одни языки многословнее других. Самый наполненный язык, на котором я когда-либо программировал, — это APL.

Но большинство основных языков примерно одинаковы в этом вопросе. При создании кода на C# мне становится дискомфортно, когда

[1] Если вы не привыкли считать null-операторы C# операторами ветвления, это может вас не убедить. Но, возможно, вас убедит следующее: встроенный в Visual Studio калькулятор метрик кода тоже в результате дает цикломатическую сложность, равную 7.

написанный мной метод приближается к 20 строкам кода. Но C# — довольно многословный язык, поэтому иногда мне приходится писать большие методы. Мой предел — примерно 30 строк кода.

Так как в любом случае я взял произвольное число, давайте сделаем его 24 по причинам, которые я объясню позже.

Итак, максимальное число строк для метода должно быть 24.

Напомню, что это зависит от конкретного языка. Я бы посчитал 24-строчную функцию `Haskell` или `F#` настолько огромной, что если бы получил ее в качестве пул-реквеста, то сразу же отклонил бы, ссылаясь на размер.

Большинство языков программирования допускают определенную гибкость в компоновке. Например, языки на основе C используют символ `;` в качестве разделителя, что позволяет вам писать в строке более одного оператора:

```
var foo = 32; var bar = foo + 10; Console.WriteLine(bar);
```

Использование `;` может помочь вам обойти правило 24 строк. Но это сводит на нет сам смысл затеи.

Цель написания небольших методов — читабельный код, который уместится в вашей голове. Чем меньше, тем лучше.

Для полноты картины теперь определим максимальную длину строки. Если и есть общепринятый стандарт, то это 80 символов. Я весьма эффективно использовал его на протяжении многих лет.

Ограничение в 80 символов выбрано неслучайно, у него долгая история. Но что насчет 24 строк? Хотя оба числа произвольны, они соответствуют размеру популярного терминала VT100 с разрешением экрана 80×24 символа.

Итак, поле размером 80×24 символа воспроизводит размер старого терминала. Значит ли это, что я предлагаю вам писать программы на терминалах? Нет, разработчики всегда неправильно это понимают. Размер 80×24 должен соответствовать максимальному размеру

метода[1]. Тогда на больших экранах вы сможете видеть несколько небольших методов одновременно. Например, вы сможете просматривать модульный тест и его результаты в конфигурации с разделенным экраном.

Эти размеры произвольны, но мне кажется, что в такой преемственности с прошлым есть что-то принципиально правильное.

Контролировать длину строки вы можете с помощью редакторов кода. В большинстве сред разработки есть возможность рисовать вертикальные линии в окнах редактирования. Можно, например, установить черту на отметке 80 символов.

Если вам интересно, почему код в этой книге написан именно так, то одна из причин — то, что длина строк в коде не превышает 80 символов.

Код в листинге 6.9 не только имеет цикломатическую сложность 7, но и содержит ровно 24 строки. Это еще одна причина для рефакторинга. Он уже у самых пределов, хотя видится мне еще не законченным.

7.2. КОД, КОТОРЫЙ УМЕЩАЕТСЯ В ВАШЕЙ ГОЛОВЕ

Ваш мозг может одновременно отслеживать только семь элементов. Учитывайте этот факт при разработке архитектуры кодовой базы.

7.2.1. Гексагональные цветки

Когда вы анализируете фрагмент кода, ваш мозг пытается интерпретировать его работу. Если нужно отслеживать очень много событий, код перестает быть моментально понятным. Кратковременная память

[1] Еще раз обращу ваше внимание, что это предельное значение выбрано произвольно. Смысл в самом наличии порогового значения [97]. Если для вашей команды подходящим размером блока будет 120 × 40, то в этом нет ничего страшного. Но в примерах из книги я использую размер блока 80 × 24, что больше всего соответствует языку C#.

не воспринимает большое количество информации, поэтому вы должны кропотливо работать над тем, чтобы сохранить структуру кода в долговременной. Легаси-код всегда будет у вас под рукой.

В связи с этим я установлю следующее правило.

> В одном фрагменте кода должно происходить не более семи событий.

Есть несколько способов измерить это. Один из них — использование цикломатической сложности. Вы можете схематически изобразить объем своей кратковременной памяти (рис. 7.3).

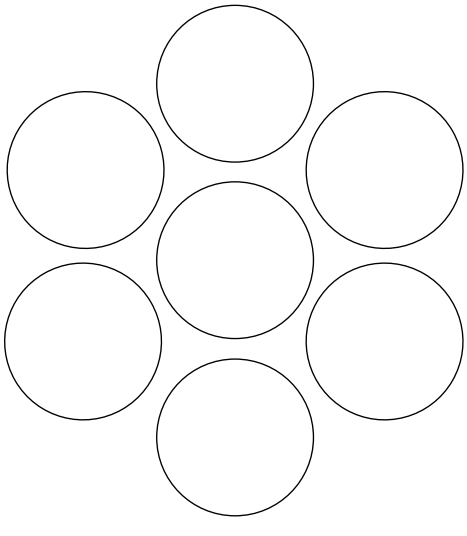


Рис. 7.3. Емкость кратковременной памяти человека изображена в виде семи «регистров»

Думайте о каждом из этих кругов как о слоте памяти или регистре. Каждый круг может содержать только один фрагмент информации [80].

Если вы сожмете эти круги вместе и представите, что они вписаны в другие круги, то наиболее компактное представление будет таким, как на рис. 7.4.

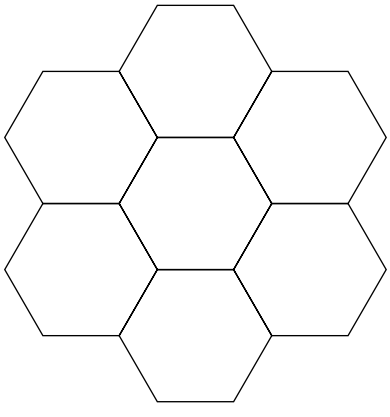


Рис. 7.4. Семь «регистров» в компактной форме. Поскольку шестиугольники можно расположить так в бесконечной сетке, эта форма похожа на стилизованный цветок. Поэтому такие диаграммы я буду называть гексагональными цветками

Задумка в том, что вы должны быть способны описать происходящее в фрагменте кода, заполнив семь шестиугольников на этом рисунке. Каким будет их содержимое для кода в листинге 6.9?

Например, как на рис. 7.5.

В каждой ячейке я отобразил результат, связанный с ветвью кода. Из метрики цикломатической сложности вы знаете, что у листинга 6.9 семь путей прохождения кода — ими я заполнил шестиугольники.

Все «слоты» заполнены. Рассматривая семь как жесткое ограничение[1], вы не сможете усложнить метод `Post`. Проблема в том, что в буду-

[1] Число семь на самом деле не является жестким ограничением. В действительности ничто здесь не опирается на него, но, с другой стороны, это хорошая визуализация.

щем вам придется добавить более сложное поведение. Например, вы можете решить отклонить все прошлые резервирования. Кроме того, бизнес-правило работает только для определенных заведений с общими столиками и отдельными креслами. Более сложная система резервирования должна уметь обрабатывать столики разных размеров, наличие дополнительных мест и т. д.

Рис. 7.5. Диаграмма ветвей метода Post (код приведен в листинге 6.9)

Для дальнейшей работы нужно декомпозировать метод `Post`.

7.2.2. Связность

Как или где декомпозировать метод `Post` (см. листинг 6.9)?

То, что код уже разбит на части несколькими пустыми строками, может вам помочь[1]. У нас есть четыре части. Первая — последовательность Guard Clauses [7] — лучший кандидат на рефакторинг.

[1] В этой книге я не описывал подробно, как и почему вы размещаете общий исходный код и то, как нужно использовать пустые строки. Это уже есть в книге Стива Макконнелла «Совершенный код» [65]. Я использую пустые строки в соответствии с этой книгой.

Почему мы так решили?

В первой части не используются члены-экземпляры владеющего класса `ReservationsController`. Вторая и третья используют свойство `Repository`. Четвертая — это всего лишь одно выражение возврата, поэтому здесь мало что можно улучшить.

То, что вторая и третья части используют член экземпляра, не препятствует их извлечению во вспомогательные методы, но первая часть более заметна. Это относится к центральному понятию ООП — связности. Мне нравится, как об этом сказал Кент Бек:

> *«Те события, которые меняются с одинаковой скоростью, принадлежат друг другу, а те, которые изменяются с разной, должны быть отдельно друг от друга» [8].*

Рассмотрим, как используются поля экземпляра класса. Максимальная связность — это когда все методы задействуют все поля класса, а минимальная — когда каждый метод использует собственный непересекающийся набор полей класса.

Блоки кода, в которых вообще не используются поля классов, выглядят еще более подозрительно. Вот почему я считаю, что лучший кандидат на рефакторинг — это первая часть кода.

Ваша первая попытка может напоминать код листинга 7.1. Этот небольшой метод имеет всего шесть строк кода и цикломатическую сложность 3. Согласно метрикам, которые мы обсуждали, он написан хорошо.

Обратите внимание, что он помечен как статический. Это важно, так как правило анализатора кода[1] обнаружило, что не используются члены экземпляра. Это может быть плохой код. Позднее мы еще вернемся к этой теме.

Улучшает ли введение вспомогательного метода `IsValid` метод `Post`? В листинге 7.2 показан результат.

---

[1] CA1822: помечайте члены как статические.

Листинг 7.1. Вспомогательный метод для определения валидности dto резервирования (Restaurant/f8d1210/Restaurant.RestApi/ReservationsController.cs)

```
private static bool IsValid(ReservationDto dto)
{
    return DateTime.TryParse(dto.At, out _)
        && !(dto.Email is null)
        && 0 < dto.Quantity;
}
```

Листинг 7.2. Метод Post, использующий новый вспомогательный метод IsValid (Restaurant/f8d1210/Restaurant.RestApi/ReservationsController.cs)

```
public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));
    if (!IsValid(dto))
        return new BadRequestResult();

    var d = DateTime.Parse(dto.At!, CultureInfo.InvariantCulture);

    var reservations =
        await Repository.ReadReservations(d).ConfigureAwait(false);
    int reservedSeats = reservations.Sum(r => r.Quantity);
    if (10 < reservedSeats + dto.Quantity)
        return new StatusCodeResult(
            StatusCodes.Status500InternalServerError);

    var r =
        new Reservation(d, dto.Email!, dto.Name ?? "", dto.Quantity);
    await Repository.Create(r).ConfigureAwait(false);

    return new NoContentResult();
}
```

На первый взгляд это выглядит как улучшение. Количество строк сокращено до 22, а цикломатическая сложность — до 5.

Вас удивило, что цикломатическая сложность уменьшилась?

В конце концов, комбинированное поведение метода `Post` и его вспомогательного метода `IsValid` не изменилось. Разве мы не должны учитывать сложность `IsValid` в сложности метода `Post`?

Закономерный вопрос, но это так не работает. Такой способ представления вызова метода может быть как негативным сценарием, так и позитивным. Если нужно отслеживать детали поведения `IsValid`, то ничего не получится. С другой стороны, вы можете рассматривать это как одну операцию, и тогда диаграмма гексагонального цветка (рис. 7.6) будет выглядеть гораздо лучше.

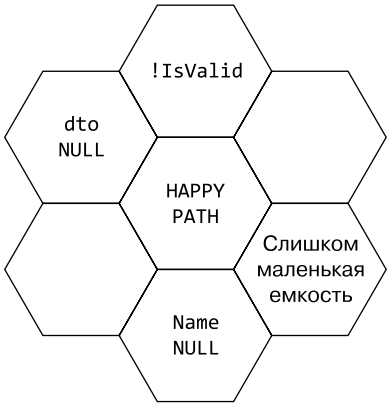


Рис. 7.6. Диаграмма гексагонального цветка, отображающая сложность кода листинга 7.2. Два пустых «регистра» — это дополнительные возможности вашей кратковременной памяти. Другими словами, код укладывается в голове

Три мелких участка были заменены одним немного большего размера.

> *«Кратковременная память измеряется участками [...] так как каждый элемент может быть меткой, указывающей на гораздо большую информационную структуру в долговременной памяти» [80].*

Ключ к такой замене — возможность заменить многие вещи одной. Вы можете сделать это, если *абстрагируете суть*. Звучит знакомо?

Это определение абстракции Роберта Мартина:

> *«Абстракция — это устранение неважного и усиление существенного» [60].*

Метод `IsValid` уточняет, что проверяет объект передачи данных, но исключает точные сведения о том, как он это делает. Изобразим для него другую гексагональную структуру кратковременной памяти (рис. 7.7).

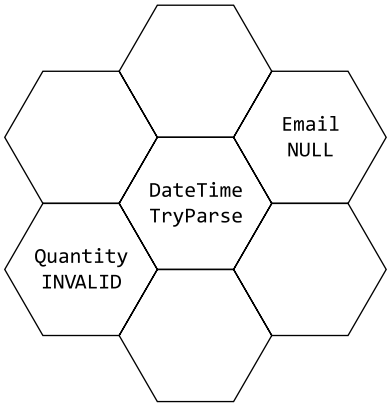


Рис. 7.7. Диаграмма гексагонального цветка, отображающая сложность кода листинга 7.1

Когда вы анализируете код `IsValid`, вам не нужно ничего знать о его контексте. Вызывающий код не влияет на метод `IsValid`, если не считать передачу ему аргумента. И `IsValid`, и `Post` укладываются в вашей голове.

7.2.3. «Завистливые функции»

Хотя сложность описанного выше рефакторинга уменьшилась, это изменение привело к другим проблемам.

Основная проблема здесь — «попахивающий» код, в котором метод `IsValid` статический (`static`)[1] — он принимает параметр `ReservationDto`, но не использует члены экземпляра класса `ReservationsController`. Это пример кода с запашком «завистливой

[1] Статический метод — это не всегда проблема, но в ООП он может ею стать. Обратите внимание на то, как вы используете static.

функции» [34]. Как предполагает рефакторинг [34], попробуйте перенести метод в объект, к которому он «завистлив».

В листинге 7.3 можно увидеть перемещение метода в класс `ReservationDto`. Я решил пока оставить его внутренним, но, возможно, позже я изменю свое решение.

Листинг 7.3. Метод IsValid перемещен в класс ReservationDto (Restaurant/0551970/Restaurant.RestApi/ReservationDto.cs)

```
internal bool IsValid
{
    get
    {
        return DateTime.TryParse(At, out _)
            && !(Email is null)
            && 0 < Quantity;
    }
}
```

Я решил реализовать элемент как свойство[1] вместо метода. Предыдущий метод «завистлив» к функциям `ReservationDto`, но теперь, когда член — часть этого класса, ему не нужны дополнительные параметры. Это мог быть метод, не принимающий никаких входных данных, но сейчас лучше выбрать свойство.

Это простая операция без предварительных условий, и она не может генерировать исключения, что соответствует правилам .NET для геттеров свойств [23].

В листинге 7.4 видно, как метод `Post` проверяет валидность `dto`.

Листинг 7.4. Фрагмент кода из метода Post. Здесь он вызывает метод IsValid из листинга 7.3 (Restaurant/0551970/Restaurant.RestApi/ReservationsController.cs)

```
if (!dto.IsValid)
    return new BadRequestResult();
```

Все тесты проходят успешно. Не забудьте сделать коммит изменений в Git и рассмотрите их пуск через конвейер развертывания [49].

---

[1] Свойство — это просто синтаксический сахар C# для геттера (и/или сеттера).

7.2.4. Потери при передаче

Даже в небольшом блоке кода может быть достаточно проблем, и устранение одной не гарантирует, что их больше не будет. То же самое и с методом `Post`.

Компилятор C# больше не может определить, что `At` и `Email` гарантированно не `null`. Вам нужно, чтобы он отключил статический анализ потока для этих ссылок с помощью оператора `!`, иначе код не будет скомпилирован. По сути, вы подавляете функцию компилятора `nullable-reference-types` (ссылочные типы, допускающие значение `null`). Это неверный шаг.

Еще одна проблема кода из листинга 7.2 в том, что код дважды эффективно анализирует свойство `At` — один раз в методе `IsValid` и еще раз в `Post`.

Кажется, слишком много теряется при передаче. Виной всему `IsValid` — он не может быть хорошей абстракцией, так как слишком много устраняет и слишком мало усиливает.

Это типичная проблема ООП. Такой элемент, как `IsValid`, выдает логический флаг, а не всю информацию, важную для нисходящего кода, например дату. Это заставляет другой код повторять проверку. Результат — дублирование кода.

Лучшая альтернатива — сбор валидных данных.

Как вы представляете себе валидные данные?

В главе 5 мы уже обсуждали, что такое инкапсуляция. Объекты должны защищать свои инварианты. Этот процесс включает в себя пред- и постусловия. Правильно инициализированный объект гарантированно будет в валидном состоянии — если это не так, инкапсуляция нарушается, так как конструктору не удалось проверить предварительное условие.

Это мотивация для доменной модели. Классы, моделирующие домен, должны фиксировать его инварианты. Это контрастирует с объектами передачи данных, которые моделируют беспорядочные взаимодействия с остальным миром.

В системе бронирования для ресторанов модель предметной области валидного резервирования уже есть — это класс `Reservation` из листинга 5.13.

7.2.5. Анализ вместо валидации

Вместо `IsValid`, который возвращает логическое значение, транслируйте объекты передачи данных [33] в объекты предметной области, если выполняются предварительные условия. Рассмотрим пример кода из листинга 7.5.

Листинг 7.5. Метод Validate возвращает инкапсулированный объект (Restaurant/a0c39e2/Restaurant.RestApi/ReservationDto.cs)

```
internal Reservation? Validate()
{
    if (!DateTime.TryParse(At, out var d))
        return null;
    if (Email is null)
        return null;
    if (Quantity < 1)
        return null;

    return new Reservation(d, Email, Name ?? "", Quantity);
}
```

Метод `Validate` использует Guard Clauses [7] для проверки предварительных условий класса `Reservation`. Этот процесс включает в себя парсинг строки `At` в соответствующее значение `DateTime`. Только если все предварительные условия выполнены, он возвращает объект `Reservation`, иначе возвращается `null`.

Тип Maybe

Обратите внимание на сигнатуру метода `Validate`:

```
internal Reservation? Validate()
```

Имя и тип метода — это первое, что вы видите, читая незнакомый код. Если в сигнатуре можно уловить суть метода, то это хорошая абстракция.

Тип возвращаемого значения метода `Validate` несет важную информацию. Напомню, что вопросительный знак указывает на то, что объект может быть `null`. Это важно при написании кода, вызывающего метод. Кроме того, при включенной функции ссылочных типов C#, допускающих значение `null`, компилятор будет выдавать предупреждения, если вы забудете обработать появление `null`.

Это относительно новый функционал в области ООП. В прошлых версиях C# все объекты всегда могли допускать значение `null`. То же самое верно и для других объектно-ориентированных языков, таких как Java.

С другой стороны, некоторые языки (например, Haskell) не имеют нулевых ссылок или делают все возможное, чтобы притвориться, что их нет (F#).

Вы все еще можете моделировать наличие и отсутствие значений в этих языках. Это можно делать явно с помощью типа `Maybe` (в Haskell) или `Option` (в F#). Вы можете легко перенести это представление на более ранние версии C# или другие объектно-ориентированные языки. Все, что вам нужно, — это полиморфизм и (желательно) дженерики [94].

Сделав это, вы могли бы смоделировать метод `Validate` так:

```
internal Maybe<Reservation> Validate()
```

В зависимости от принципа работы API Maybe вызывающая сторона будет вынуждена обрабатывать оба случая: без резервирования или только с одним. До появления в C# 8 ссылочных типов, допускающих значение `null`, я обучал организации использовать вместо `null` объекты Maybe. Разработчики быстро понимают, насколько безопаснее становится их код.

Если вы не можете использовать функцию C# ссылочных типов, допускающих значение `null`, объявите нулевые ссылки недопустимыми возвращаемыми значениями и вместо этого используйте контейнер Maybe, если хотите указать на возможное отсутствие возвращаемого значения.

Вызывающий код должен проверить, является ли возвращаемое значение нулевым, и действовать соответствующим образом. В листинге 7.6 показано, как метод `Post` обрабатывает нулевое значение.

Листинг 7.6. Метод Post вызывает метод Validate для dto и ответвляется в зависимости от того, является ли возвращаемое значение null (Restaurant/a0c39e2/Restaurant.RestApi/ReservationsController.cs)

```
public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));

    Reservation? r = dto.Validate();
    if (r is null)
        return new BadRequestResult();

    var reservations = await Repository
        .ReadReservations(r.At)
        .ConfigureAwait(false);
    int reservedSeats = reservations.Sum(r => r.Quantity);
    if (10 < reservedSeats + r.Quantity)
        return new StatusCodeResult(
            StatusCodes.Status500InternalServerError);

    await Repository.Create(r).ConfigureAwait(false);

    return new NoContentResult();
}
```

Это решает все проблемы, связанные со статическим методом `IsValid` из листинга 7.1. Метод `Post` не должен подавлять статический анализатор потока компилятора и дублировать парсинг даты.

Цикломатическая сложность `Post` теперь снижена до 4 — это укладывается в вашей голове (рис. 7.8).

Метод `Validate` считается лучшей абстракцией — он усиливает существенное, например представляет ли `dto` валидное резервирование. Это достигается за счет проецирования входных данных в более сильное представление тех же данных.

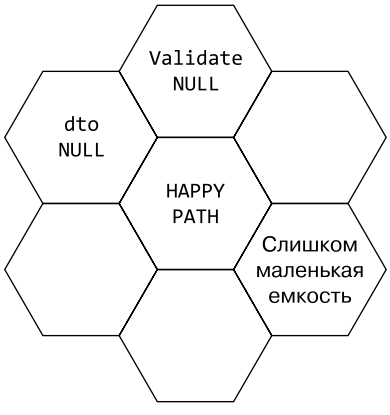


Рис. 7.8. Диаграмма гексагонального цветка для метода Post из листинга 7.6

Алексис Кинг называет эту технику *«анализ вместо валидации»*:

> *«Что такое парсер? На самом деле это просто функция, которая получает менее структурированный ввод и выдает более структурированный вывод. По своей природе парсер — частичная функция (некоторые доменные значения не соответствуют ни одному значению в диапазоне), поэтому все анализаторы должны иметь некоторое представление о сбое. Иногда на вход парсеру поступает текст, но это вовсе не обязательно» [54].*

`Validate` здесь — это парсер: в качестве входных данных он принимает менее структурированный `ReservationDto` и создает более структурированный `Reservation` в качестве выходных. `Validate` следовало бы назвать `Parse`, но я опасался, что это может сбить с толку читателей, которые плохо знакомы с парсингом.

7.2.6. Фрактальная архитектура

Рассмотрим диаграмму, подобную изображенной на рис. 7.8, где есть метод `Post`: только четыре из семи слотов заняты.

Но вы знаете, что фрагмент, представляющий метод `Validate`, усиливает существенное и несколько устраняет сложность. И хотя вам не нужно думать о скрытой за фрагментом сложности, она все еще есть (рис. 7.9).

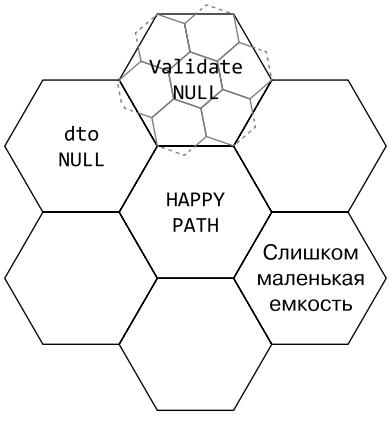


Рис. 7.9. Диаграмма гексагонального цветка, предполагающая, что каждый фрагмент может скрывать другую сложность

Вы можете увеличить масштаб и детальнее рассмотреть фрагмент `Validate`. На рис. 7.10 показано, что структура остается та же.

Цикломатическая сложность метода `Validate` — 5. Если вы используете это как критерий сложности, лучше заполнить пять из семи слотов фрагментами.

Теперь стало заметно, что при увеличении масштаба участок будет иметь ту же форму цветка, что и вызывающая сторона. Что произойдет при уменьшении масштаба?

`Post` не имеет прямых вызывающих объектов. Платформа ASP.NET вызывает методы `Controller` на основе конфигурации в классе `Startup`. Как изменился этот класс?

С написания кода листинга 4.20 класс не менялся. Цикломатическая сложность *всего класса* равна 5. Ее можно легко изобразить в виде диаграммы гексагонального цветка (рис. 7.11).

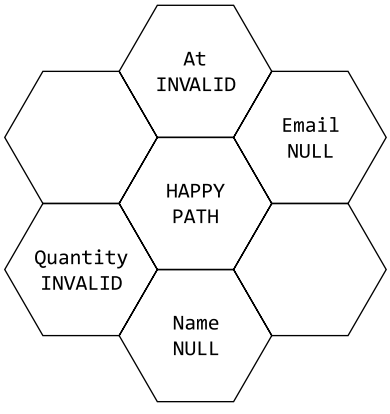


Рис. 7.10. Диаграмма гексагонального цветка метода Validate из листинга 7.5

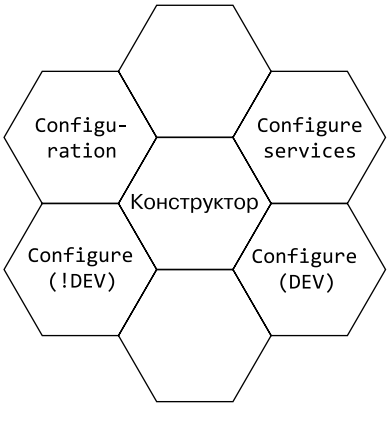


Рис. 7.11. Все элементы сложности класса Startup. Большинство его фрагментов имеют цикломатическую сложность 1, поэтому занимают только один шестиугольник. Цикломатическая сложность метода Configure равна 2, поэтому метод занимает два шестиугольника: один, где IsDevelopment равен true, а другой — false

Даже общее определение приложения умещается в вашей голове. Так и должно оставаться.

Представьте, что вы новый член команды и впервые просматриваете кодовую базу. Если вы пытаетесь понять, как работает приложение, лучше всего будет начать с точки входа. Это класс `Program`, который не изменился с листинга 2.4. Если вы знакомы с ASP.NET, то быстро разберетесь, что ничего неожиданного здесь не происходит. Чтобы понять приложение, нужно проанализировать класс `Startup`.

При анализе `Startup` вы приятно удивитесь, что он укладывается в вашей голове. Анализируя метод `Configure`, вы быстро поймете, что система использует стандартную систему ASP.NET Model View Controller [33] и обычный механизм маршрутизации.

Анализируя метод `ConfigureServices`, вы узнаете, что приложение считывает строку подключения из своей системы конфигурации и использует ее для регистрации объекта `SqlReservationsRepository` в контейнере внедрения зависимостей [25]. Это должно означать, что код использует внедрение зависимостей и реляционную БД.

Это высокоуровневый взгляд на систему: вы не получите никаких подробностей, но узнаете, где их искать при необходимости. Если вы хотите изучить реализацию БД, начните с изучения кода `SqlReservationsRepository`. Если хотите узнать, как обрабатывается конкретный HTTP-запрос, изучите связанный с ним класс `Controller`.

Изучая каждый фрагмент кодовой базы, вы поймете, что каждый класс или метод укладывается у вас в голове на данном уровне абстракции. Вы можете изобразить фрагменты кода в виде гексагонального цветка, примеры которого вы уже видели в этой главе.

Независимо от уровня масштабирования структура сложности не меняется. Это напоминает математические фракталы, что и привело меня к названию фрактальной архитектуры. Важно, чтобы код был простым для понимания на всех уровнях абстракции.

В отличие от математических фракталов вы не можете увеличивать кодовую базу до бесконечности. Рано или поздно вы достигнете наивысшего уровня разрешения, где будут методы, не вызывающие никакой другой ваш код. Например, методы класса `SqlReservationsRepository` (см. листинг 4.19).

Другой способ показать такой стиль архитектуры — это фрактальное дерево, в котором конечные узлы представляют наивысший уровень разрешения (рис. 7.12). На нижнем уровне (в стволе) ваш мозг может обрабатывать до семи фрагментов, представленных семью ответвлениями. На каждой ветви ваш мозг снова может обрабатывать еще семь ветвей и т. д. Математическое фрактальное дерево бесконечно, но рано или поздно все равно придется прекратить отрисовку ветвей.

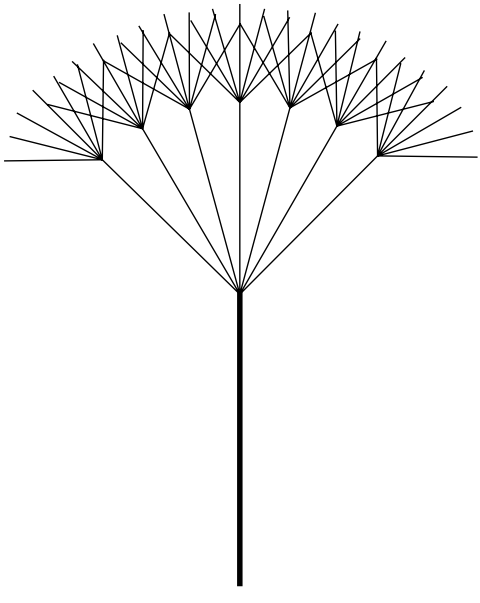


Рис. 7.12. Фрактальное дерево с семью ветвями

Фрактальная архитектура — это способ организовать код так, чтобы он был вам понятен вне зависимости от точки обзора. Низкоуровневые детали должны находиться в одном абстрактном фрагменте, а высокоуровневые должны быть либо неактуальны на этом этапе масштабирования, либо как-то иначе явно отображаться как параметры метода или внедренные зависимости. Помните, что *то, что вы видите, — это все, что здесь есть* [51].

Фрактальная архитектура не возникает сама по себе. Важно учитывать сложность каждого блока кода, который вы пишете. Вы можете вычислять цикломатическую сложность, следить за строками кода или подсчитывать число задействованных в методе переменных. Ваш способ оценки сложности не так важен, как поддержание ее на низком уровне.

Глава 16 знакомит вас с готовой кодовой базой примеров. Завершенная система будет гораздо сложнее того, с чем вы сталкивались до сих пор, но она по-прежнему отвечает требованиям фрактальной архитектуры.

7.2.7. Подсчет переменных

Вы можете проанализировать сложность иначе, подсчитав количество переменных в методе. Иногда я делаю так, чтобы взглянуть на вещи под другим углом.

Но прежде, чем это сделать, убедитесь, что вы считаете все задействованные объекты. Сюда входят локальные переменные, аргументы метода и поля класса.

Например, метод `Post` в листинге 7.6 включает пять переменных: `dto`, `r`, `reservations`, `Repository` и `ReservedSeats`. Три из них — локальные, `dto` — параметр, а `Repository` — свойство (которое поддерживается неявным, автоматически сгенерированным полем класса). Эти пять элементов нужно контролировать, и ваш мозг вполне способен на это.

Часто я применяю такой подход, когда думаю о том, можно ли добавить в метод еще один параметр. Не слишком ли много четырех? Да, эти четыре аргумента находятся в пределах семи, но если они будут взаимодействовать с пятью локальными переменными и тремя полями класса, то происходящих событий будет очень много. Один из выходов из такой ситуации — введение Parameter Object [34].

Очевидно, этот тип анализа сложности не работает с интерфейсами или абстрактными методами, так как нет реализации.

7.3. ЗАКЛЮЧЕНИЕ

Код не становится устаревшим сразу — он деградирует со временем, медленно накапливая мусор, до тех пор пока это не становится заметно.

Как отметили Брайан Фут и Джозеф Йодер, высококачественный код — как нестабильное равновесие:

> *«Хорошая архитектура основывается скорее на понимании движения к цели как непрерывного процесса исследований, а не на понимании самой цели как зафиксированного артефакта» [28].*

Очень важно активно предотвращать деградацию кода. Это можно сделать с помощью контроля его разных показателей, таких как количество строк кода, цикломатическая сложность, или просто подсчета переменных.

Эти метрики отнюдь не универсальны. Они могут быть полезными проводниками, но используйте их по своему усмотрению. Просто помните, что такие метрики мониторинга могут повысить вашу осведомленность о деградации кода.

Сочетая метрики с агрессивными пороговыми значениями, вы привыкаете обращать внимание на качество кода. Вы начинаете понимать, когда разбить блок кода на более мелкие компоненты.

Метрики сложности не дают вам понимания того, какие фрагменты нужно декомпозировать. На эту тему написано уже очень много книг [60; 27; 34], но есть несколько моментов, на которые стоит обратить внимание, — это связность, функциональная «завистливость» и валидация.

Вы должны стремиться к такой архитектуре кодовой базы, чтобы анализируемый код легко укладывался в вашей голове независимо от точки обзора. На высоком уровне происходит семь или менее событий, которые нужно отслеживать, на низком — не более семи. Все это остается в силе и на промежуточном уровне.

На каждом уровне масштабирования сложность кода остается в гуманных пределах. Такое самоподобие на разных уровнях разрешения достаточно похоже на фракталы, и я называю это *фрактальной архитектурой*.

Это не происходит само по себе, но, если достичь такого результата, код становится значительно проще для понимания, чем легаси-код, так как в основном нужно задействовать кратковременную память.

В главе 16 вы познакомитесь с итоговым вариантом кодовой базы и сможете проанализировать, как концепция фрактальной архитектуры проявляется в реальных условиях.

8 ПРОЕКТИРОВАНИЕ API

Сложные блоки кода нужно декомпозировать (рис. 8.1). В главе 7 обсуждалось, где делить код на части, а в этой вы узнаете, как проектировать новые детали.

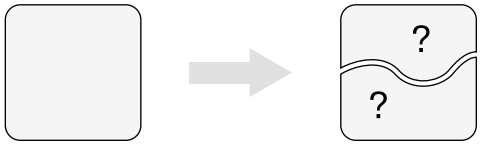


Рис. 8.1. Разделите большой и сложный блок кода на более мелкие. Как должны выглядеть новые блоки? В этой главе вы познакомитесь с некоторыми принципами проектирования API

Вы можете декомпозировать код по-разному — нет единственно правильного способа. Но плохих гораздо больше, чем хороших. Не сойти с правильного пути проектирования API вам помогут ваши навыки и вкус. К счастью, навык можно приобрести. В соответствии с темой этой книги вы можете применить эвристический подход к проектированию API.

8.1. ПРИНЦИПЫ ПРОЕКТИРОВАНИЯ API

Аббревиатура API расшифровывается как *application programming interface* (программный интерфейс приложения). Это определенный набор компонентов, который позволяет одной программе обмениваться данными с другой.

8.1.1. Аффорданс (возможность)

Что для вас значит термин *«интерфейс»*? Вы можете думать о нем как о ключевом слове языка, например как в листинге 6.5. Но в контексте API мы используем его в более широком смысле. Интерфейс — это *возможность*: набор методов, значений, функций и объектов, с помощью которых можно взаимодействовать с другим кодом. При хорошей инкапсуляции интерфейс — это набор операций, сохраняющих инварианты задействованных объектов. Иначе говоря, операции гарантируют, что все состояния объекта валидны.

API позволяет взаимодействовать с инкапсулированным пакетом кода точно так же, как дверная ручка позволяет взаимодействовать с дверью. Для описания таких взаимодействий Дональд А. Норман использует термин *«аффорданс»*:

> *«Термин "аффорданс" относится к взаимодействию между физическим объектом и человеком (или любым вступающим во взаимодействие агентом, будь то животное, человек, машина или робот). Аффорданс — это отношения между свойствами объекта и способностями агента, определяющими, как объект может быть использован. Стул обеспечивает поддержку, поэтому предполагает сидение. Многие стулья перенести с места на место может один человек (они предполагают возможность их поднять), но некоторые можно сдвинуть с места только с помощью сильного человека или группы людей. Если ребенок или сравнительно слабый человек не может поднять стул, то у этого предмета нет аффорданса (стул не предполагает возможность его поднять)» [71].*

Я считаю, что это определение хорошо применимо к дизайну интерфейсов. Такой API, как `IReservationsRepository` (см. листинг 6.5),

позволяет считывать данные о резервировании, относящиеся к определенной дате, и добавлять новые. Вам доступен вызов методов, только если вы можете предоставить необходимые входные аргументы. Взаимодействие между клиентским кодом и API аналогично взаимодействию между вызывающей стороной и инкапсулированным объектом. Объект предоставляет свои возможности только тому клиентскому коду, который удовлетворяет обязательным предварительным условиям. Если `Reservation` нет, вы не можете вызвать `Create`.

Дональд Норман также писал:

> *«Ежедневно мы сталкиваемся с тысячами предметов, многие из которых оказываются для нас в новинку. Многие новые объекты похожи на известные нам, но многие уникальны, и при этом мы всё равно неплохо справляемся. Как же мы это делаем? Почему при встрече с неизведанным объектом мы знаем, как с ним взаимодействовать? Есть несколько базовых принципов, позволяющих нам ответить на этот вопрос. Один из самых важных — рассмотрение возможностей» [71].*

Когда вы впервые видите стул, глядя на его форму, вы сразу понимаете, как можно его использовать. У офисных стульев есть дополнительные возможности: например, регулировка их высоты. У одних стульев найти нужный рычаг легко, у других это сделать сложнее. Но все рычаги выглядят одинаково, и тот, который, как вы думали, должен регулировать высоту, вместо этого будет регулировать угол наклона сиденья.

Как узнать возможности API? Работая с компилируемым, статически типизированным языком, вы можете использовать систему типов. Среда разработки может использовать информацию о типе для отображения операций, доступных для этого объекта (рис. 8.2).

Это обеспечивает определенную степень обнаруживаемости и называется *разработкой, управляемой точками*[1]. Как только вы вводите знак

[1] Впервые я услышал этот термин в 2012 году в Копенгагене в выступлении Фила Трелфорда на конференции GOTO. Я не нашел другого определения этого термина.

точки после объекта, вам предоставляется выбор методов, которые вы можете вызывать.

Рис. 8.2. IDE может отображать доступные методы для объекта сразу по мере ввода. В Visual Studio это называется IntelliSense

8.1.2. Poka-Yoke, или «защита от ошибок»

Пример распространенной ошибки — конструкция швейцарского армейского ножа. Я знаком с разработчиками, полагающими, что хороший API позволяет выполнять как можно больше действий. Подобно швейцарскому армейскому ножу, он может иметь множество возможностей, но ни одна из них не подойдет для достижения цели так, как обычный специализированный инструмент (рис. 8.3). Итак, мы близки к понятию God Class («класс Бога»)[1].

Хороший дизайн интерфейса учитывает не только то, что возможно, но и то, что должно быть заведомо невозможным — аффордансы. Элементы, предоставляемые API, указывают на возможности, но непредоставленные операции сообщают о том, что вы не должны делать.

[1] God Class [15] — антипаттерн ООП, описывающий объект, который хранит в себе «слишком много» или делает «слишком много». Другими словами, описывающий классы, содержащие десятки элементов и реализованные в тысячах строк кода в одном файле.

Рис. 8.3. Швейцарский армейский нож может пригодиться в безвыходной ситуации, но он не заменит нам обычные инструменты

Создавайте API так, чтобы их было трудно использовать не по назначению. Одна из основных составляющих хорошей разработки ПО — *обеспечение качества* [82], то есть защита от ошибок. В бережливом производстве этот процесс известен под японским термином *poka-yoke*, что означает «защита от ошибок». Он широко используется в программной инженерии [1].

Метод Poka-Yoke бывает двух видов: активный и пассивный. Активная защита от ошибок включает проверку новых артефактов при первом их появлении. Яркий пример — разработка через тестирование [1], где все время нужно запускать автоматические тесты.

Но нам больше интересна идея пассивной защиты от ошибок. В физическом мире можно найти много примеров этому процессу. Кабельные разъемы, такие как USB и HDMI, можно использовать только по назначению, барьеры с ограничителем высоты (рис. 8.4) предупреждают водителей о том, что здесь их транспортное средство не сможет проехать. Такие системы не требуют активной проверки.

Проектировать API нужно точно так же: *чтобы недопустимые действия были невозможными для выполнения* [69]. Если действие недопустимо, лучше спроектировать API так, чтобы его невозможно было выразить в коде. Зафиксируйте отсутствие возможности в дизайне

API, чтобы то, что должно быть невозможно, даже не компилировалось[1]. Ошибка компилятора дает более быструю обратную связь, чем исключение во время выполнения [99].

Рис. 8.4. Барьер с ограничителем высоты. Слишком высокий грузовик не сможет проехать, так как врежется в ограничитель, но с небольшим ущербом

8.1.3. Пишите для читателей

Вспомните времена, когда вы писали сочинения в школе. Учитель наверняка всегда настаивал на том, чтобы вы учитывали *тему*, того, кто *пишет*, и того, кто будет *читать*. В электронной переписке это будут *отправитель* и *получатель* соответственно.

[1] Это просто сделать в языках программирования с типами суммы, например Haskell и F#. В ООП прямой эквивалент — более многословный шаблон проектирования Visitor [107].

Я встречал разработчиков, которые с грустью вспоминали школьные дни и сейчас счастливы, что занимаются разработкой ПО, а литература осталась далеко позади.

Но хочу вас расстроить.

Литература по-прежнему актуальна в вашей профессиональной жизни. В школе обучают подобным навыкам не просто так. Отправитель и получатель важны при составлении электронного письма, разработке документации и, что удивительно, при написании кода.

Код больше читают, чем пишут.

Пишите код для будущих специалистов, для тех, кто будет его читать. Однажды этим специалистом можете стать и вы.

8.1.4. Предпочитайте комментариям хорошо написанный код

Вы, наверное, слышали, что вместо добавления комментариев лучше писать чистый код [61]. По мере развития кодовой базы комментарии могут стать бесполезными. Даже правильный комментарий со временем становится обманчивым. В конечном счете единственный артефакт, которому можно доверять, — это код. Доверяйте не комментариям в нем, а актуальным операторам и выражениям, скомпилированным в работающее ПО (листинг 8.1).

Листинг 8.1. Комментарий, поясняющий назначение кода. Не делайте так. Замените его верно названным методом, как в листинге 8.2 (Restaurant/81b3348/Restaurant.RestApi/MaitreD.cs)

```
// Отказ от резервирования, если оно выпадает в нерабочее
// время ресторана
if (candidate.At.TimeOfDay < OpensAt ||
    LastSeating < candidate.At.TimeOfDay)
    return false;
```

По возможности замените комментарий на метод с полезным и понятным именем [61] (см. листинг 8.2).

Листинг 8.2. Название вызывающего метода заменяет комментарий. Сравните с листингом 8.1 (Restaurant/f3cd960/Restaurant.RestApi/MaitreD.cs)

```
if (IsOutsideOfOpeningHours(candidate))
  return false;
```

Не все комментарии бесполезны [61], но лучше использовать хорошо названные методы.

8.1.5. Исключение имен

Никогда не останавливайтесь на достигнутом. Комментарии со временем могут терять свою актуальность — то же самое может произойти и с именами методов. Надеюсь, вы все-таки уделяете им достаточно внимания, но все же случается, что при изменении реализации метода имя не обновляется.

К счастью, в языке со статической типизацией вы можете использовать типы. Создавайте API так, чтобы они определяли свои контракты с типами. Рассмотрим обновленную версию `IReservationsRepository` (листинг 8.3). В ней есть третий метод `ReadReservation`. Это описательное имя. Но достаточно ли оно самодокументировано?

Вопрос, которым я часто задаюсь, исследуя незнакомый API: должен ли я проверять возвращаемое значение на `null`? Как бы вы сообщили об этом устойчиво и последовательно?

Можете попытаться взаимодействовать с помощью описательных имен. Например, присвоив методу имя `GetReservationOrNull`. Это сработает, но не застрахует вас от изменений в поведении. Позже вы можете решить изменить дизайн API, чтобы `null` больше не было допустимым возвращаемым значением, но забудете изменить имя.

Листинг 8.3. IReservationsRepository с дополнительным методом ReadReservation (Restaurant/ee3c786/Restaurant.RestApi/IReservationsRepository.cs)

```
public interface IReservationsRepository
{
    Task Create(Reservation reservation);
```

```
    Task<IReadOnlyCollection<Reservation>> ReadReservations(
        DateTime dateTime);

    Task<Reservation?> ReadReservation(Guid id);
}
```

Обратите внимание, что благодаря встроенному в C# функционалу ссылочных типов, допускающих значение `null`, эта информация уже включена в сигнатуру типа метода[1]. Тип возвращаемого значения — `Task<Reservation?>`. Напомню, что вопросительный знак указывает на то, что объект `Reservation` может принимать значение `null`.

Теперь попробуйте исключить имена методов и попытайтесь понять, что они делают:

```
public interface IReservationsRepository
{
    Task Xxx(Reservation reservation);
    Task<IReadOnlyCollection<Reservation>> Xxx(DateTime dateTime);
    Task<Reservation?> Xxx(Guid id);
}
```

Что обозначает запись `Task Xxx(Reservation reservation)`? Объект `Reservation` принимается в качестве входных данных, но ничего не возвращает[2]. Поскольку возвращаемого значения нет, какое действие он должен выполнять?

Возможно, он сохраняет бронь. Он также может преобразовать ее в электронное письмо и отправить или может записывать информацию в лог. Когда вы знаете, что объект, определяющий метод, называется `IReservationsRepository`, то подразумеваемый контекст — сохраняемость. Это позволяет исключить ведение журнала и отправку по электронной почте в качестве альтернативы.

---

[1] Если выбранный вами язык программирования не делает явным различие между ссылочными типами, допускающими и не допускающими нулевое значение, можете вместо этого использовать концепцию Maybe, описанную во врезке «Тип Maybe» в главе 7. В этом случае сигнатура метода ReadReservation будет выглядеть так: Task<Maybe<Reservation>> ReadReservation(Guid id).

[2] Строго говоря, возвращается объект Task без дополнительных данных. Представьте, что Task — это асинхронный эквивалент void.

Но все еще неясно, создает ли этот метод новую запись в БД или обновляет существующую. Может, даже он выполняет два действия. Технически возможно и то, что он удаляет запись, хотя лучшим вариантом имени для операции удаления будет `Task Xxx(Guid id)`.

Теперь давайте выясним, что выполняет метод `Task<IReadOnlyCollection<Reservation>> Xxx(DateTime dateTime)`. Он принимает дату в качестве входных данных и возвращает коллекцию резервирований в качестве выходных. Поэтому несложно догадаться, что это запрос, основанный на дате.

Наконец, метод `Task<Reservation?> Xxx(Guid id)` принимает ID в качестве входных данных и может возвращать или не возвращать одно резервирование. Это однозначно поиск на основе идентификатора.

Этот метод работает до тех пор, пока объекты допускают лишь небольшое количество взаимодействий. Пример кода содержит всего три члена, и все они разных типов. Когда вы объединяете сигнатуры методов с именем класса или интерфейса, вы можете догадаться, что делает метод.

Обратите внимание, что больше всего времени уйдет на рассуждения о методе `Create`. Поскольку возвращаемого типа фактически нет, вам придется думать о его намерениях, основываясь только на типе ввода, в то время как при рассмотрении запросов у вас есть как типы ввода, так и типы вывода, чтобы намекнуть на намерение.

Скрывайте имена методов за знаком `x` — это поможет вам встать на место будущего читателя кода, а следовательно, и облегчить для них понимание написанного. Вы можете полагать, что придумали описательное и полезное имя, но в другом контексте оно может оказаться бессмысленным.

Имена по-прежнему важны, но лучше не повторять то, что уже указано в типах. Это дает вам возможность сообщить читателю что-то, чего он не может определить по типам.

Обратите внимание на важность «поддержания остроты лезвия вашего инструмента». Это еще одна причина выбирать специализированные API вместо «швейцарского армейского ножа». Когда объект

предоставляет только три-четыре метода, тип каждого из них обычно отличается от других методов в этом контексте. Когда у вас есть десятки методов для одного и того же объекта, это вряд ли будет работать хорошо.

Типы методов будут полезны, когда только они могут устранить неоднозначность их отличия друг от друга. Если все методы возвращают строку или целое число, их типы вряд ли будут полезны. Это еще одна причина избегать API со строковой типизацией [3].

8.1.6. Command Query Separation (CQS), или разделение команд и запросов

Когда вы исключаете имена, на первый план выходят статические типы. Давайте рассмотрим сигнатуру метода `void Xxx()`. Она почти ничего не говорит вам о том, что делает этот метод. Мы можем сказать лишь то, что он должен выполнять какое-то побочное действие. Он ничего не возвращает, но для чего-то ведь он нужен?

Понятно, что по имени метода легче догадаться, что он делает. Его можно назвать `void MoveToNextHoliday()`, или `void Repaint()`, или вообще как угодно.

При такой структуре метода, как `void Xxx()`, единственный способ общаться с читателем — это выбрать хорошее имя. По мере добавления типов вы получаете больше вариантов дизайна. Рассмотрим запись типа `void Xxx(Email x)`. До сих пор точно не ясно, какую роль выполняет аргумент `Email`, но здесь должно быть заложено какое-то побочное действие. Что бы это могло быть?

Очевидно, раз мы говорим об электронной почте, то это может быть ее отправка. Но не все так однозначно. Метод может еще и удалить электронное письмо.

Что есть побочное действие? Оно изменяет состояние чего-либо. Это может быть локальный эффект — например, изменение состояния объекта; или глобальный — изменение состояния приложения в целом. Он может включать удаление строки из базы, редактирование

файла на диске, перерисовку графического пользовательского интерфейса или отправку электронной почты.

Цель хорошего дизайна API — упорядочить код так, чтобы он был максимально прост для понимания. Напомню, что цель инкапсуляции — скрыть детали реализации. Так, код, реализующий метод, может использовать локальные изменения состояния, и вы не должны считать их побочными эффектами. Рассмотрим вспомогательный метод из листинга 8.4.

Листинг 8.4. Метод с локальным изменением состояния, но без очевидных побочных эффектов (Restaurant/9c134dc/Restaurant.RestApi/MaitreD.cs)

```
private IEnumerable<Table> Allocate(
    IEnumerable<Reservation> reservations)
{
    List<Table> availableTables = Tables.ToList();
    foreach (var r in reservations)
    {
        var table = availableTables.Find(t => t.Fits(r.Quantity));
        if (table is { })
        {
            availableTables.Remove(table);
            if (table.IsCommunal)
                availableTables.Add(table.Reserve(r.Quantity));
        }
    }

    return availableTables;
}
```

Этот метод создает локальную переменную `availableTables`, которую модифицирует перед возвратом. Вы можете подумать, что это и будет побочный эффект, поскольку изменяется состояние доступных таблиц. С другой стороны, метод `Allocate` не изменяет состояние определяющего его объекта и возвращает `availableTables` как доступную только для чтения коллекцию[1].

[1] IEnumerable<T> — это стандартная реализация .NET паттерна проектирования Iterator [39].

Когда вы пишете код, вызывающий метод `Allocate`, все, что вам нужно знать: если вы предоставите ему коллекцию резервирований, то получите коллекцию таблиц. Что касается вас, вы не будете наблюдать никаких побочных эффектов.

Методы с побочными эффектами не должны возвращать данные. Иначе говоря, их возвращаемый тип должен быть `void`, что делает их простыми для распознавания. Встречая метод, который не возвращает никаких данных, вы будете знать, что *смысл его существования* (от фр. *raison d'être*) — в выполнении побочного действия. Такие методы называются *командами* [67].

Чтобы можно было различать процедуры с побочными эффектами и без них, возвращающие данные методы должны быть без побочных эффектов. Поэтому, когда вы видите сигнатуру такого метода, как `IEnumerable<Table> Allocate(IEnumerable<Reservation> reservations)`, вы должны понимать, что она не выполняет побочных действий, поскольку имеет возвращаемый тип. Такие методы называются *запросами* [67][1].

Гораздо проще рассуждать об API, если разделять команды и запросы. Не возвращайте данные из методов с побочными эффектами и не вызывайте побочные эффекты из методов, возвращающих данные. Следование этому правилу поможет вам различать эти два типа функций, не читая код реализации.

Это и есть принцип CQS[2], или *разделение команд и запросов*. Как и большинство других методов из этой книги, он не автоматический.

---

[1] Внимание: запрос здесь — это не обязательно запрос к базе данных, хотя и может им быть. Различие между командами и запросами было проведено в 1988 году Бертраном Мейером [67]. Тогда реляционные БД не были так распространены, как сейчас, поэтому термин «запрос» не был так тесно связан с командами базы данных, как сегодня.

[2] Будьте внимательны: не путайте CQS с CQRS (Command Query Responsibility Segregation, разделение ответственности команд и запросов). Это архитектурный стиль, который берет свою терминологию от CQS (отсюда сходство аббревиатуры), но развивает понятие гораздо шире.

Компилятор не нуждается в этом правиле[1] и не следует ему, так что ответственность за него лежит на вас. При необходимости вы можете составить чек-лист.

Как вы уже знаете из подраздела 8.1.5, легче рассуждать о запросах, чем о командах, поэтому отдавайте предпочтение именно запросам.

Технически возможно написать метод с побочным эффектом, который возвращает данные. Это не команда и не запрос. Компилятору все равно, но, когда вы следуете разделению команд и запросов, эта комбинация недопустима. Вы всегда можете применить этот принцип, но нужна некоторая практика, прежде чем вы поймете, как решать разные запутанные ситуации[2].

8.1.7. Иерархия коммуникации

Как и комментарии, имена тоже могут стать неактуальными. Есть одно общее правило.

> Не говорите комментарием ничего, что можно сказать именем метода. Не говорите именем метода ничего, что можно сказать типом.

Ниже приведены правила в порядке убывания приоритета.

1. Указывайте разные типы API.
2. Присваивайте методам полезные и понятные имена.
3. Пишите полезные комментарии.
4. Предоставляйте наглядные примеры в виде автотестов.
5. Пишите полезные сообщения при коммите в Git.
6. Создавайте полезную и понятную документацию.

---

[1] Если только компилятором не является Haskell или PureScript.

[2] Самая сложная проблема, с которой обычно сталкиваются разработчики, — это как добавить строку в базу данных и вернуть сгенерированный идентификатор вызывающей стороне. Это тоже можно решить с соблюдением принципа CQS [95].

Типы — это часть процесса компиляции. Если вы ошибетесь с определением типов вашего API, ваш код, скорее всего, не скомпилируется. Ни один из других вариантов коммуникации с читателем не обладает таким качеством.

Понятные имена методов — по-прежнему часть кодовой базы. Вы работаете с ними каждый день, и это очень полезный способ сообщить читателю о ваших намерениях.

Но не всё можно легко передать с помощью правильного присвоения имен — например, *причину*, по которой вы решили написать код определенным образом. Тогда нужно добавлять комментарии [61].

Также у вас могут быть соображения, относящиеся к конкретному изменению, сделанному вами в коде. Они должны быть документированы сообщениями при коммите.

Наконец, на несколько высокоуровневых вопросов лучше всего отвечает документация. К таким вопросам относятся способы настройки среды разработки или общая цель кодовой базы. Вы можете сохранить данные в файле `readme` или другой документации.

Обратите внимание: хотя я и не отвергаю старомодную документацию, но считаю ее наименее эффективным способом взаимодействия с разработчиками. Код никогда не устаревает — это единственный артефакт, который всегда актуален.

8.2. ПРОЕКТИРОВАНИЕ API: ПРИМЕРЫ

Как вы применяете рассмотренные принципы проектирования API к коду? Как их можно использовать для решения нетривиальной задачи? Ответы на эти вопросы дадут примеры из этого раздела.

Логика, реализованная в `ReservationsController`, проста. Рассмотрим код в листинге 7.6. Здесь вместимость ресторана жестко запрограммирована на десять посадочных мест. Правило принятия решения не учитывает количество групп гостей, поэтому подразумевается, что все гости сидят за одним столом. Типичная конфигурация хипстерских ресторанов — барная стойка с видом на кухню.

В коде листинга 7.6 не учитывается и время резервирования. Подразумевается, что есть только одно место в день.

Конечно, я обедал в таких ресторанах, но это редкость. В большинстве заведений более одного стола, и у них могут быть дополнительные места. Если вы забронировали столик на 18:30, у кого-то другого он может быть зарезервирован на 21:00. И так у вас будет два с половиной часа на то, чтобы поужинать.

Система бронирования должна учитывать и часы работы. Если ресторан открывается в 18:00, резервирование на 17:30 должно быть отклонено. Точно так же нужно отклонять и резервирования, поступающие на дату из прошлого.

Конфигурация столов, дополнительные места и время работы должны быть настраиваемыми. Все эти требования достаточно сложны, и вам придется учитывать, остается ли код в рамках ограничений, предложенных в этой книге.

Цикломатическая сложность должна быть равна семи или менее, методы не должны быть слишком большими или включать слишком много переменных.

Вам нужно делегировать это бизнес-решение отдельному объекту.

8.2.1. Класс MaitreD (метрдотель)

Только две строки кода в листинге 7.6 обрабатывают бизнес-логику. Эти строки кода повторяются в листинге 8.5 для внесения ясности.

Листинг 8.5. Единственные две строки кода из листинга 7.6, которые фактически принимают бизнес-решение (Restaurant/a0c39e2/Restaurant.RestApi/ReservationsController.cs)

```
int reservedSeats = reservations.Sum(r => r.Quantity);
if (10 < reservedSeats + r.Quantity)
```

С учетом новых требований решение будет значительно сложнее. Есть смысл определить доменную модель [33]. Как мы назовем класс?

Если вы хотите использовать самый распространенный язык [26], на котором говорят эксперты в предметной области, можете назвать его *maître d'*. В официальных ресторанах maître d' — это *метрдотель*, который наблюдает за гостевой зоной ресторана (в отличие от *шеф-повара*, который управляет кухней).

В обязанности метрдотеля входят еще и прием резервирования с распределением столиков. Добавление класса `MaitreD` выглядит как правильный подход к предметно-ориентированному проектированию [26].

В отличие от предыдущих глав я опущу итеративную разработку и сразу покажу вам результат. Если вам интересно, какие модульные тесты я написал и какие шаги предпринял, все это можно найти в виде коммитов в репозитории Git, который прилагается к книге. Проанализируйте API `MaitreD` в листингах 8.6 и 8.7. К каким выводам вы пришли?

В коде в листингах 8.6 и 8.7 показан только общедоступный API. Я скрыл от вас код реализации. В этом и есть смысл инкапсуляции. Вы должны иметь возможность взаимодействовать с объектами `MaitreD`, не зная деталей реализации. Не так ли?

Листинг 8.6. Конструктор MaitreD. Есть и другая перегрузка, которая принимает массив params (Restaurant/62f3a56/Restaurant.RestApi/MaitreD.cs)

```
public MaitreD(
    TimeOfDay opensAt,
    TimeOfDay lastSeating,
    TimeSpan seatingDuration,
    IEnumerable<Table> tables)
```

Листинг 8.7. Сигнатура метода экземпляра WillAccept в MaitreD (Restaurant/62f3a56/Restaurant.RestApi/MaitreD.cs)

```
public bool WillAccept(
    DateTime now,
    IEnumerable<Reservation> existingReservations,
    Reservation candidate)
```

Как создать новый объект `MaitreD`? При вводе `new MaitreD(`, как только вы введете левую скобку, IDE отобразит то, что нужно для продолжения (рис. 8.5). Вам необходимо предоставить аргументы `opensAt`, `lastSeating`, `seatingDuration` и `tables`, ни один из которых не может быть `null`.

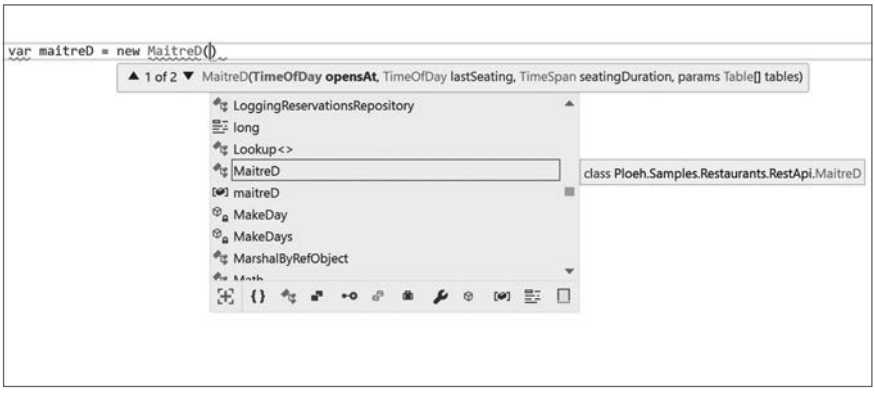


Рис. 8.5. IDE, отображающая требования конструктора на основе информации о статическом типе

Как вы думаете, что здесь нужно сделать? Что поставить вместо `opensAt`? Требуется значение `TimeOfDay` — пользовательский тип, созданный специально для этого. Надеюсь, что правильно назвал его. Если вам интересно, как создавать экземпляры `TimeOfDay`, можете посмотреть его общедоступный API. Точно так же работает и параметр `lastSeating`.

Можете ли вы выяснить, для чего предназначен параметр `seatingDuration`? Надеюсь, что это тоже не вызывает у вас сложностей.

Как вы думаете, для чего нужен параметр таблицы? Вы никогда раньше не были знакомы с классом `Table`, поэтому вам также придется изучить и его общедоступный API. Дальнейшее толкование я намерен пропустить. Суть не в том, что я должен рассказать вам об API, а в том, чтобы дать представление, как о нем рассуждать.

Таким же образом вы можете проанализировать метод `WillAccept` из листинга 8.7. Если я сделал свою работу верно, то пути взаимодействия с ним должны быть ясны. По предоставлению необходимых аргументов метод сообщит вам, примет ли резервирование `candidate`.

Есть ли у него побочные эффекты? Он возвращает значение, поэтому выглядит как запрос, и в соответствии с принципом CQS не должен иметь побочных эффектов. Это действительно так. Значит, вы можете вызывать метод, не беспокоясь о последствиях. Единственное, чего следует ожидать, — он будет использовать несколько циклов CPU и вернет логическое значение.

8.2.2. Взаимодействие с инкапсулированным объектом

Вы должны уметь взаимодействовать с хорошо разработанным API, не зная деталей реализации. Получится ли у вас сделать это с объектом `MaitreD`?

`WillAccept` требует наличия трех аргументов. Сигнатура метода приведена в коде листинга 8.7. Вам понадобится валидный экземпляр класса `MaitreD`, `DateTime`, представляющий настоящее время `now`, коллекция `existingReservations` и потенциальное резервирование `candidate`.

Предполагая, что `ReservationsController` уже имеет валидный объект `MaitreD`, вы можете заменить две строки кода в листинге 8.5 одним вызовом `WillAccept` (листинг 8.8). Несмотря на возросшую сложность всей системы, размер и сложность метода `Post` остаются низкими. Все новое поведение находится в классе `MaitreD`.

Листинг 8.8. Вы можете заменить две строки бизнес-логики из листинга 8.5 одним вызовом WillAccept (Restaurant/62f3a56/Restaurant.RestApi/ReservationsController.cs)

```
if (!MaitreD.WillAccept(DateTime.Now, reservations, r))
```

Метод `Post ReservationsController` использует `DateTime.Now` для предоставления аргумента `now`. Он уже содержит коллекцию

существующих бронирований `reservations` из внедренного репозитория `Repository`, а также проверенное резервирование-кандидат `r` (листинг 7.6). В условном выражении используется логическое отрицание (`!`), поэтому метод `Post` отклоняет резервирование, когда `WillAccept` возвращает значение `false`.

Как определен объект `MaitreD` в листинге 8.8? Это свойство только для чтения, инициализируемое с помощью конструктора `ReservationsController`, приведенного в листинге 8.9.

Листинг 8.9. Конструктор ReservationsController (Restaurant/62f3a56/Restaurant.RestApi/ReservationsController.cs)

```
public ReservationsController(
    IReservationsRepository repository,
    MaitreD maitreD)
{
    Repository = repository;
    MaitreD = maitreD;
}

public IReservationsRepository Repository { get; }
public MaitreD MaitreD { get; }
```

Код похож на Constructor Injection [25], кроме того, что `MaitreD` не является полиморфной зависимостью. Почему я решил поступить именно так? Может ли быть хорошей идеей формальная зависимость от `MaitreD`? Разве это не просто деталь реализации?

Рассмотрим альтернативный пример: передавать все значения конфигурации одно за другим через конструктор `ReservationsController` (листинг 8.10).

Это сразу кажется странным. Конечно, у конструктора `ReservationsController` больше нет общедоступной зависимости от `MaitreD`, но она все еще существует. Если вы измените конструктор `MaitreD`, вам придется изменить и `ReservationsController`. Вариант кода из листинга 8.9 требует меньших затрат на обслуживание, ведь, если вы измените конструктор `MaitreD`, вам нужно будет отредактировать только те фрагменты, где создается внедренный объект `MaitreD`.

Листинг 8.10. Конструктор ReservationsController с развернутыми значениями конфигурации для MaitreD. По сравнению с листингом 8.9 это не кажется лучшей альтернативой (Restaurant/0bb8068/Restaurant.RestApi/ReservationsController.cs)

```
public ReservationsController(
    IReservationsRepository repository,
    TimeOfDay opensAt,
    TimeOfDay lastSeating,
    TimeSpan seatingDuration,
    IEnumerable<Table> tables)
{
    Repository = repository;
    MaitreD =
        new MaitreD(opensAt, lastSeating, seatingDuration, tables);
}
```

Все это происходит в методе `ConfigureServices` класса `Startup` (листинг 8.11). `MaitreD` — неизменяемый класс. Это предусмотрено дизайном. Одно из многих преимуществ такого сервиса без сохранения состояния — то, что он потокобезопасен, так что вы можете зарегистрировать его со временем жизни по шаблону Singleton [25].

Листинг 8.11. Загрузка настроек ресторана из конфигурации приложения и регистрация объекта MaitreD, содержащего эти значения. Метод ToMaitreD приведен в листинге 8.12 (Restaurant/62f3a56/Restaurant.RestApi/Startup.cs)

```
var settings = new Settings.RestaurantSettings();
Configuration.Bind("Restaurant", settings);
services.AddSingleton(settings.ToMaitreD());
```

В листинге 8.12 вы можете увидеть метод `ToMaitreD`. Свойства `OpensAt`, `LastSeating`, `SeatingDuration` и `Tables` принадлежат объекту `RestaurantSettings` с некорректной инкапсуляцией. Из-за того, как работает система конфигурации ASP.NET, вы должны определять объекты конфигурации так, чтобы они могли быть заполнены значениями, считанными из файла. В некотором смысле такие объекты похожи на объекты передачи данных [33] (DTO).

В отличие от DTO, которые поступают в виде документов JSON во время работы службы, вы мало что можете сделать, если парсинг

значений конфигурации завершается сбоем. В этом случае приложение не может быть запущено. Поэтому метод `ToMaitreD` не проверяет значения, которые передает конструктору `MaitreD`. Если они недействительны, конструктор выдаст исключение и приложение завершится ошибкой, оставив запись в логе на сервере.

Листинг 8.12. Метод ToMaitreD преобразует значения, считанные из конфигурации приложения, в объект MaitreD (Restaurant/62f3a56/Restaurant.RestApi/Settings/RestaurantSettings.cs)

```
internal MaitreD ToMaitreD()
{
    return new MaitreD(
        OpensAt,
        LastSeating,
        SeatingDuration,
        Tables.Select(ts => ts.ToTable()));
}
```

8.2.3. Детали реализации

Приятно осознавать, что можно использовать такой класс, как `MaitreD`, не зная всех деталей реализации. Но иногда вашей задачей может быть изменение поведения объекта. Если это так, вам нужно углубиться во фрактальную архитектуру на уровень ниже. Вам придется прочитать код.

В листинге 8.13 приведена реализация метода `WillAccept`. Его цикломатическая сложность равна 5, листинг содержит 20 строк кода, длина каждой строки — до 80 символов, код активизирует семь объектов.

Это не вся реализация. Способ оставаться в рамках легкого в понимании кода — это активно делегировать фрагменты реализации другим фрагментам. Уделите минутку тому, чтобы взглянуть на код и обдумать, поняли ли вы его суть.

Вы еще мало знакомы с классом `Seating` и не знаете, что делает метод `Fits`. Но все же я надеюсь, что у вас есть представление о том, куда

направить свое внимание в зависимости от причины для просмотра кода. Что вы будете делать, чтобы изменить способ размещения таблиц в методе? Что предпримете, если есть ошибка в обнаружении переполнения мест?

Вы можете рассмотреть метод `Allocate` из листинга 8.4. Вы уже видели его раньше. Глядя на этот код, можно забыть о методе `WillAccept`. Использование `Allocate` — это еще одна операция углубления во фрактальную архитектуру. Помните, что *то, что вы видите, — это все, что здесь есть* [51]. То, что вам нужно знать, должно быть прямо здесь, в самом коде.

Листинг 8.13. Метод WillAccept (Restaurant/62f3a56/Restaurant.RestApi/MaitreD.cs)

```
public bool WillAccept(
    DateTime now,
    IEnumerable<Reservation> existingReservations,
    Reservation candidate)
{
    if (existingReservations is null)
        throw new ArgumentNullException(nameof(existingReservations));
    if (candidate is null)
        throw new ArgumentNullException(nameof(candidate));
    if (candidate.At < now)
        return false;
    if (IsOutsideOfOpeningHours(candidate))
        return false;

    var seating = new Seating(SeatingDuration, candidate);
    var relevantReservations =
        existingReservations.Where(seating.Overlaps);
    var availableTables = Allocate(relevantReservations);
    return availableTables.Any(t => t.Fits(candidate.Quantity));
}
```

Метод `Allocate` отлично с этим справляется, активируя шесть объектов. Помимо свойств объекта `Tables`, все они объявлены и используются внутри метода. Это значит, что вам не нужно держать в уме еще какой-то контекст, влияющий на работу метода. Все умещается в вашей голове.

Метод по-прежнему делегирует часть своей реализации другим объектам. Он вызывает `Reserve` для `table`, и снова появляется метод `Fits`. Если вам интересно узнать о нем, можете взглянуть на него в листинге 8.14.

Здесь и близко нет приближения к пределу возможностей нашего мозга, но все же абстрагируются два фрагмента (`Seats` и `quantity`) в один. Это еще одна операция углубления во фрактальную архитектуру. Анализируя исходный код `Fits`, вам нужно отслеживать только `Seats` и `quantity`. Вам не нужно беспокоиться о коде, который *вызывает* этот метод, чтобы понять, как он работает. Все умещается в вашей голове.

Листинг 8.14. Метод Fits. Seats — это целочисленное свойство, доступное только для чтения (Restaurant/62f3a56/Restaurant.RestApi/Table.cs)

```
internal bool Fits(int quantity)
{
  return quantity <= Seats;
}
```

Я не приводил вам в пример метод `Reserve` или класс `Seating`, но они следуют тем же принципам проектирования. Все реализации учитывают наши когнитивные ограничения. Всё это запросы. Если вас интересуют детали реализации, можете обратиться к репозиторию Git, который прилагается к книге.

8.3. ЗАКЛЮЧЕНИЕ

Пишите код для читателей. Как выразился Мартин Фаулер,

> *«кто угодно может написать понятный для компьютера код. Хороший же программист может написать код, понятный человеку» [34].*

Очевидно, что код должен привести к работающей программе. Именно это Фаулер имеет в виду под «кодом, понятным компьютеру». Но это недостаточно высокая планка. Чтобы код был надежным, его нужно писать так, чтобы он был понятен людям.

Инкапсуляция — важная часть процесса, включающего в себя разработку API. Вспомните определение абстракции Роберта Мартина:

> *«Абстракция — это устранение неважного и усиление существенного» [60].*

Детали реализации должны оставаться неактуальными, до тех пор пока вам действительно не понадобится их изменить. Проектируйте API так, чтобы о них можно было рассуждать извне. В этой главе были изучены некоторые фундаментальные принципы проектирования, помогающие продвигать разработку API в этом направлении.

9 КОМАНДНАЯ РАБОТА

По молодости я не очень любил работу в команде. Обычно школьные задания один я выполнял быстрее, чем в группе. Мне казалось, что я всегда в тени своих одноклассников, и мне всегда приходилось отстаивать свою правоту.

Вспоминая об этом, я думаю, что выбирал для себя профессию, предполагающую минимальное взаимодействие с людьми. И, скорее всего, у меня в этом плане много общего с другими программистами.

Но хочу вас огорчить, разработчик ПО довольно редко работает в одиночку.

Обычно вы работаете в команде с другими программистами, владельцами продуктов, менеджерами, техподдержкой, дизайнерами и другими специалистами. Это ничем не отличается от профессии инженера, о которой мы говорили. Все они тоже работают в команде.

Основная обязанность инженера — контроль работы разных процессов. В этой главе я расскажу вам о некоторых полезных процессах разработки ПО. Используйте их, чтобы помочь себе и своим коллегам эффективнее работать с кодом.

Внимание: всегда разграничивайте процесс и результат. Как и в случае с чек-листами, следование процессам значительно повышает шансы на успех. Но наиболее важная часть любого процесса — это понимание лежащей в его основе мотивации. Понимая, почему тот или иной процесс полезен, вы знаете, когда ему следовать, а когда отклоняться от него, ведь важнее всего итоговый результат.

Помните, что результаты могут быть как положительными, так и отрицательными. Как мы уже обсуждали, бессмысленно измерять только прямые результаты ваших действий. Некоторые будут иметь немедленный позитивный эффект в данный момент, но в долговременной перспективе принесут негативный. Например, со временем может накопиться технический долг.

Процесс можно сравнить с работой прокси-сервера: не факт, что все будет хорошо, но это помогает.

9.1. GIT

Большинство организаций по разработке ПО сегодня используют Git вместо других систем контроля версий, таких как CVS или Subversion. Несмотря на то что Git — это распределенная система управления, вы используете его с централизованной службой, такой как GitHub, Azure DevOps Services, Stash, GitLab и т. д.

У таких служб есть дополнительные возможности, например управление рабочими элементами, статистика или автоматическое резервное копирование. Менеджеры часто считают их необходимыми, но не уделяют должного внимания фактическому функционалу управления исходным кодом.

Точно так же большинство знакомых мне программных инженеров работают с Git как с вариантом интеграции своего кода с остальной кодовой базой команды. Они мало задумываются о своем взаимодействии с этой системой.

При таком использовании Git становится второстепенным продуктом, и вы можете упустить некоторые возможности. Старайтесь подходить к работе с Git с умом.

9.1.1. Сообщение коммита

При создании коммита нужно написать сообщение для него. Большинство программистов на этом этапе сталкиваются с определенными сложностями. Вы должны написать хоть *что-нибудь*; и если Git отклоняет пустые сообщения коммита, он примет все остальное.

Разработчики обычно пишут только о том, что есть в коммите. Например: «Имя добавлено», «Нет пустых событий» или «Обработка `CustomerUpdatedAdded`»[1]. Но это не так полезно, как могло бы быть.

Рассмотрим иерархию отношений из раздела 8.1.7. Все, что вы пишете и сохраняете, в будущем пригодится вам и всей вашей команде. С другой стороны, сосредоточьтесь на *общении*, а не на написании. Не тратьте время на объяснение того, что изменилось в коммите. Эту информацию можно просмотреть и с помощью `git diff`.

> Сосредоточьтесь на *общении*, а не на написании.

Верьте или нет, но для сообщений коммитов Git есть стандарт, известный как правило *50/72*, и это не официальный стандарт. Он, скорее, основан на опыте работы с этим инструментом [81]. Стандарт такой:

- пишите резюме в повелительном наклонении и так, чтобы в нем было не более 50 символов;
- при добавлении дополнительного текста оставляйте вторую строку пустой;
- можно добавить любое количество дополнительного текста, но отформатируйте его так, чтобы его объем был не более 72 символов.

[1] Это реальные, слегка анонимизированные примеры.

Эти правила основаны на работе разных функций Git. Например, чтобы увидеть перечень коммитов, можете использовать команду `git log --oneline`:

```
$ git log --oneline
8fa3e47 (HEAD) Make /reservations URL segment lowercase
fbf74ae Return IDs from database in range query
033388a Return 404 Not Found for non-guid id
0f97b34 Return 404 Not Found for absent reservation
ee3c786 Read existing reservation
62f3a56 Introduce TimeOfDay struct
```

Такой список показывает сводную строку для каждого коммита, но не показывает остальную часть сообщения коммита. Даже если вы не используете Git из командной строки, это может сделать кто-то другой. Кроме того, некоторые графические пользовательские интерфейсы, которые делают Git более дружелюбным, на самом деле взаимодействуют с API командной строки Git. Поэтому старайтесь следовать правилу 50/72.

Заголовок играет роль краткого содержания и позволяет вам перемещаться по истории репозитория. Это исключение из правила, согласно которому не нужно объяснять, *что* именно изменилось в коммите.

Писать заголовок в повелительном наклонении — это скорее рекомендации, чем требование. Раньше я писал сообщения коммита в прошедшем времени, следуя правилам форматирования 50/72, и это не доставляло мне проблем. Мне казалось логичным описывать выполненную работу в прошедшем времени, а о правиле использования повелительного наклонения мне никто тогда не говорил. Узнав о нем, я очень неохотно менял свои привычки.

Обычно форма повелительного наклонения короче прошедшего времени. Например, «верните» короче, чем «было возвращено». Это дает вам преимущество при попытке уместить заголовок в 50 символов.

Нет необходимости писать что-то, кроме заголовка, если коммит маленький и самопоясняющий.

Пишите правильно

При написании имейлов, комментариев к коду, ответов на отчеты об ошибках, сообщений об исключениях или сообщений коммитов вам не нужно следовать правилам компилятора или интерпретатора. Просто соблюдайте грамматику.

Многие программисты считают, что все, кроме кода, не имеет никакого значения и можно писать это как угодно. Ниже я привел несколько примеров:

- «Если бы нам нужно было вернуть его в систему контроля версий. но я сомневаюсь, что он вернется, это юридический вопрос». Удивительно, но в этом примере не соблюдена элементарная пунктуация.
- «Спасибо, Пауло, за ваше подстрекательство!» К чему мог подстрекать Пауло? Может, имеется в виду проницательность (insight)? Элементарнейшая ошибка.
- «Чтобы меню открывались одновременно». Что произойдет, если *дерево* или меню *for* будут открыты одновременно?

Если у вас дислексия или вы пишете комментарии не на своем родном языке, то это еще можно понять. Но если это не так, пожалуйста, пишите грамотно.

Бывало ли у вас такое, что вы пытались найти текст в электронной почте, в системе отслеживания заявок или даже в сообщениях коммитов, но ваши поиски не увенчались успехом, даже если вы знали, что искомое точно присутствует? Потратив много времени впустую, вы обнаруживаете, что причина была в том, что ключевое слово написано с ошибкой.

Помимо потраченного времени, неграмотные текстовые сообщения выглядят непрофессионально, замедляют процесс чтения и затрудняют понимание. Может даже сложиться негативное впечатление о вашем профессионализме. Пожалуйста, пишите без ошибок.

Насколько часто коммит поясняет сам себя? Что ж, так же часто, как и код, — реже, чем вы думаете. Если сомневаетесь, добавьте больше контекста.

Различия коммитов (`diff`) уже содержат информацию о том, *что изменилось*. Сам же код — это артефакт, управляющий поведением ПО. Не нужно повторять эту информацию в сообщении коммита.

Хотя вы можете найти ответы на вопросы «что» и «как» в любом месте, сообщение о коммите — лучшее из них для объяснения того, почему было сделано изменение или почему оно приняло такую форму. Вот краткий пример:

```
Ввести структуру TimeOfDay

Это делает роли параметров конструктора для MaitreD яснее.
Большая часть типа TimeOfDay автоматически сгенерирована Visual Studio.
```

Сообщение отвечает на два вопроса.

- Почему был введен тип `TimeOfDay`?
- Почему бóльшая часть кода не тестировалась?

Другие примеры сообщений коммитов, отвечающих на вопросы «*почему*», вы можете найти в прилагаемом к книге репозитории.

Затруднения с пониманием смысла кода — наверное, самая острая проблема программной инженерии [24], так что очень важно писать как можно яснее.

9.1.2. Непрерывная интеграция

В большинстве организаций по разработке ПО непрерывная интеграция уже внедрена. За исключением того, что это не так.

Хотя, кажется, все *знают*, что непрерывная интеграция — это правильная практика разработки программного обеспечения, многие путают ее с наличием сервера непрерывной интеграции. Такой сервер — это хорошо, но сам по себе он не гарантирует непрерывной интеграции.

Непрерывная интеграция — это *практика*. Вы *постоянно интегрируете* свой код с кодом, над которым работают ваши коллеги.

Интеграция означает *слияние*, и не нужно воспринимать слово «*постоянно*» слишком буквально. Смысл в том, чтобы у вас была возможность регулярно делиться своим кодом с другими специалистами. Насколько регулярно? По крайней мере каждые четыре часа[1].

Я знаю много разработчиков, которые хвалили Git за то, что он решает проблему слияния кода. Как ни странно, это не так. Но Git способствует созданию рабочего процесса, отличного от рабочих процессов централизованной системы управления версиями.

Проблема слияния кода такая же, как и с любым другим видом параллельной работы над общим ресурсом. Это та же проблема, что и с транзакциями БД. Есть несколько клиентов, которые хотят изменить общий ресурс. При системе управления исходным кодом ресурс — это код, а не строки базы данных. Но проблема остается той же.

Этот вопрос можно решить несколькими способами. БД всегда предлагали транзакции в качестве решения, что предполагало блокировку ресурсов. Среда Visual SourceSafe работала так же. Как только файл был немного изменен, SourceSafe помечала его как снятый с контроля, и никто не мог его редактировать, пока он не будет поставлен на учет снова.

Бывало так, что люди уходили домой на день, оставив файлы снятыми с учета, а те, кто работали по другому графику, не могли ничего с этими файлами сделать. Пессимистическая блокировка плохо масштабируется.

Оптимистическая блокировка обычно более масштабируема, если конфликт маловероятен [55]. Прежде чем приступить к изменению ресурса, вы делаете его снапшот[2] и только затем редактируете. Чтобы

[1] Чем чаще, тем лучше. Вы можете интегрировать код при каждом внесении изменений с успешным прохождением тестов.

[2] Вы можете использовать хеш или версию строки, сгенерированную базой данных.

сохранить изменения, вы сравниваете текущее состояние со снапшотом (рис. 9.1). Если ресурс не изменился, с тех пор как вы начали с ним работать, можете безопасно сохранить свои изменения.

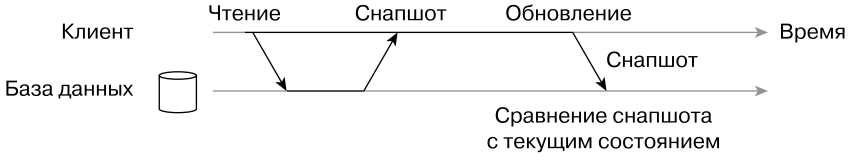


Рис. 9.1. Оптимистическая блокировка. Сначала клиент считывает текущую версию ресурса из БД. Пока клиент редактирует ресурс, он сохраняет копию снапшота. Чтобы обновить ресурс, он отправляет и копию снапшота. БД сравнивает снапшот с текущим состоянием ресурса. Обновление завершается, только если снапшот соответствует текущему состоянию

Даже если ресурс был отредактирован, есть возможность объединить два изменения. Если строка БД была отредактирована, но были изменены другие столбцы, вы все равно можете применить свои правки. Слияние при изменении файла кода возможно, если ваш коллега внес правки в часть файла, отличную от той, что редактировали вы[1].

Но если вы одновременно редактируете одну и ту же строку кода, возникает конфликт слияния. Как его избежать? Точно так же, как и в случае оптимистической блокировки. Вы не можете гарантировать, что этого никогда не произойдет, но вы можете снизить вероятность этого. Чем меньше времени вы тратите на редактирование кода, тем меньше вероятность того, что кто-то еще внесет изменения в ту же часть и в то же время.

Возможно, вы слышали, как непрерывную интеграцию сравнивают с «бегом по стволу». Некоторые воспринимают это настолько буквально, что не создают ветки в Git, а вместо этого создают все на *мастер-ветке*.

[1] Но нет никакой гарантии того, что объединенный результат целесообразен.

Единственное, чего вы так добьетесь, — это покажете всем, что не понимаете, в чем проблема. Проблема — в *конкуренции* действий, а не в названии ветки Git, над которой вы работаете. Если вы не используете методику моб-программирования[1], всегда есть риск того, что один из ваших коллег будет редактировать ту же строку кода, что и вы.

Снизьте этот риск: вносите небольшие изменения и совершайте слияние как можно чаще. Я рекомендую выполнять интеграцию *минимум* каждые четыре часа. Я выбрал такой период, так как это примерно половина рабочего дня. Не нужно сидеть над кодом целый день, прежде чем вы поделитесь им с остальной частью команды. В этом случае ваши локальные репозитории Git будут расходиться, и как результат — возникнут проблемы со слиянием.

Если вы не можете выполнить задачу за четыре часа, отметьте ее флагом[2] и в любом случае интегрируйте код [49].

9.1.3. Малые коммиты

В программировании много вариативности. Бывает, что вы производите большое количество кода за четыре часа, а бывает, что вы полдня сидите без единой работающей строчки. Чтобы разобраться в ошибке или воспроизвести ее, вам может понадобиться несколько часов.

Изучение незнакомого API может занять несколько дней. Бывает, что приходится переписывать код, на который вы уже потратили пару часов. И это нормально.

Основное преимущество Git — *маневренность*, благодаря которой вы можете экспериментировать (рис. 9.2). Попробуйте написать какой-нибудь код. Если он работает — коммит. Если нет — сброс. Эффективнее всего делать много небольших коммитов. Если последний созданный коммит был небольшим, значит, удаляя его, вы избавитесь от той части кода, которую действительно хотите удалить.

---

[1] Подробнее о моб-программировании вы можете прочитать в подразделе 9.2.2.

[2] Подробнее см. в разделе 10.1.

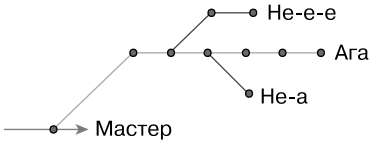


Рис. 9.2. При создании небольших коммитов ошибки будут обходиться вам дешево. Вы даже можете коммитнуть ошибки и оставить их на вспомогательных ветках на тот случай, если вдруг окажется, что они вам понадобятся позже. Ветка «Ага» выглядит многообещающе. По достижении необходимой контрольной точки в ней интегрируйте ее с мастер-веткой

Маневренность

Чем более изменчиво ваше окружение, тем ценнее ваша способность реагировать на непредвиденные события.

Маневренность — это военная концепция боевой авиации, показывающая, насколько быстро вы можете производить обмен между кинетической и потенциальной энергией, набирать и сбрасывать импульс [74]. Насколько хорошо вы умеете крутить монетку, например.

Дело не только в скорости. Речь идет о возможности изменить направление и разогнаться. Это полезно и при разработке ПО.

На тактическом уровне Git обеспечивает превосходную маневренность. Вы можете быть в середине какого-то процесса, но если вы понимаете, что вам нужно сделать что-то еще, вы можете легко спрятать свои изменения и начать заново. Если сомневаетесь, улучшит ли тот или иной рефакторинг ваш код, протестируйте его. Если думаете, что изменение улучшит ситуацию, — коммит. В противном случае — сброс.

Это не просто система контроля версий — это тактическое преимущество.

Git — это распределенная система контроля версий. Пока вы не поделитесь своими изменениями с другими специалистами или системами,

коммиты будут только на вашем локальном жестком диске. Это означает, что вы можете отредактировать свою историю коммитов, прежде чем отправить ее.

Мне всегда нравилось редактировать локальные ветки Git, прежде чем отправлять их. Не то чтобы я считал необходимым скрывать свои ошибки или казаться сверхдальновидным. Просто это дает мне свободу экспериментировать и по-прежнему позволяет оставлять последовательный и понятный след коммита.

При выполнении больших коммитов вам будет сложно управлять историей своего кода. Вы можете пожалеть о некоторых внесенных изменениях, но при объединении их с несвязанными правками в одном большом коммите (рис. 9.3) вы не можете легко отменить только те, от которых хотите избавиться.

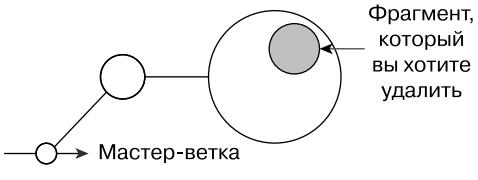


Рис. 9.3. При выполнении больших коммитов впоследствии может быть сложно удалить только их часть. Здесь коммит содержит некоторый код, который вы хотели бы удалить, но он — неотъемлемая часть коммита; в нем есть и код, который важно сохранить

Ваша история коммитов должна быть в виде серии снапшотов рабочего ПО. Никогда не нужно коммитить нерабочий код. С другой стороны, коммитить нужно каждый раз, когда ваш код успешно собирается. Делайте микрокоммиты [78].

- Переименовать символ — коммит.
- Извлечь метод — коммит.
- Встроить метод — коммит.

- Добавить тест и добиться его прохождения — коммит.
- Добавить Guard Clause — коммит.
- Исправить способ форматирования кода — коммит.
- Добавить комментарий — коммит.
- Удалить избыточный код — коммит.
- Исправить опечатку — коммит.

На практике нельзя сделать все коммиты маленькими. В репозитории примеров Git к этой книге есть много примеров микрокоммитов, но кое-где вы также сможете найти и большие.

Чем больше у вас мелких коммитов, тем легче вам маневрировать.

Через несколько часов работы вы всегда можете отказаться от части неподходящих вариантов. В результате у вас будет несколько небольших полезных коммитов. Очистите свою локальную ветку и интегрируйте ее с *мастером*.

9.2. КОЛЛЕКТИВНОЕ ВЛАДЕНИЕ КОДОМ

Есть ли у вас такая часть кодовой базы, над которой работает только один разработчик? Что будет, когда он уедет в отпуск или заболеет? А если он вообще уволится?

Вы можете организовать работу с кодом несколькими способами, но если кто-то один владеет частью кодовой базы, это становится проблемой. Каждый владелец становится критическим ресурсом — единственной точкой отказа. Это усложняет и рефакторинг. Вы не можете легко переименовать метод, если только один разработчик владеет им, а другой — кодом, вызывающим этот метод [30].

Делясь кодом, вы увеличиваете фактор автобуса. В идеале не должно быть такой части кодовой базы, которой владеет только один человек.

Фактор автобуса

Фактор автобуса — количество участников проекта, после потери которых он не сможет быть завершен оставшимися сотрудниками.

Например, сколько участников команды может быть «сбито автобусом», прежде чем разработка остановится?

Конечно, вы хотите, чтобы эта цифра была как можно больше. Если фактор равен *1*, значит, после потери *одного* участника команды разработка застопорится.

Некоторым специалистам не нравится такой термин. Вместо этого они иногда спрашивают: сможет ли команда продолжить работу над проектом, если один из них выиграет в лотерею и уволится? Этот фактор можно называть *фактором лотереи*, но смысл остается тот же.

Независимо от названия, основная идея здесь — повысить осведомленность о том, что обстоятельства меняются. Участники команды приходят и уходят. В дополнение к выигрышу в лотерею или попаданию под автобус людей могут перевести на другое место работы, или они просто могут уволиться по мириадам причин.

Дело не в том, чтобы реально *измерить* какой-либо фактор, а в том, чтобы организовать работу так, чтобы ни один участник команды не был незаменимым.

Когда у вас в команде больше одного специалиста, они могут распределить задачи между собой. Например, одни программисты занимаются разработкой пользовательского интерфейса, а другие — бэкенд-разработкой. Коллективное владение кодом не запрещает распределять задачи, но может привести к дублированию ответственности (рис. 9.4).

Я стараюсь избегать задач по разработке пользовательского интерфейса, но если в моей команде есть только один специалист по разработке

пользовательского интерфейса, я должен взять на себя ответственность и за эту часть кодовой базы. Как только над разработкой интерфейса будет работать более одного участника, я могу решить, что задача будет выполнена качественно и вовремя, и сосредоточиться на интересных мне задачах.

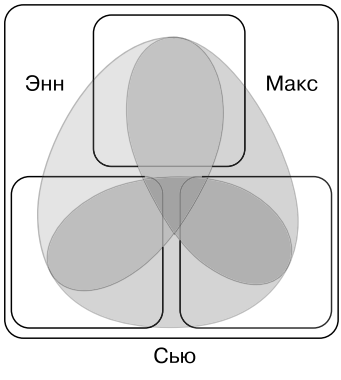


Рис. 9.4. Три разработчика (Энн, Макс и Сью) работают над кодовой базой. Энн работает над левым и верхним модулями (например, HTTP API и модель предметной области), Макс — над верхним и правым, а Сью взяла в работу два нижних. Все они разделяют ответственность за части кодовой базы между собой

Как мы уже знаем, единственный важный артефакт — это код. Коллективное владение кодом означает, что вы должны отвечать утвердительно на следующий вопрос: «Есть ли в команде более одного участника, которому комфортно работать только с определенной частью кода?»

Иначе говоря, как минимум два активных сопровождающих кодовой базы должны одобрить изменения.

Вы можете делать это формально и неформально, включая парное программирование и код-ревью. Суть в том, что любое изменение кода затрагивает более одного участника.

9.2.1. Парное программирование

В парном программировании [5] участвуют два разработчика ПО, совместно работающие над одной и той же проблемой. Есть несколько стилей такого программирования [12], но у всех одна общая черта: совместная работа происходит в режиме реального времени.

Этот процесс включает в себя непрерывный код-ревью на ходу [12]. Код, созданный в процессе парного программирования, отражает в себе соглашение о деталях реализации. Полученные коммиты уже содержат код, удобный как минимум для двух человек.

Я знаю команды, которые считали парное программирование слишком неформальным. Они добавляли заметки о соавторах в сообщениях коммитов или при интеграции изменений с *мастер-веткой*. Это совершенно не обязательно.

Парное программирование может поспособствовать коллективному владению кодом.

> *«Последовательное парное программирование гарантирует, что каждая строка кода была затронута или просмотрена как минимум двумя людьми. Это увеличивает шансы на то, что любой участник команды будет чувствовать себя комфортно, изменяя код практически в любом месте. Это делает кодовую базу более согласованной, чем если бы ее смотрели только отдельные разработчики.*
>
> *Само по себе парное программирование не гарантирует коллективного владения кодом. Эффективнее всего чередовать людей по разным парам и областям кода, чтобы предотвратить разрозненность знаний» [12].*

Это почти как если бы парное программирование сопровождалось код-ревью в реальном времени и неформальным процессом утверждения в качестве побочного эффекта. Поскольку вы уже два члена команды, работающие над кодом, вам не нужно ждать, пока кто-то еще одобрит изменения.

Но даже так парное программирование нравится не всем. Лично я, как типичный интроверт [16], считаю такой процесс утомительным. Это оставляет мало места для размышлений и требует синхронизации расписаний.

Я не настаиваю на том, чтобы абсолютно все команды занимались парным программированием, но против вышеперечисленных преимуществ возражать трудно[1]. Несмотря на это, едва ли эффективно заниматься парным программированием все время [12]. Лучше комбинировать парное программирование с другими методами взаимодействия с кодовой базой и добиться того соотношения, которое подходит вашей конкретной организации.

9.2.2. Моб-программирование

Если два программиста работают над одной задачей — это хорошо, три — вроде бы еще лучше. А что насчет четырех? Или пяти?

Если у вас есть возможность арендовать конференц-зал или другое место, где группа людей будет совместно писать код, вы можете участвовать в моб-программировании[2].

Достаточно сложно убедить руководство (или даже ваших коллег-разработчиков) в продуктивности парного программирования. Предполагается, что два человека, работающие над одной и той же проблемой, должны демонстрировать вдвое меньшую продуктивность, чем двое людей, одновременно работающих над двумя отдельными. Еще труднее убедить скептиков в том, что три и более человека, работающие над одной проблемой, не снизят общую производительность.

Надеюсь, что моя книга убедила вас в том, что продуктивность не связана со скоростью печати на клавиатуре.

---

[1] Есть доказательства того, что парное программирование — довольно эффективный способ работы [116].

[2] Мне не нравится термин «моб-программирование» (от англ. mob programming), так как для меня толпа (от англ. mob) — это бездумная орава людей. Командное программирование [84] — более подходящий термин.

В экономике есть закон убывающей отдачи. Представьте, что вы пытаетесь организовать моб-программирование с участием 50 человек. Большинству будет мало что предложить, или, наоборот, ничего не будет сделано, если у каждого возникнут свои соображения по тому или иному вопросу.

Оптимальный вариант — создание малых групп.

Моб-программирование не мой привычный метод работы, но иногда он довольно полезен. Например, если вы обучаете программированию. В рамках одного задания я проводил два-три дня в неделю с другими программистами, помогая им применять методы TDD-разработки к их коду. Через несколько месяцев я уехал в отпуск, а они продолжали заниматься разработкой через тестирование. Моб-программирование отлично подходит для обучения и передачи знаний.

Поскольку над одним набором изменений кода работает несколько человек, вы получаете все преимущества проверки и одобрения группового программирования, которые предоставляет вам парное программирование.

Попробуйте при случае. Возможно, вам понравится.

9.2.3. Задержка код-ревью

Как убедительно доказывает Лоран Боссавит, большинство общеизвестных сведений о разработке ПО — это скорее миф, чем реальность [13]. Лишь несколько практик доказанно эффективны, и код-ревью — одна из таких [20].

Это один из самых эффективных способов поиска дефектов в коде [65], но большинство организаций им не пользуются, так как специалисты считают, что это замедляет разработку.

Код-ревью правда может привнести задержку в процесс разработки. Но было бы ошибкой думать, что разработка будет эффективнее, если большинство багов останутся незамеченными на очень долгое время.

Во многих организациях, с которыми я сотрудничал, частью работы (обычно называемой функциональностью) занимается один программист. Когда он объявляет о *выполненной* работе, дальнейшая проверка не проводится.

У разных организаций разные сигналы *готовности*. Некоторые используют фразу *«сделано сделано»* (от англ. done done), подразумевая, что работа выполнена, только когда функциональность завершена и доступна для использования в продакшене.

Как мы уже говорили в разделе 3.1.2, слишком ограниченный фокус на предоставлении ценности может заставить нас упустить из виду проблемы, возникающие при внедрении в продакшен ненадежной, неустойчивой, ложной функциональности.

Рисунок 9.5 иллюстрирует ситуацию, где функциональность объявлена завершенной. Позже, когда в ней обнаруживается баг, вы уже работаете над другой задачей, а исправление ошибок не входит в планы. Команда может все исправить, но, так как это незапланированное происшествие, приходится либо работать сверхурочно, либо срывать дедлайны по другим функциональностям.

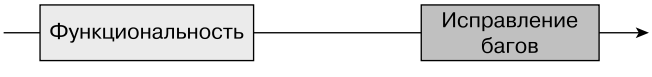


Рис. 9.5. Многие организации не проводят код-ревью. С того момента, как разработчик объявляет, что функциональность реализована, может пройти много времени, прежде чем баг будет обнаружен. Это приводит к незапланированной работе

Сорванные сроки означают, что разработка перешла в режим «кранч» (от англ. crunch — «хруст, треск»), когда работники вынуждены работать сверхурочно и постоянно тушить пожар. Никогда не бывает достаточно времени делать что-то «правильно», так как всегда появляется новая непредвиденная проблема, которую приходится решать. И так по кругу.

Код-ревью поможет эффективно обнаруживать проблемы до объявления о выполненной работе. Предотвращение ошибок становится частью процесса, а не частью проблемы.

Проблема с типичными подходами к разработке показана на рис. 9.6. Разработчик отправляет выполненную часть на рассмотрение. Затем проходит много времени, прежде чем дело дойдет непосредственно до ревью.

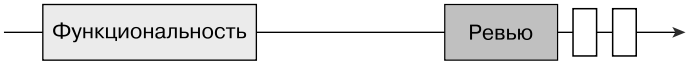


Рис. 9.6. Как не нужно делать код-ревью: допускать прохождение большого количества времени между завершением реализации функциональности и ревью. Блоки меньшего размера — улучшения, основанные на первоначальном ревью, и последующие ревью сделанных улучшений

На рис. 9.7 показано очевидное решение проблемы: сократите время ожидания. Сделайте код-ревью частью ежедневного ритма организации.

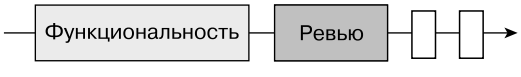


Рис. 9.7. Сократите время ожидания между завершением реализации функциональности и код-ревью. Ревью обычно провоцирует некоторые улучшения и последующее более короткое ревью этих улучшений. Эти действия обозначены блоками меньшего размера

У большинства сотрудников уже сформировался свой привычный распорядок, которому они следуют. Нужно сделать код-ревью частью этой рутины. Можете сделать это индивидуально или встроить в ежедневный ритм команды. Многие команды проводят ежедневные созвоны или совещания. Иногда такие мероприятия тормозят рабочий процесс. Обед — еще один перерыв в работе.

Можно, например, выделить для ревью по полчаса каждое утро и по полчаса после обеда[1].

Помните, что следует вносить только небольшие наборы изменений — те, работа над которыми занимает менее половины рабочего дня. Если вы введете такую практику и все участники команды будут проверять эти небольшие изменения дважды в день, максимальное время ожидания составит около четырех часов.

9.2.4. Отклонение набора изменений

Как-то раз я помог одной организации перейти к коллективному владению кодом. И одной из практик, которым я хотел научить их, было совершение маленьких шагов.

Вскоре я получил пул-реквест от удаленного разработчика, о котором не слышал уже пару недель. Это было что-то невероятное: тысячи строк для ревью, распределенные по 50 файлам.

Я не стал проводить ревью. И сразу же отклонил запрос на том основании, что объем кода чересчур большой[2]. Позднее я показал всей команде, как вносить небольшие изменения. В результате я больше никогда не получал пул-реквесты такого огромного размера.

При каждом проведении код-ревью *отказ* должен быть потенциальной опцией. Ревью ничего не стоит, если является лишь формальностью и проводится для галочки.

Я часто вижу, как люди отправляют большие изменения на ревью. Такой набор изменений отражает дни (или недели) работы — и поэтому его проверка может занимать очень много времени [78]. Ревью может затянуться на дни, пока автор пытается разобраться с мириадами ваших опасений.

---

[1] Вы также можете выделить полчаса перед обедом, а также перед тем, как пойти домой на весь день, но вы, скорее всего, что-то пропустите, так как будете заняты чем-то другим.

[2] Руководство поддержало мое решение. Иногда консультантам дозволено делать то, что не разрешено обычным сотрудникам. Несправедливо, но это так.

Либо так, либо вы просто сдаетесь и принимаете изменения, потому что у вас есть другая работа.

Никогда так не делайте. Отклоняйте большие наборы изменений.

Рецензенты часто неохотно отклоняют изменения, отражающие многие дни работы. Это распространенная проблема, известная как *ошибка утопленных затрат* [51]. Итак, ваш коллега уже потратил много времени на внесение изменений, но если вы считаете, что вам придется тратить еще больше на поддержание плохого дизайна, то выбор очевиден. Сократите свои потери. Время, которое ваш коллега потратил впустую, уже потеряно. Не тратьте еще больше времени на плохо организованный код.

Отказываться от дней или недель работы больно. Отказ от нескольких часов работы более приемлем. Это еще одна причина вносить небольшие изменения, на которые уходит полдня.

Кроме того, обзор кода, который занимает больше часа, попросту неэффективен [20].

9.2.5. Код-ревью

Главный вопрос, на который должен быть пролит свет после код-ревью: «Справлюсь ли я с поддержанием этого?»

Это действительно так[1]. Вы можете думать, что разработчик будет рад поддерживать свой код. Если вы тоже готовы поддерживать его, то вас уже двое и вы находитесь на пути к коллективному владению кодом.

Что вы должны проверять во время ревью?

Самый главный критерий — читабельность кода. Достаточно ли он понятен?

---

[1] Честно говоря, вы не должны забывать еще более важный и фундаментальный вопрос: «Затрагивает ли изменение обоснованную проблему?» Иногда вы неправильно понимаете свое задание. Я так делал. Мы все так делаем. Помните об этом вопросе при код-ревью. Это может быть причиной отказа от изменения, но я не считаю это главной целью код-ревью.

Помните, что документация (если она вообще есть) обычно устаревает, комментарии могут вводить в заблуждение и т. д. В конечном счете вы можете доверять только самому коду. Однажды придется его поддерживать, а автора рядом может уже не быть.

Иногда код-ревью осуществляется разработчиком и рецензентом совместно, что нежелательно по двум причинам:

- разработчик не может самостоятельно судить о том, читабелен ли код;
- разработчик может быстро уговорить рецензента не обращать внимания на проблемы.

Код-ревью должно проводиться рецензентом, *читающим* код в собственном темпе. Разработчик только недавно написал код, поэтому не может оценить, читаем ли он, и не должен активно участвовать в чтении кода.

Хотя отказ — это нормально, ваша работа как рецензента не в том, чтобы обидеть автора или доказать свое превосходство. Вам обоим нужно достигнуть соглашения о том, как двигаться дальше.

Придирки бесполезны, поэтому не придирайтесь к форматированию кода или к именам переменных[1]. Подумайте о том, помещается ли код в вашей голове. Не слишком ли длинные или сложные методы?

Кори Хаус предлагает обратить внимание на следующие пункты [47].

- Работает ли код, как задумано?
- Понятно ли намерение?
- Есть ли ненужное дублирование?
- Может ли существующий код решить эту проблему?
- Может ли код быть проще?
- Все ли тесты исчерпывающие и понятные?

[1] Вы всегда можете изменить форматирование позже или исправить опечатку в имени переменной. Если исправление некритично, не позволяйте ему затягивать ревью. Но следует устранять опечатки в общедоступных API, так как их исправление — это внесение критических изменений.

Это неполный список, но он дает представление о том, что нужно искать.

Результат код-ревью — это обычно не бинарное решение «принять/отклонить». Вместо этого создается список предложений, которые разработчик и рецензент могут использовать для диалога. Хотя разработчик должен отсутствовать при фактическом чтении кода, дружеское взаимодействие может помочь ускорить остальную часть процесса.

Обычно вы соглашаетесь на некоторые улучшения. Автор возвращается к их реализации и отправляет новые изменения на повторное ревью (рис. 9.7). Это итеративный процесс. Последующие ревью выполняются быстрее, после чего вы приходите к консенсусу и внедряете изменения.

Все участники команды должны быть разработчиками, и все должны проверять код других участников. Быть рецензентом — это не привилегия и не бремя для избранных.

Такой процесс работы не только стимулирует коллективное владение кодом, но и побуждает всех проводить ревью как полагается.

9.2.6. Пул-реквесты

Онлайн-службы Git, такие как GitHub, Azure DevOps Services и т. д., поддерживают *GitHub flow*[1], процесс, облегчающий работу команды, в котором вы создаете ветки на своем локальном компьютере, но используете централизованную службу для обработки слияний.

Чтобы объединить ветку с мастером, можете отправить пул-реквест. Это запрос на интеграцию ваших изменений с мастер-веткой.

Чаще всего вы можете выполнить слияние самостоятельно. Но вы должны сделать политикой команды то, что кто-то другой должен проверить и утвердить изменения. Это просто еще один способ выполнить код-ревью.

---

[1] Не путать с Gitflow.

При создании пул-реквеста помните о правилах работы с Git [91]:

- делайте каждый пул-реквест как можно меньше. Еще меньше, чем вы думаете;
- в каждом пул-реквесте делайте только одно действие. При выполнении нескольких действий создайте для каждого отдельный пул-реквест;
- избегайте переформатирования, если только это не единственная цель пул-реквеста;
- убедитесь, что код собирается;
- убедитесь, что все тесты пройдены;
- добавляйте тесты для нового поведения;
- пишите правильные сообщения коммитов.

При *ревью* пул-реквеста применимы все правила, касающиеся проведения код-ревью. К тому же GitHub flow — это асинхронный рабочий процесс, поэтому ревью обычно выполняется в письменном виде. Имейте в виду, что на письме намерения и интонация легко теряются. Возможно, вы и не будете иметь в виду ничего плохого, используя определенную фразу, но получатель сочтет ее обидной. Будьте предельно вежливы и используйте смайлики, чтобы показать свое дружелюбное отношение.

Как рецензент, вы должны выделить время для надлежащего проведения ревью. Всегда учитывайте, что, если пул-реквест слишком большой, лучше его отклонить[1], чем бездумно принять.

Если вы все-таки решите выполнить ревью, *сотрудничайте с разработчиком*, чтобы внести улучшения. Не просто указывайте на то, что вам не нравится, а предложите конкретные альтернативы. Не забывайте отмечать, когда что-то вам нравится. Извлеките (pull) код и запустите его на своем компьютере [113].

---

[1] Новички часто отправляют слишком большие запросы на изменение кода, так как не знают, как разделить свою задачу на подзадачи. Это и есть причина существования данной книги. Это не то, что можно уяснить на ходу с первого дня. Помогайте своим коллегам.

9.3. ЗАКЛЮЧЕНИЕ

У каждого участника команды есть свои сильные стороны. Вполне естественно, что вы предпочитаете работать с той частью кодовой базы, которая вам больше всего подходит. Если все будут так делать, то у каждого может появиться чувство собственности. Это нормально, до тех пор пока сохраняется *слабое владение кодом* [30], когда у кода есть «естественный» владелец или главный разработчик, но каждый может вносить изменения.

Вы должны стремиться к коллективному владению кодом с помощью процессов, которые предусматривают, что за каждое изменение в кодовой базе отвечает более одного человека. Это можно сделать неформально, с помощью парного или моб-программирования, или более формально, с помощью код-ревью.

Как уже обсуждалось в подразделе 1.3.4, настоящие инженеры работают в командах и утверждают работу друг друга [40]. Наблюдать за происходящим более чем одной парой глаз — одна из самых эффективных инженерных практик, которую вы можете использовать при разработке ПО.

II Устойчивость

Первая часть была посвящена ускорению. Мы рассмотрели структуру на примере кода, развивающегося с нуля (без кода) до развернутой функциональности.

Если у вас есть развернутая функциональность, у вас есть рабочая система. Но одной функциональности недостаточно. Вам придется добавить больше. Несмотря на все усилия, в процессе написания кода вы все равно обнаружите баги в программе.

Разве интересно разгоняться до большой скорости, только чтобы врезаться в стену? По достижении определенной скорости важно сохранить ее.

Часть II посвящена поддержанию наиболее эффективной скорости работы. Как вы добавляете новую функциональность в существующую кодовую базу? Как устраняете ошибки? А что насчет сквозной функциональности и производительности?

В части II мы обсудим эти вопросы с акцентом на дополнении существующего кода. Примеры будут из той же кодовой базы, но из более широкого диапазона коммитов.

В репозитории Git я оставил материалы к части II в рабочем виде, неотшлифованными. Я не буду пытаться скрывать свои ошибки, поэтому вы увидите коммиты, которые отменяют работу предыдущих, и т. д.

Я писал развернутые сообщения коммитов всякий раз, когда чувствовал, что коммит содержит что-то, на что стоит обратить внимание. При желании вы можете прочитать лог как своего рода приложение.

10 РАСШИРЕНИЕ КОДОВОЙ БАЗЫ

Реальность профессиональной разработки ПО такова, что вы в основном работаете с существующим кодом. Ранее было уже много сказано о начале работы над новой кодовой базой и о том, как максимально эффективно перейти от нуля к работающей системе. Разработка с нуля сопряжена с определенным набором проблем, но они отличаются от тех, что обычно связаны с внесением изменений в существующую кодовую базу.

В основном вы будете редактировать продакшен-код. Даже если вы занимаетесь разработкой через тестирование, вам придется добавлять новые тесты и параллельно изменять существующий продакшен-код.

Процесс изменения структуры существующего кода без изменения его поведения называется *рефакторингом*. На эту тему есть очень много доступной информации [34; 53; 27], поэтому я не буду повторять тот же материал в книге. Вместо этого я сосредоточусь на добавлении нового поведения в кодовую базу.

Добавление поведения можно разделить на три типа, таких как:

- совершенно новая функциональность;
- улучшения существующего поведения;
- исправление багов.

Об исправлении багов вы узнаете в главе 12, а в этой мы рассмотрим два первых пункта. Совершенно новое поведение — самое простое изменение.

10.1. ФУНКЦИОНАЛЬНЫЕ ФЛАГИ

Когда ваша задача — добавить совершенно новую функциональность, бóльшая часть кода, который вы планируете написать, будет уже *новым* кодом, а не изменениями, внесенными в существующую кодовую базу.

Возможно, есть существующая инфраструктура кода, которую вы можете использовать, и скорее всего, вам придется внести в нее изменения, прежде чем добавить новую функциональность. Но в большинстве случаев это не требуется. Самая большая проблема, с которой вы можете столкнуться[1], — это необходимость придерживаться практики непрерывной интеграции.

Как вы уже знаете, вы должны объединять свой код с мастер-веткой как минимум дважды в день. Другими словами, вы можете работать над задачей максимум четыре часа, прежде чем вам нужно будет внедрить изменения. Что делать, если вы не можете выполнить задачу за четыре часа?

Большинству людей неудобно слияние незавершенной функциональности с мастер-веткой, особенно если в команде практикуется непрерывное развертывание. Это будет означать, что в продакшен-системе развернута неполная функциональность, что нежелательно.

Решение в том, чтобы различать саму функциональность и реализующий ее код. Вы можете развернуть неполноценный код в своей продакшен-среде, если реализуемое им поведение недоступно. Скройте функциональность с помощью функциональных флагов [49].

---

[1] За исключением того, что сама функциональность может быть сложной в реализации.

10.1.1. Календарь

Рассмотрим пример кодовой базы ресторана. По окончании работы над функциональностью резервирования я захотел добавить в систему календарь, чтобы позволить клиенту просматривать доступные даты для бронирования определенных мест. Это можно реализовать с помощью пользовательского интерфейса, чтобы показать, доступна ли дата для дополнительного резервирования, и т. д.

Добавление календаря — сложная задача. Вам нужно включить навигацию от месяца к месяцу, расчет максимального числа оставшихся мест для данного временного интервала и т. д. Маловероятно, что вы уложитесь в четыре часа[1].

Прежде чем я начал работу, домашний ресурс REST API выдал ответ в формате JSON (листинг 10.1).

Листинг 10.1. Пример HTTP-взаимодействия с домашним ресурсом REST API. При запросе GET для индексной страницы / вы принимаете массив ссылок в формате JSON. Как вы можете понять из наличия localhost в URL, я взял этот пример с помощью запуска системы на моем компьютере для разработки. При запросе ресурса из развернутой системы URL-адрес указывает правильное имя хоста

```
GET / HTTP/1.1

HTTP/1.1 200 OK
Content-Type: application/json
{
  "links": [
    {
      "rel": "urn:reservations",
      "href": "http://localhost:53568/reservations"
    }
  ]
}
```

[1] Изучив кодовую базу примеров, вы сможете сравнить коммит, который начинает эту работу, с коммитом, который ее завершает. Их разделяет почти два месяца! Итак, в промежутке у меня был четырехнедельный отпуск, я выполнял другую работу и т. д. Но вся работа по-прежнему заняла бы примерно от одной до двух недель. Это точно не было сделано за четыре часа!

Система — это настоящий REST API, который использует элементы управления гипермедиа (то есть *ссылки*) [2], а не OpenAPI (который раньше называли Swagger) или что-то подобное. Клиент, который хочет выполнить резервирование, запрашивает единственный задокументированный URL-адрес API (домашний ресурс) и ищет ссылку с типом отношения `urn:reservations`. Фактический URL-адрес должен быть непрозрачным для клиента.

До того как я начал работать над функциональностью календаря, сгенерировавший ответ код из листинга 10.1 выглядел так же, как в листинге 10.2.

Листинг 10.2. Код, отвечающий за генерацию вывода из листинга 10.1. CreateReservationsLink — это закрытый (private) вспомогательный метод (Restaurant/b6fcfb5/Restaurant.RestApi/HomeController.cs)

```
public IActionResult Get()
{
    return Ok(new HomeDto { Links = new[]
    {
        CreateReservationsLink()
    } });
}
```

Когда я начал работать над функциональностью календаря, я вскоре понял, что задача займет по времени более четырех часов, поэтому я ввел функциональный флаг [49]. Это позволило мне написать метод `Get` (листинг 10.3).

Листинг 10.3. Генерация ссылок календаря, скрытых за функциональным флагом. По умолчанию флаг enableCalendar установлен в значение false, что приводит к выводу из листинга 10.1. Реализация новой функциональности выделена серым цветом (Restaurant/cbfa7b8/Restaurant.RestApi/HomeController.cs)

```
public IActionResult Get()
{
    var links = new List<LinkDto>();
    links.Add(CreateReservationsLink());
    if (enableCalendar)
    {
        links.Add(CreateYearLink());
        links.Add(CreateMonthLink());
```

```
        links.Add(CreateDayLink());
    }
    return Ok(new HomeDto { Links = links.ToArray() });
}
```

Переменная `enableCalendar` — это логическое значение (флаг), которое в конечном счете происходит из файла конфигурации. В контексте листинга 10.3 это поле класса, предоставленное через конструктор контроллера (листинг 10.4).

Листинг 10.4. Конструктор HomeController получает функциональный флаг (Restaurant/cbfa7b8/Restaurant.RestApi/HomeController.cs)

```
private readonly bool enableCalendar;

public HomeController(CalendarFlag calendarFlag)
{
    if (calendarFlag is null)
        throw new ArgumentNullException(nameof(calendarFlag));

    enableCalendar = calendarFlag.Enabled;
}
```

Класс `CalendarFlag` — это просто оболочка логического значения. Она концептуально избыточна, но необходима из-за технической детали: встроенный контейнер внедрения зависимостей ASP.NET отвечает за сопоставление классов с их зависимостями и отказывается рассматривать тип значения[1] как зависимость. Чтобы обойти эту проблему, я представил оболочку `CalendarFlag`[2].

Во время запуска система считывает разные значения из своей конфигурации. Они используются для настройки соответствующих

[1] В C# известен как struct.

[2] Я мог смириться с этим решением, так как знал, что оно временное. Как только функциональность будет полностью реализована, вы можете удалить ее флаг. Альтернатива внедрению классов-оболочек для примитивных зависимостей — полный отказ от встроенного контейнера внедрения зависимостей. Я сделал бы это в кодовой базе, если бы мне пришлось поддерживать ее годами, но я признаю, что у этого есть свои преимущества и недостатки. Подробнее о том, как это сделать в ASP.NET, вы можете прочитать в книге Dependency Injection Principles, Practices and Patterns [25].

служб. В листинге 10.5 приведен пример считывания значения `EnableCalendar` и настройки службы `CalendarFlag`.

Листинг 10.5. Настройка функционального флага на основе его значения конфигурации (Restaurant/cbfa7b8/Restaurant.RestApi/Startup.cs)

```
var calendarEnabled = new CalendarFlag(
    Configuration.GetValue<bool>("EnableCalendar"));
services.AddSingleton(calendarEnabled);
```

Если значения конфигурации `EnableCalendar` нет, метод `GetValue` возвращает значение по умолчанию, которое для логических значений в .NET равно `false`. Поэтому я просто не настраивал функциональность, а значит, мог продолжать слияние и развертывание в продакшене, не раскрывая это поведение.

Но в автоматизированных интеграционных тестах я изменил конфигурацию для включения этой функциональности (листинг 10.6). Это значит, что я все еще могу использовать интеграционные тесты для управления поведением новой функциональности.

Листинг 10.6. Переопределение конфигурации функционального флага для тестирования. Выделенные строки кода, по сравнению с кодом листинга 4.22, являются новыми (Restaurant/cbfa7b8/Restaurant.RestApi.Tests/RestaurantApiFactory.cs)

```
protected override void ConfigureWebHost(IWebHostBuilder builder)
{
    if (builder is null)
        throw new ArgumentNullException(nameof(builder));

    builder.ConfigureServices(services =>
    {
        services.RemoveAll<IReservationsRepository>();
        services.AddSingleton<IReservationsRepository>(
            new FakeDatabase());

        services.RemoveAll<CalendarFlag>();
        services.AddSingleton(new CalendarFlag(true));
    });
}
```

Кроме того, при необходимости выполнить пробное тестирование, взаимодействуя с новой функциональностью календаря по-особенному, я мог бы установить флаг `EnableCalendar` в значение `true` в моем локальном файле конфигурации, и поведение тоже было бы активно.

Однажды, после нескольких недель работы, мне наконец удалось реализовать функциональность и включить ее в продакшене. Я удалил класс `CalendarFlag`, и весь условный код, который зависел от этого флага, больше не компилировался. После этого для упрощения фрагментов кода, где использовался флаг, в основном нужно было *полагаться на компилятор* [27]. Удалять код всегда приятно, так как это значит, что теперь нужно меньше поддерживать.

В результате домашний ресурс теперь выдает следующий ответ (листинг 10.7).

Листинг 10.7. Пример HTTP-взаимодействия с домашним ресурсом REST API, теперь со ссылками на календарь. Сравните с листингом 10.1

```
GET / HTTP/1.1

HTTP/1.1 200 OK
Content-Type: application/json
{
  "links": [
    {
      "rel": "urn:reservations",
      "href": "http://localhost:53568/reservations"
    },
    {
      "rel": "urn:year",
      "href": "http://localhost:53568/calendar/2020"
    },
    {
      "rel": "urn:month",
      "href": "http://localhost:53568/calendar/2020/10"
    },
    {
      "rel": "urn:day",
```

```
      "href": "http://localhost:53568/calendar/2020/10/20"
    }
  ]
}
```

Здесь показано, как использовать флаг функции, чтобы скрыть ее до полной реализации. Этот пример основан на REST API, где легко скрыть незавершенное поведение: просто не отображайте новую возможность в виде ссылки. В других типах приложений вы можете использовать флаг, чтобы скрыть соответствующие элементы пользовательского интерфейса и т. п.

10.2. ПАТТЕРН STRANGLER («ДУШИТЕЛЬ»)

Как правило, вы добавляете новую функциональность с помощью добавления нового кода в существующей кодовой базе. Улучшение существующей функциональности — это нечто другое.

Однажды я руководил рефакторингом в направлении более глубокого понимания [26]. Мы с коллегой определили, что для реализации новой функциональности нужно изменить фундаментальный класс нашей кодовой базы.

Обычно такие озарения редко посещают нас в нужное время, но мы все же хотели внести изменения, и руководство позволило нам это сделать.

Неделю спустя наш код все еще не компилировался.

Я надеялся, что смогу внести изменения в рассматриваемый класс, а затем, *опираясь на компилятор* [27], определить места вызовов, требующие модификации. Проблема была в наличии множества ошибок компиляции, и их исправление не было простым вопросом поиска и замены.

В итоге мой босс отвел меня в сторону, чтобы сообщить, что его не устраивает такое положение дел. Я мог лишь согласиться.

После этого небольшого выговора он позволил мне продолжить работу, и спустя несколько дней титанических[1] усилий она была завершена.

Я не хочу, чтобы в моей карьере был еще один подобный провал.

Кент Бек сказал:

> *«Сделайте каждое желаемое изменение легким (предупреждение: это может быть трудно), затем проведите это легкое изменение» [6].*

Я действительно пытался сделать изменения легкими, но не осознавал, насколько это будет тяжело. Но это не должно быть так. Следуйте простому эмпирическому правилу.

> Вносите любое существенное изменение не сразу, а постепенно.

Это правило также известно как паттерн Strangler («Душитель») [35]. Не беспокойтесь, он не имеет ничего общего с насилием. Он назван в честь необычного растения — фикуса-душителя. Дело в том, что в тропических условиях у растений возникает жесткая конкуренция. Солнечный свет закрыт кронами вековых деревьев, чьи крепкие корни вбирают в себя все полезные ресурсы из почвы. Пробиться новому ростку становится крайне сложно. Но фикусы-душители нашли выход. Их семена изначально попадают на кроны таких деревьев, где много света, и там пускают свои побеги. Они растут медленно, но со временем их корни спускаются вниз до самой земли, обвивая ствол носителя. И как только это происходит, скорость их роста удваивается. Теперь ствол не может расти вширь, так как фикус сдавливает его в своих горячих объятиях (рис. 10.1).

---

[1] Вснесу ясность: героизм — это не инженерная практика. Он слишком непредсказуем и стимулирует развитие ошибки утопленных затрат. Попробуйте обойтись без него.

Рис. 10.1. Этапы жизни фикуса-душителя. Слева — дерево, посередине дерево, опоясанное фикусом, а справа остался только фикус-душитель

Мартин Фаулер первоначально описал паттерн «Душитель» в контексте крупномасштабной архитектуры, как способ постепенной замены устаревшей системы более новой. Я обнаружил, что это полезно почти в любом масштабе.

В ООП паттерн можно применять как на уровне метода, так и на уровне класса. В первом случае вы сначала добавляете новый метод, постепенно перемещаете вызывающие объекты и, наконец, удаляете старый. Во втором — сначала добавляете новый класс, постепенно перемещаете вызывающие объекты и, наконец, удаляете старый.

Мы рассмотрим оба примера, начиная с уровня метода.

10.2.1. Паттерн Strangler. Уровень метода

Для реализации функции календаря из раздела 10.1 мне нужен был способ считывания сведений о резервировании на несколько дат. Но текущая реализация интерфейса `IReservationsRepository` выглядела как в листинге 10.8. Метод `ReadReservations` принимал в качестве входных данных одно значение `DateTime` и возвращал все резервирования на эту дату.

Листинг 10.8. Интерфейс IReservationsRepository с методом ReadReservations, ориентированным на одну дату (Restaurant/53c6417/Restaurant.RestApi/IReservationsRepository.cs)

```
public interface IReservationsRepository
{
    Task Create(Reservation reservation);

    Task<IReadOnlyCollection<Reservation>> ReadReservations(
        DateTime dateTime);

    Task<Reservation?> ReadReservation(Guid id);

    Task Update(Reservation reservation);

    Task Delete(Guid id);
}
```

Мне нужно было создать метод, возвращающий резервирование на определенный диапазон дат. Скорее всего, вы захотите добавить новую перегрузку метода и оставить все как есть. Технически это возможно, но подумайте о таксе на обслуживание: чем больше кода вы добавляете, тем больше кода придется обслуживать. Добавление метода в интерфейсе означает, что его нужно будет обслуживать и во всех реализациях.

Лучше заменить устаревший метод `ReadReservations` новым. Это возможно, так как считывание резервирований для диапазона дат вместо одной даты фактически ослабляет предварительные условия. Текущий метод можно рассматривать как частный случай, где диапазон отражает всего одну дату.

Но если бóльшая часть вашего кода уже вызывает текущий метод, быстрое и неосторожное внесение изменений может оказаться слишком объемным. Вместо этого сначала добавьте новый метод, постепенно перенесите места вызовов и, наконец, удалите старый. В листинге 10.9 показан пример интерфейса `IReservationsRepository` с добавленным новым методом.

При внесении такого метода код не может скомпилироваться, пока вы не добавите его во все классы, реализующие интерфейс. У кодовой базы резервирования ресторана есть только два средства реализации:

`SqlReservationsRepository` и `FakeDatabase`. Я добавил реализацию к обоим классам в одном и том же коммите, но это все, что мне нужно было сделать. Даже с реализацией SQL работа займет 5–10 минут.

Листинг 10.9. Интерфейс IReservationsRepository с дополнительным методом ReadReservations, ориентированным на диапазон дат. Выделенный код, по сравнению с листингом 10.8, является новым (Restaurant/fa29d2f/Restaurant.RestApi/IReservationsRepository.cs)

```
public interface IReservationsRepository
{
    Task Create(Reservation reservation);

    Task<IReadOnlyCollection<Reservation>> ReadReservations(
        DateTime dateTime);

    Task<IReadOnlyCollection<Reservation>> ReadReservations(
        DateTime min, DateTime max);

    Task<Reservation?> ReadReservation(Guid id);

    Task Update(Reservation reservation);

    Task Delete(Guid id);
}
```

Другой способ: добавить новую перегрузку метода `ReadReservations` как в `SqlReservationsRepository`, так и в `FakeDatabase`, но оставить их генерировать исключение `NotImplementedException`. Затем, в следующих коммитах, можно использовать TDD-разработку, чтобы добиться желаемого поведения. На каждом этапе этого процесса у меня был бы набор коммитов, которые я мог объединить с мастер-веткой.

Еще один вариант — сначала добавить методы с идентичными сигнатурами к конкретным классам и только потом, когда все они будут на месте, добавить метод в интерфейс.

В любом случае вы можете *постепенно* разрабатывать новый метод, так как нет кода, где он бы использовался.

После утверждения этого метода вы сможете изменять места вызовов *по одному*. Делая так, вы можете тратить столько времени, сколько вам нужно. Вы можете совершить слияние с мастер-веткой в любое

время в течение этого процесса, даже если это означает развертывание в продакшене. В листинге 10.10 приведен фрагмент кода, вызывающий новую перегрузку.

Листинг 10.10. Фрагмент кода, вызывающий новую перегрузку метода ReadReservations. Две первые выделенные строки — новые, а последняя отредактирована для вызова нового метода вместо исходного ReadReservations (Restaurant/0944d86/Restaurant.RestApi/ReservationsController.cs)

```
var min = res.At.Date;
var max = min.AddDays(1).AddTicks(-1);
var reservations = await Repository
    .ReadReservations(min, max)
    .ConfigureAwait(false);
```

Я менял код вызова по одному месту за раз и после каждого изменения делал коммит в Git. После нескольких коммитов все было готово: больше не было кода, вызывающего исходный метод `ReadReservations`.

Наконец я мог удалить `ReadReservations`, оставив интерфейс `IReservationsRepository` (листинг 10.11).

Листинг 10.11. Интерфейс IReservationsRepository после завершения паттерна Strangler. Старый ReadReservations больше не используется. Применяется только новая версия. Сравните с листингами 10.8 и 10.9 (Restaurant/bcffd6b/Restaurant.RestApi/IReservationsRepository.cs)

```
public interface IReservationsRepository
{
    Task Create(Reservation reservation);

    Task<IReadOnlyCollection<Reservation>> ReadReservations(
        DateTime min, DateTime max);

    Task<Reservation?> ReadReservation(Guid id);

    Task Update(Reservation reservation);

    Task Delete(Guid id);
}
```

При удалении метода из интерфейса не забудьте удалить его и из всех реализующих классов. Компилятор не будет выдавать ошибку, если вы его оставите, но вам придется продолжать его обслуживать.

10.2.2. Паттерн Strangler. Уровень класса

Паттерн Strangler можно применить и на уровне класса. Если у вас есть класс, для которого вы хотите провести рефакторинг, но беспокоитесь, что на его изменение уйдет слишком много времени, можете добавить новый класс, перемещать по очереди вызывающие объекты и, наконец, удалить старый.

Вы можете найти несколько таких примеров в кодовой базе онлайн-бронирования ресторанов. В одном из них я обнаружил, что перестарался с добавлением новой функциональности[1]. Мне нужно было смоделировать распределение резервирований по таблицам в зависимости от заданного времени, поэтому я добавил универсальный класс `Occurrence<T>`, который мог связать любой тип объекта со временем (листинг 10.12).

Листинг 10.12. Конструктор и свойства класса Occurrence<T>. Этот класс связывает любой тип объекта со временем, но оказалось, что я перестарался (Restaurant/4c9e781/Restaurant.RestApi/Occurrence.cs)

```
public Occurrence(DateTime at, T value)
{
    At = at;
    Value = value;
}

public DateTime At { get; }
public T Value { get; }
```

[1] Да, даже когда я изо всех сил стараюсь следовать всем методам из этой книги, я тоже могу ошибаться. Несмотря на правило делать самое простое, что могло бы сработать [22], я иногда усложняю код, так как «мне это позже обязательно понадобится». Но бить себя по голове за свою ошибку непродуктивно. Нужно просто признать ее и попробовать исправить.

После реализации функций, в которых требовался класс `Occurrence<T>`, я понял, что на самом деле мне не нужно, чтобы он был универсальным. Весь код, в котором использовался объект, содержал набор таблиц с соответствующими резервированиями.

Дженерики немного усложняют код, и хотя иногда они бывают полезны при определенных обстоятельствах, они делают некоторые вещи более абстрактными. Например, у меня был метод с сигнатурой из листинга 10.13.

Листинг 10.13. Метод, возвращающий универсальный тип с тройным вложением. Слишком абстрактно? (Restaurant/4c9e781/Restaurant.RestApi/MaitreD.cs)

```
public IEnumerable<Occurrence<IEnumerable<Table>>> Schedule(
  IEnumerable<Reservation> reservations)
```

Примите во внимание рекомендации из раздела 8.1.5. Глядя на типы, сможете ли вы понять, что делает метод `Schedule`? Что вы думаете о `IEnumerable<Occurrence<IEnumerable<Table>>>`? Разве метод не был бы проще, имей он сигнатуру из листинга 10.14?

Листинг 10.14. Метод, возвращающий коллекцию объектов TimeSlot. Это тот же метод, что и в листинге 10.13, но с более конкретным типом возвращаемого значения (Restaurant/7213b97/Restaurant.RestApi/MaitreD.cs)

```
public IEnumerable<TimeSlot> Schedule(
  IEnumerable<Reservation> reservations)
```

`IEnumerable<TimeSlot>` кажется более подходящим типом возвращаемого значения, поэтому я хотел провести рефакторинг от класса `Occurrence<T>` к классу `TimeSlot`.

Кода, использующего `Occurrence<T>`, уже было много, и мне было неудобно выполнять такой рефакторинг за короткий промежуток времени. Вместо этого я решил использовать шаблон Strangler: сначала добавить новый класс `TimeSlot`, затем переносить вызывающие объекты один за другим и, наконец, удалить класс `Occurrence<T>`.

Итак, я добавил в базу кода `TimeSlot`. В листинге 10.15 показаны его конструктор и свойства, чтобы вы могли понять, как он выглядит.

Листинг 10.15. Конструктор и свойства класса TimeSlot (Restaurant/4c9e781/Restaurant.RestApi/TimeSlot.cs)

```
public TimeSlot(DateTime at, IReadOnlyCollection<Table> tables)
{
  At = at;
  Tables = tables;
}

public DateTime At { get; }
public IReadOnlyCollection<Table> Tables { get; }
```

Как только я добавил этот класс, я мог закоммитить его в Git и объединить с мастер-веткой. Это не нарушило бы никакой функциональности.

Затем я мог бы начать перенос кода с использования `Occurrence<T>` на использование `TimeSlot`. Я начал с нескольких вспомогательных методов (листинг 10.16).

Листинг 10.16. Сигнатура вспомогательного метода, принимающего параметр Occurrence. Сравните с листингом 10.17 (Restaurant/4c9e781/Restaurant.RestApi/ScheduleController.cs)

```
private TimeDto MakeEntry(Occurrence<IEnumerable<Table>> occurrence)
```

Вместо того чтобы он принимал параметр `Occurrence<IEnumerable<Table>>`, я хотел изменить его, чтобы он принимал параметр `TimeSlot` (листинг 10.17).

Листинг 10.17. Сигнатура вспомогательного метода, принимающего параметр TimeSlot. Сравните с листингом 10.16 (Restaurant/0030962/Restaurant.RestApi/ScheduleController.cs)

```
private static TimeDto MakeEntry(TimeSlot timeSlot)
```

Код, который *вызывал* вспомогательный метод `MakeEntry`, сам был вспомогательным методом, который получал аргумент `IEnumerable<Occurrence<IEnumerable<Table>>>`, но я хотел постепенно переносить вызывающие объекты. Это можно было сделать, добавив метод временного преобразования в листинге 10.18. Он поддерживает преобразование между старым и новым классом. Как только я завершил миграцию Strangler, то удалил его вместе с самим классом.

Листинг 10.18. Метод временного преобразования из Occurrence в TimeSlot (Restaurant/0030962/Restaurant.RestApi/Occurrence.cs)

```
internal static TimeSlot ToTimeSlot(
    this Occurrence<IEnumerable<Table>> source)
{
    return new TimeSlot(source.At, source.Value.ToList());
}
```

Мне пришлось перенести метод `Schedule` из листинга 10.13 в версию листинга 10.14. Поскольку у меня было несколько вызывающих объектов, я хотел перенести каждый отдельно, делая коммит изменений в Git. Это означало, что мне нужно, чтобы две версии `Schedule` существовали рядом в течение ограниченного времени. Это невозможно, так как они различаются только типом возвращаемого значения, а C# не поддерживает перегрузку типа возвращаемого значения.

Чтобы обойти эту проблему, я сначала использовал рефакторинг переименования метода (Rename Method) [34], чтобы переименовать исходный метод `Schedule` в `ScheduleOcc`[1]. Затем я скопировал и вставил его, изменил тип возвращаемого значения и снова изменил имя нового метода на `Schedule`. Теперь у меня был исходный метод `ScheduleOcc` и новый с улучшенным типом возвращаемого значения, но без вызывающих объектов. Опять же это место, где вы можете коммитить свои изменения и выполнить слияние с мастер-веткой.

Теперь я мог с помощью двух методов переносить вызывающие объекты по одному, сверяя свои изменения с Git для каждого. Опять же эту задачу вы можете выполнять постепенно, не мешая другой работе. Как только все вызывающие объекты вызвали новый метод `Schedule`, я удалил метод `ScheduleOcc`.

`Schedule` был не единственным методом, который возвращал данные, использующие `Occurrence<T>`, но я мог перенести другие методы в `TimeSlot`, применяя тот же прием.

По завершении миграции я удалил класс `Occurrence<T>`, включая вспомогательный метод преобразования из листинга 10.18.

---

[1] Occ — от англ. occurrence — «возникновение».

Во время этого процесса промежуток между коммитами ни разу не занимал более пяти минут и все коммиты оставляли систему в согласованном состоянии, которое можно было интегрировать и развернуть.

10.3. ВЕРСИОНИРОВАНИЕ

По возможности изучите спецификацию семантического управления версиями (Semantic Versioning) [83]. Это займет у вас около 15 минут. Если коротко, то в спецификации используется схема *major.minor.patch*. Вы увеличиваете *мажорную* версию только при внесении критических изменений, *минорную* — при введении новой функциональности, а увеличение версии *патча* — при исправлении ошибки.

Даже если вы не будете использовать семантическое управление версиями, я считаю, что это даст вам более четкое представление о критических и некритических изменениях. При разработке и сопровождении монолитного приложения без API критические изменения могут быть неважны, но, как только другой код начинает зависеть от вашего, это будет иметь значение.

Так происходит независимо от того, где находится зависимый код. Очевидно, что обратная совместимость очень важна, если у вас есть внешние платежеспособные клиенты, зависящие от вашего API. Но даже если система, зависящая от вашего кода, — это «просто» еще одна кодовая база вашей организации, все равно стоит подумать о совместимости.

Каждый раз при нарушении совместимости вам нужно координировать свои действия с вызывающими. Иногда это происходит быстро и внезапно, например: «Ваше последнее изменение сломало наш код!» Было бы лучше, если бы вы могли заранее предупредить клиентов.

Но будет куда лучше, если вы сможете избежать критических изменений. В семантическом управлении версиями это означает использование одной и той же мажорной версии долгое время. На то, чтобы привыкнуть к этому, может уйти некоторое время.

Когда-то я поддерживал библиотеку с открытым исходным кодом, которая оставалась в основной версии 3 более четырех лет! Последний выпуск версии 3 был 3.51.0. Судя по всему, за эти четыре года мы добавили 51 новую функцию, но так как мы не нарушили совместимость, то и не увеличили мажорную версию.

10.3.1. Заблаговременное предупреждение

При *необходимости* нарушить совместимость будьте внимательны. По возможности всегда предупреждайте пользователей заранее. Рассмотрите иерархию коммуникации из подраздела 8.1.7, чтобы выяснить лучший путь взаимодействия.

Например, некоторые языки программирования позволяют объявлять методы устаревшими с помощью аннотаций. В .NET это называется `[Obsolete]`, в Java — `@Deprecated` (листинг 10.19). Это приведет к тому, что компилятор C# выдаст предупреждение для всего кода, вызывающего этот метод.

Листинг 10.19. Устаревший метод. Атрибут [Obsolete] определяет метод как устаревший и дает подсказку об альтернативных действиях (Restaurant/4c9e781/Restaurant.RestApi/CalendarController.cs)

```
[Obsolete("Use Get method with restaurant ID.")]
[HttpGet("calendar/{year}/{month}")]
public Task<ActionResult> LegacyGet(int year, int month)
```

Если вы понимаете, что *должны* нарушить совместимость, подумайте, можете ли вы объединить более одного критического изменения в один выпуск. Это не всегда хорошая идея, но иногда важно сделать именно так. Каждый раз, внося критическое изменение, вы вынуждаете разработчиков клиентских приложений с ними разбираться. Если у вас уже есть несколько небольших критических изменений, то объединение их в один выпуск существенно упростит работу вашим коллегам.

С другой стороны, не стоит выпускать несколько критических изменений, если каждое из них вынуждает разработчиков клиентских приложений к масштабной переработке. Проявите благоразумие; это, в конце концов, *искусство* программной инженерии.

10.4. ЗАКЛЮЧЕНИЕ

Вы работаете с существующими кодовыми базами. Добавляя новые функции, улучшая уже существующие или исправляя ошибки, вы вносите изменения в существующий код. Будьте внимательны и делайте все постепенно.

Если вы работаете над функциональностью, реализация которой требует много времени, может возникнуть соблазн разработать ее в отдельной ветке. Не делайте этого — это приведет к большим проблемам при слиянии. Просто спрячьте функцию за функциональным флагом и чаще выполняйте интегрирование [49].

Если вы хотите провести масштабный рефакторинг, подумайте об использовании шаблона Strangler. Вместо того чтобы выполнять редактирование сразу на месте, изменяйте код, позволив новому и старому вариантам сосуществовать некоторое время. Это позволит вам постепенно переносить вызывающие объекты. Вы даже можете выполнять это как задачу обслуживания, чередуя ее с другой работой. Удаляйте старый метод или класс только после завершения миграции.

Если метод или класс — части опубликованного объектно-ориентированного API, удаление метода или класса может стать критическим изменением. В этом случае нужно явно рассмотреть версионирование. Сначала объявите API устаревшим, чтобы предупредить пользователей о предстоящем изменении, а удаляйте его только при выпуске новой мажорной версии.

11 РЕДАКТИРОВАНИЕ МОДУЛЬНЫХ ТЕСТОВ

Немногие кодовые базы запускаются с помощью практик из первой части книги. У них объемные методы, высокая степень сложности, плохая инкапсуляция и небольшое автоматизированное тестовое покрытие. Мы называем такие кодовые базы *легаси-кодом*. Я не буду вдаваться в подробности, так как на эту тему уже есть отличная книга «Ускоряйся! Наука DevOps. Эффективная работа с унаследованным кодом» [27].

11.1. РЕФАКТОРИНГ МОДУЛЬНЫХ ТЕСТОВ

Если у вас уже есть надежный набор автотестов, вы можете применить многие уроки *рефакторинга* [34]. В этой книге мы обсуждаем, как изменить структуру существующего кода без изменения его поведения. Большинство описанных методов встроены в современные IDE, например переименование, извлечение вспомогательных методов, перемещение кода и т. д. На эту тему я тоже не буду тратить много времени, так как она подробно освещена в других источниках [34].

11.1.1. Смена подушки безопасности

В то время как «Рефакторинг» [34] объясняет, как изменить структуру продакшен-кода, и использует в качестве подушки безопасности автоматизированный набор тестов, книга «Шаблоны тестирования xUnit» [66] идет с подзаголовком «Рефакторинг тестового кода»[1].

Вы пишете тестовый код, чтобы убедиться, что ваш продакшен-код работает. Как я уже говорил, при написании кода легко допустить ошибку. Как же тогда узнать об ошибках в тестовом коде?

Вручную это сделать почти невозможно, но некоторые из описанных ранее методов могут вам помочь. Используя тесты в качестве драйвера для продакшен-кода, вы вступаете в своего рода двойную бухгалтерию [63], где тесты поддерживают продакшен-код на месте, а он предоставляет обратную связь о тестах.

Еще один инструмент, который можно использовать, — чек-лист «красный, зеленый, рефакторинг». Когда тест проходит неудачно, вы знаете, что он на самом деле проверяет то, что вы хотите проверить. Если вы никогда не редактируете тест, вы можете доверять ему.

Что произойдет, если вы отредактируете тестовый код?

Чем больше вы редактируете тестовый код, тем меньше вы можете ему доверять. Но основа рефакторинга — это набор тестов:

> *«Необходимое предварительное условие для рефакторинга [...] — это надежные тесты» [34].*

Формально говоря, вы не можете осуществить рефакторинг модульных тестов.

---

[1] Хотя, если честно, это больше книга о паттернах проектирования, чем о рефакторинге.

На практике вам иногда придется редактировать код модульных тестов. Но вы должны понимать, что, в отличие от продакшен-кода, здесь нет подушки безопасности. Всегда редактируйте тесты осторожно и сознательно.

11.1.2. Добавление нового тестового кода

Самое безопасное редактирование в тестовом коде — это добавление нового кода. Вы можете добавить совершенно новые тесты, что не снизит благонадежности существующих.

Понятно, что добавление совершенно нового тестового класса может быть самым изолированным изменением, которое вы можете сделать. Но вы можете добавить и новые тестовые методы к существующему классу. Предполагается, что методы тестирования не зависят друг от друга, поэтому добавление нового не должно влиять на существующие.

Вы также можете добавить тест-кейсы к параметризованному тесту. Если, например, у вас есть тест-кейсы из листинга 11.1, можете добавить еще одну строку кода (листинг 11.2). Это вряд ли будет опасно.

Листинг 11.1. Параметризованный тестовый метод с тремя тест-кейсами. В листинге 11.2 приведен обновленный код после добавления нового кейса (Restaurant/b789ef1/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Theory]
[InlineData(null, "j@example.net", "Jay Xerxes", 1)]
[InlineData("not a date", "w@example.edu", "Wk Hd", 8)]
[InlineData("2023-11-30 20:01", null, "Thora", 19)]
public async Task PostInvalidReservation(
```

Листинг 11.2. Тестовый метод с новым тест-кейсом (выделен, в сравнении с листингом 11.1) (Restaurant/745dbf5/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Theory]
[InlineData(null, "j@example.net", "Jay Xerxes", 1)]
[InlineData("not a date", "w@example.edu", "Wk Hd", 8)]
[InlineData("2023-11-30 20:01", null, "Thora", 19)]
[InlineData("2022-01-02 12:10", "3@example.org", "3 Beard", 0)]
public async Task PostInvalidReservation(
```

Вы можете добавлять утверждения к уже существующим тестам. В листинге 11.3 приведен модульный тест с одним утверждением, а в листинге 11.4 приведен тот же тест после добавления еще двух.

Листинг 11.3. Одно утверждение в тестовом методе. В листинге 11.4 приведен обновленный код после добавления дополнительных утверждений (Restaurant/36f8e0f/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
Assert.Equal(
    HttpStatusCode.InternalServerError,
    response.StatusCode);
```

Листинг 11.4. Этап проверки после добавления еще двух утверждений по сравнению с листингом 11.3. Добавленные строки выделены (Restaurant/0ab2792/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
Assert.Equal(
    HttpStatusCode.InternalServerError,
    response.StatusCode);
Assert.NotNull(response.Content);
var content = await response.Content.ReadAsStringAsync();
Assert.Contains(
    "tables",
    content,
    StringComparison.OrdinalIgnoreCase);
```

Эти два примера взяты из тест-кейса, который проверяет, что произойдет при переполнении ресторана. В листинге 11.3 тест только подтверждает, что ответ HTTP — `500 Internal Server Error`[1]. Два новых утверждения проверяют, содержит ли ответ HTTP подсказку о том, что может быть не так, например сообщение `No tables available` (Нет доступных таблиц).

Я часто встречаю программистов, которые усвоили, что метод тестирования может содержать только одно утверждение. Наличие нескольких утверждений называется рулеткой утверждений. Считаю, что такой подход все слишком упрощает. Вы можете рассматривать добавление новых утверждений как усиление постусловий.

[1] Все еще спорное проектное решение. Для получения более подробной информации см. сноску в подразделе 6.2.1.

С утверждением в листинге 11.3 любой ответ `500 Internal Server Error` прошел бы тест. Это будет включать настоящую ошибку, например отсутствующую строку подключения, что может привести к ложноотрицательным результатам, так как основная ошибка может остаться незамеченной.

Добавление утверждений усиливает постусловия. Любая старая ошибка `500 Internal Server Error` больше не подойдет. HTTP-ответ тоже должен сопровождаться содержимым, в котором должна быть как минимум строка `tables`.

Это напоминает мне принцип подстановки Лисков [60]. Есть много способов выразить его, и в одном из них мы говорим, что подтипы могут ослаблять предусловия и усиливать постусловия, но не наоборот. Вы можете думать о подтипах как об упорядочении, и вы можете думать о времени так же (рис. 11.1). Как подтип зависит от своего супертипа, так и момент времени зависит от предыдущих моментов времени. Продвигаясь вперед во времени, вы можете усиливать постусловия системы точно так же, как подтипу разрешается усиливать постусловие супертипа.

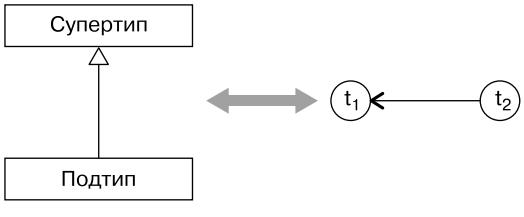


Рис. 11.1. Иерархия типов образует ориентированный граф, на что указывает стрелка от подтипа к супертипу. Время тоже образует ориентированный граф, на что указывает стрелка от точки t_2 до t_1. Оба варианта отражают способ упорядочения элементов

Посмотрите на это с другой стороны: добавление новых тестов или утверждений — нормальная практика. Удаление тестов или утверждений ослабило бы гарантии системы. Вы ведь этого не хотели бы? В данном случае кроются регрессионные ошибки и критические изменения.

11.1.3. Разделяйте рефакторинг тестового и продакшен-кода

При правильном выполнении многие изменения кода безопасны. Некоторые рефакторинги, описанные в книге «Рефакторинг» [34], теперь включены в современные IDE. Самые важные из них — разные операции переименования, такие как «переименовать переменную» и «переименовать метод». Некоторые включают метод извлечения или метод перемещения.

Такие рефакторинги безопасны — вы можете быть уверены, что они не изменят поведение кода. Это относится и к тестовому коду. Можете смело использовать эти рефакторинги как в продакшене, так и в тестовом коде.

Другие изменения более рискованны[1]. При внесении изменений в продакшен-код хороший набор тестов предупредит вас о любых проблемах. При внесении изменений в свой тестовый код гарантий никаких нет.

Вернее, это не совсем так....

Тестовый и продакшен-код связаны друг с другом (рис. 11.2). Если вы добавили в продакшен код с ошибкой, но *не изменяли тесты*, они могут предупредить вас о проблеме. Нет гарантий, что это произойдет, так как у вас может не быть тестов, выявляющих данную ошибку, но вам может повезти. К тому же, если ошибка представляет собой регрессию, у вас уже должен быть тест этого сценария.

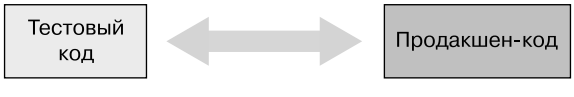


Рис. 11.2. Тестовый и продакшен-код связаны

Точно так же, если вы отредактируете тестовый код, *не изменив продакшен*, ошибка может проявиться в качестве непройденного теста.

[1] Например, добавить параметр.

Опять же нет никакой гарантии того, что это произойдет. Например, вы можете сначала использовать метод извлечения, чтобы превратить набор утверждений во вспомогательный метод. Сам по себе этот рефакторинг безопасен. Но представьте, что теперь вы ищете другие вхождения этого набора утверждений и заменяете их вызовом нового вспомогательного метода. Это уже не так безопасно, так как вы можете ошибиться. Возможно, вы замените небольшую *вариацию* набора утверждений вызовом вспомогательного метода. Но если этот вариант подразумевал усиленный набор постусловий, вы просто непреднамеренно ослабите тесты.

От некоторых ошибок трудно уберечься, другие же сразу бросаются в глаза. Если вместо ослабления постусловий вы случайно усилите их слишком сильно, тесты могут провалиться. Позже вы можете проверить неудачные тест-кейсы и понять, где допустили ошибку.

Поэтому, *когда* вам нужно провести рефакторинг тестового кода, постарайтесь сделать это, не касаясь продакшена.

Вы можете думать об этом правиле как о переходе от продакшен-кода к тестовому и обратно к продакшену (рис. 11.3).

Например, я работал над кодовой базой ресторана, чтобы добавить возможность связи через электронную почту. Я уже реализовал поведение, согласно которому при резервировании система должна отправить вам электронное письмо с подтверждением.

Взаимодействие с окружением лучше всего моделировать как полиморфный тип, и базовым классам я предпочитаю интерфейсы, подобные представленным в коде листинга 11.5.

Листинг 11.5. Начальная итерация интерфейса IPostOffice (Restaurant/b85ab3e/Restaurant.RestApi/IPostOffice.cs)

```
public interface IPostOffice
{
    Task EmailReservationCreated(Reservation reservation);
}
```

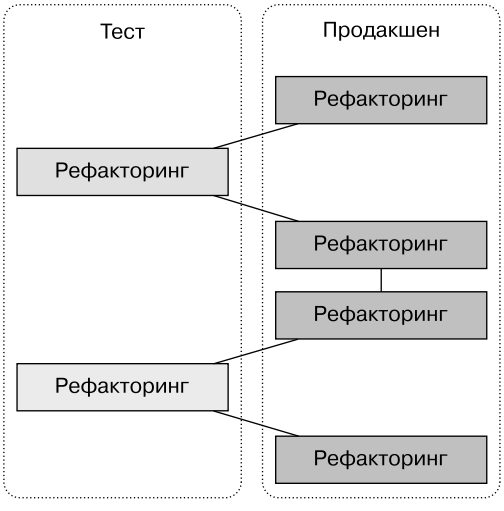


Рис. 11.3. Выполняйте рефакторинг тестового кода отдельно от продакшена. После каждого отдельного рефакторинга делайте коммит. Безопаснее вносить изменения в продакшен-код, поэтому вы можете проводить его рефакторинг чаще, чем тестового. Другие, более безопасные изменения, такие как переименование метода, могут коснуться как тестового, так и продакшен-кода. Они не показаны на этом рисунке

В рамках модульного тестирования сценария отправки письма по электронной почте я добавил Test Spy (шпион) [66] из листинга 11.6, чтобы следить за косвенным выводом [66].

Листинг 11.6. Первоначальная версия SpyPostOffice, реализующая версию IPostOffice из листинга 11.5 (Restaurant/b85ab3e/Restaurant.RestApi.Tests/SpyPostOffice.cs)

```
public class SpyPostOffice : Collection<Reservation>, IPostOffice
{
    public Task EmailReservationCreated(Reservation reservation)
    {
        Add(reservation);
        return Task.CompletedTask;
    }
}
```

Обратите внимание, что `SpyPostOffice` наследуется от базового класса коллекции. Это позволяет реализации добавить (`Add`) к себе резервирование. Тест может использовать это поведение, чтобы проверить, вызывает ли система метод `EmailReservationCreated` (отправляет электронное письмо).

Тест может создать экземпляр `SpyPostOffice`, передать его конструкторам или методам, принимающим аргумент `IPostOffice`, проверить тестируемую систему [66] и затем проверить ее состояние (листинг 11.7).

Листинг 11.7. Утверждение, что ожидаемое (expected) резервирование находится в коллекции postOffice. Переменная postOffice служит объектом SpyPostOffice (Restaurant/b85ab3e/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
Assert.Contains(expected, postOffice);
```

Определив поведение, я начал работу над связанной функциональностью. Система должна отправить электронное письмо и при удалении резервирования. Для этого я добавил новый метод в интерфейс `IPostOffice` (листинг 11.8).

Листинг 11.8. Вторая итерация интерфейса IPostOffice. Новый в сравнении с листингом 11.5 метод выделен (Restaurant/1811c8e/Restaurant.RestApi/IPostOffice.cs)

```
public interface IPostOffice
{
    Task EmailReservationCreated(Reservation reservation);

    Task EmailReservationDeleted(Reservation reservation);
}
```

Поскольку я добавил новый метод в интерфейс `IPostOffice`, мне пришлось реализовать этот метод и в классе `SpyPostOffice`. И `EmailReservationCreated`, и `EmailReservationDeleted` принимают аргумент `Reservation`, поэтому я мог бы просто добавить резервирование в сам Test Spy [66].

Но когда я начал писать модульный тест для нового поведения, то понял: хотя я и могу написать утверждение как в листинге 11.7, у меня

получится лишь проверить, содержит ли Test Spy [66] ожидаемое резервирование. Я не мог проверить, как оно туда попало, независимо от того, с помощью какого метода (`EmailReservationCreated` или `EmailReservationDeleted`) я добавил Test Spy.

Для этого мне пришлось улучшить «чувствительность» `SpyPostOffice`.

Я уже приступил к ряду изменений, которые коснулись продакшена. Интерфейс `IPostOffice` — его часть, и у него была продакшен-реализация (`SmtpPostOffice`). Я был в процессе внесения изменений в продакшен-код, как вдруг понял, что мне нужно провести рефакторинг тестового.

Это одна из многих причин, по которым Git настолько меняет правила игры, даже для индивидуальной разработки. Это пример маневренности, которую он предлагает. Я просто спрятал[1] свои изменения и независимо отредактировал класс `SpyPostOffice` (листинг 11.9).

Листинг 11.9. Рефакторинг SpyPostOffice (фрагмент). Класс Observation — вложенный класс (не показан), содержащий Event и Reservation (Restaurant/b587eef/Restaurant.RestApi.Tests/SpyPostOffice.cs)

```
internal class SpyPostOffice :
    Collection<SpyPostOffice.Observation>, IPostOffice
{
    public Task EmailReservationCreated(Reservation reservation)
    {
        Add(new Observation(Event.Created, reservation));
        return Task.CompletedTask;
    }

    internal enum Event
    {
        Created = 0
    }
```

[1] git stash сохраняет ваши ненужные файлы в «скрытом» коммите и сбрасывает репозиторий в HEAD. По окончании работы вы можете получить этот коммит с помощью git stash pop.

Я добавил вложенный класс `Observation`, чтобы отслеживать тип взаимодействия и само резервирование, и изменил базовый класс на коллекцию объектов `Observation`.

Это нарушило некоторые из моих тестов, так как утверждение из кода в листинге 11.7 будет искать объект `Reservation` в коллекции объектов `Observation`. Это не проверка типа, поэтому мне пришлось слегка повозиться.

Мне удалось сделать это, не касаясь продакшен-кода. В итоге все тесты были пройдены. Это не гарантирует того, что я не ошибся при рефакторинге, но по крайней мере устраняет определенную категорию ошибок[1].

После рефакторинга тестового кода я извлек скрытые изменения и продолжил с места, на котором остановился. В листинге 11.10 приведен обновленный класс `SpyPostOffice`.

Хотя эти изменения включали в себя редактирование тестового кода, они были безопаснее, так как были лишь дополнениями. Мне не пришлось вносить изменения в существующий тестовый код.

Листинг 11.10. Обновленный класс SpyPostOffice теперь реализует версию IPostOffice из листинга 11.8 (Restaurant/1811c8e/Restaurant.RestApi.Tests/SpyPostOffice.cs)

```
internal class SpyPostOffice :
    Collection<SpyPostOffice.Observation>, IPostOffice
{
    public Task EmailReservationCreated(Reservation reservation)
    {
        Add(new Observation(Event.Created, reservation));
        return Task.CompletedTask;
    }

    public Task EmailReservationDeleted(Reservation reservation)
    {
        Add(new Observation(Event.Deleted, reservation));
        return Task.CompletedTask;
    }
```

[1] Например, что изменения в тестах случайно усилили некоторые предварительные условия.

```
internal enum Event
{
    Created = 0,
    Deleted = 1
}
```

11.2. НЕПРОЙДЕННЫЕ ТЕСТЫ

Если вам *нужно* редактировать и тестовый, и продакшен-код одновременно, рассмотрите возможность проверки тестов, специально заставив их временно не работать.

Удивительно легко писать тавтологические утверждения [105]. Они никогда не подведут, даже если продакшен-код неисправен.

Не доверяйте тесту, который не был неудачным. Если вы изменили тест, вы можете временно изменить тестируемую систему, чтобы сделать тест непройденным. Например, закомментируйте некоторый продакшен-код или верните жестко запрограммированное значение. Затем запустите тест, который отредактировали, и убедитесь, что он не проходит.

Опять же Git дает нам маневренность. При необходимости изменить тесты и продакшен-код одновременно вы можете внести свои изменения как staged-контент (предшествующее коммиту состояние, эдакий буфер. — *Примеч. ред.*) и только тогда саботировать тестируемую систему. Когда вы увидите, что тест завершен неудачей, вы можете отбросить изменения в своем рабочем каталоге и закоммитить staged-изменения.

11.3. ЗАКЛЮЧЕНИЕ

Будьте внимательны при редактировании кода модульного теста. У вас нет никакой подстраховки.

Некоторые изменения относительно безопасны. Добавление новых тестов, новых утверждений или новых тест-кейсов, как правило, тоже. Применение рефакторинга, встроенного в вашу IDE, тоже может быть безопасным.

Другие изменения в тестовом коде более рискованны, хотя и могут быть желательны. Вы должны поддерживать тестовый код. Важно, чтобы он, как и продакшен-код, умещался в вашей голове. Поэтому иногда вам следует реорганизовать тестовый код, чтобы улучшить его внутреннюю структуру.

Если вы, например, решаете проблему дублирования, извлекая вспомогательные методы, убедитесь, что вы редактируете только тестовый код и не трогаете при этом продакшен. Отметьте эти изменения для тестового кода в Git как отдельные коммиты. Это *не гарантирует того*, что вы не допустили ошибок в тестовом коде, но повышает ваши шансы на успех.

12 УСТРАНЕНИЕ НЕПОЛАДОК

Профессиональная разработка ПО включает не только разработку функциональности. Есть еще встречи, отчеты о времени, действия по обеспечению соответствия и... дефекты.

Вы постоянно сталкиваетесь с ошибками и проблемами. Ваш код не компилируется, программа не делает то, что должна, работает слишком медленно и пр.

Чем лучше вы решаете эти проблемы, тем продуктивнее работаете. Большинство ваших навыков устранения дефектов могут быть основаны на зыбучих песках индивидуального опыта [4], но *есть* приемы, которые вы можете изучить.

В этой главе я поделюсь некоторыми из них.

12.1. ПОНИМАНИЕ

Лучший мой совет таков.

> Постарайтесь понять, что происходит.

Если вы не понимаете, почему что-то не работает[1], сделайте это приоритетом. Есть такое понятие, как случайное программирование [50]: напишите много кода и наблюдайте, что в итоге приживется. Когда похоже, что код работает, разработчики переходят к следующей задаче. Либо они не понимают, почему код работает, либо они могут не понять, что на самом деле это не так.

Если вы понимаете код с самого начала, скорее всего, вам будет легче устранять дефекты.

12.1.1. Научный подход

Когда дефект становится очевидным, большинство программистов сразу же переходят в режим устранения неполадок. Они хотят *решить* проблему. Для разработчиков, которые программируют случайно [50], решение проблемы обычно включает в себя использование разных случайных методов, которые, возможно, когда-то срабатывали для решения аналогичной задачи. Если первый метод не сработал, разработчик переходит к следующему. Этот процесс может включать в себя перезапуск службы, перезагрузку компьютера, запуск инструмента с повышенными привилегиями, изменение небольших фрагментов кода, вызов малопонятных подпрограмм и т. д. Когда кажется, что проблема исчезла, они прекращают работу, не пытаясь понять причину [50].

Думаю, не стоит говорить, что для решения проблем этот способ не очень подходит.

Чтобы решить проблему, вы в первую очередь должны понять, почему она возникла. Если вы не знаете, обратитесь за помощью. Но, как правило, разработчик примерно представляет, где кроется неполадка. В этом случае примените следующий научный метод [82]:

- сделать прогноз *(гипотеза)*;
- провести эксперимент;
- сравнить результат с прогнозом. Повторять, пока не поймете, что происходит.

[1] Или если вы не понимаете, почему что-то работает.

Не пугайтесь термина «научный подход». Вам не нужно надевать лабораторный халат или планировать рандомизированное контролируемое двойное слепое исследование. Но постарайтесь выдвинуть опровержимую гипотезу. Это может быть просто прогноз, например: «Если я перезагружу компьютер, проблема исчезнет» или «Если я вызову эту функцию, возвращаемое значение будет 42».

Разница между этой техникой и «случайным программированием» в цели. Вы делаете это, не чтобы решить проблему, а чтобы понять ее и разобраться с ней.

Типичный эксперимент — модульный тест с гипотезой о том, что при его запуске он потерпит неудачу. Подробнее об этом — в подразделе 12.2.1.

12.1.2. Упрощение

Подумайте, может ли *удаление* части кода решить проблему.

Самый распространенный метод решения проблемы — добавление дополнительного кода. Так, если проблема единичная, ее нужно решить с помощью включения большего количества кода для обработки этого конкретного случая.

Иногда это полезно, но, скорее всего, проблема будет проявлением основной ошибки реализации. Вы будете удивлены тому, как часто *упрощение* кода может решить ваши проблемы.

Я знаю много примеров такого «предвзятого отношения» в нашей отрасли. Люди, решающие проблемы, которых у меня никогда не было, так как я усердно работаю над простотой своего кода:

- те, кто разрабатывает сложные контейнеры внедрения зависимостей [25], вместо того чтобы просто составлять графы объектов в коде;
- те, кто разрабатывает сложные «библиотеки mock-объектов», вместо того чтобы писать в основном чистые функции;
- те, кто создает сложные схемы восстановления пакетов, а не просто проверяет зависимости в системе контроля версий;

- те, кто использует расширенные инструменты сравнения вместо более частого слияния;
- те, кто использует запутанные объектно-реляционные преобразователи (ORM), вместо того чтобы изучать (и поддерживать) SQL.

Я могу еще долго продолжать этот список.

Честно говоря, найти простое решение иногда очень *трудно*. Например, у меня ушло десять лет на создание все более сложных вариантов в объектно-ориентированном коде, прежде чем я нашел решения попроще. Оказывается, многие вещи, которые сложны в традиционном ООП, становятся простыми в функциональном программировании. Открыв для себя некоторые из этих концепций, я нашел способы их использования в рамках объектно-ориентированного программирования.

Дело в том, что аббревиатура KISS[1] бесполезна сама по себе. Как только с ее помощью можно что-то упростить?

Часто приходится много *думать*, чтобы добиться простоты[2], но старайтесь стремиться к ней в любом случае. Подумайте, можно ли решить проблему, *удалив* код.

12.1.3. Метод утенка

Прежде чем мы обсудим специфические методы решения проблем, я хочу поделиться некоторыми общими приемами. Нет ничего необычного в том, чтобы застрять на какой-то проблеме. Как вы из этого выпутываетесь?

Возможно, вы анализируете проблему, не имея ни малейшего представления о том, как действовать дальше. Как я уже говорил, в первую очередь важно понять проблему. Что делать, если у вас в голове чистый лист?

---

[1] «Будь проще, глупец!» (Keep it simple, stupid).

[2] Рич Хикки обсуждает простоту в выступлении Simple Made Easy [45]. Я во многом обязан своей точкой зрения на простоту именно ему.

Если у вас плохо с тайм-менеджментом, вы можете застрять с проблемой надолго, так что *контролируйте* свое время. Создайте временны́е рамки для процесса. Например, выделите 25 минут на анализ проблемы. Если вы не добились результата, сделайте перерыв.

Во время перерыва всегда отходите от компьютера. Сходите за кофе. Когда вы встаете со стула и отходите от монитора, в вашем мозге активизируются совсем другие процессы. Через пару минут отвлечения от проблемы вы, скорее всего, начнете думать о чем-то другом. Возможно, вы встретите коллегу. Возможно, обнаружите, что нужно заправить кофемашину. Что бы ни произошло, все это временно отвлекает вас от проблемы. Часто этого достаточно, чтобы взглянуть на волнующий вас вопрос свежим взглядом.

Я бесчисленное количество раз возвращался к проблеме после прогулки, только для того чтобы понять, что думал неправильно.

Если небольшой прогулки недостаточно, обратитесь за помощью, допустим, к своему коллеге.

Я сталкивался с этим достаточно часто: начинаю объяснять проблему, но сам же себя обрываю на полуслове: «Я придумал!» Простое объяснение проблемы приводит к новому пониманию.

Но если коллеги поблизости нет, вы можете попытаться объяснить проблему игрушечному утенку (рис. 12.1).

Рис. 12.1. Игрушечный утенок. Поговорите с ним. Это поможет решить ваши проблемы

На самом деле это не обязательно должен быть игрушечный утенок. Но этот метод известен как *метод утенка*, поскольку один программист действительно разговаривал с такой игрушкой [50].

Вместо того чтобы использовать уточку, я обычно начинаю писать вопрос на Q&A-сервисе (вопросы и ответы о программировании) для айтишников Stack Overflow. Чаще всего я понимаю, в чем проблема, еще до того, как закончу формулировать вопрос[1].

Если же я *не смогу понять*, в чем проблема, то у меня уже будет письменный вопрос, который я смогу опубликовать.

12.2. ДЕФЕКТЫ

Однажды я устроился на новую работу в небольшом стартапе по разработке ПО. Вскоре я спросил своих коллег, не хотят ли они использовать разработку через тестирование. Они не пользовались ею ранее, но хотели узнать что-то новое. После того как я ввел их в курс дела, они решили, что это неплохо.

Через несколько месяцев после того, как мы внедрили TDD-разработку, руководитель решил поговорить со мной. Он заметил, что с тех пор, как мы начали использовать тесты, дефектов стало значительно меньше.

Я очень горжусь этим по сей день. Изменение качества было настолько значительным, что руководитель это заметил. Не исходя из подсчетов и бумаг, а просто потому, что это было так важно, что не могло остаться незамеченным.

Можно уменьшить количество дефектов, но нельзя их устранить. Но сделайте себе одолжение — не позволяйте им накапливаться.

> Идеальное количество дефектов равно нулю.

[1] Когда такое происходит, я не поддаюсь заблуждению об утопленных затратах. Даже если я потратил на написание вопроса какое-то время, я обычно удаляю его, так как считаю, что он вряд ли заинтересует кого-то еще.

Добиться отсутствия ошибок не так уж нереально, как может показаться. В бережливой разработке ПО это называется *встраиванием качества* (building quality in) [82]. Не откладывайте работу с дефектами на потом. В разработке программного обеспечения *«позже» значит «никогда»*.

При обнаружении ошибки сделайте ее устранение приоритетом. Отложите все остальные дела на потом[1] и вместо этого исправьте дефект.

12.2.1. Воспроизведение дефектов в виде тестов

Сначала вы можете даже не понимать, в чем проблема, но, когда вы думаете, что осознали, проведите эксперимент: понимание должно помочь вам сформулировать гипотезу, позволяющую спланировать эксперимент.

Это может быть, например, автоматизированный тест. Гипотеза в том, что, когда вы запустите его, он *провалится*. Если во время реального запуска он *действительно* не проходит, вы подтверждаете гипотезу. Как бонус у вас есть неудачный тест, который воспроизводит дефект и позже будет служить регрессионным тестом.

Если же тест прошел успешно, то эксперимент не удался и ваша гипотеза неверна. Вам нужно будет пересмотреть его, чтобы спроектировать новый эксперимент. Возможно, вам придется повторить этот процесс несколько раз.

Когда у вас наконец есть провальный тест, все, что нужно сделать, — это заставить его завершиться успешно. Иногда это может быть сложно, но, по моему опыту, обычно это не так. Самое тяжелое в устранении дефекта — понять и воспроизвести его.

Я приведу пример из системы онлайн-резервирования столиков в ресторане. Пока я проводил исследовательское тестирование, при

[1] Разве не замечательно, что с помощью Git вы можете просто сохранить свою текущую работу в хранилище (stash)?

обновлении бронирования я заметил кое-что странное (листинг 12.1). Можете ли вы определить проблему?

Проблема в том, что свойство `email` содержит `name` и наоборот. Кажется, я случайно их где-то перепутал. Это первоначальная гипотеза, но может потребоваться небольшое исследование, чтобы выяснить, где именно.

Разве я не следовал принципам разработки через тестирование? Тогда как это могло произойти?

Листинг 12.1. Обновление резервирования с помощью запроса PUT. В этом взаимодействии проявляется дефект. Можете определить его?

```
PUT /reservations/21b4fa1975064414bee402bbe09090ec HTTP/1.1
Content-Type: application/json
{
  "at": "2022-03-02 19:45",
  "email": "pan@example.com",
  "name": "Phil Anders",
  "quantity": 2
}

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
{
  "id": "21b4fa1975064414bee402bbe09090ec",
  "at": "2022-03-02T19:45:00.0000000",
  "email": "Phil Anders",
  "name": "pan@example.com",
  "quantity": 2
}
```

Проблема могла произойти из-за того, что я реализовал `SqlReservationsRepository`[1] как Humble Object («скромный объект») [66]. Это настолько простой объект, что вы можете решить не тестировать его. Я часто использую эмпирическое правило: если цикломатическая сложность равна 1, тест (тоже с цикломатической сложностью 1) может быть неоправданным.

[1] См. листинг 4.19.

Но вы все равно можете ошибиться, даже если цикломатическая сложность равна 1. В листинге 12.2 приведен пример дефектного кода. Попробуйте определить проблему.

Учитывая, что вы уже знаете, в чем проблема, вы, вероятно, можете догадаться, что конструктор `Reservation` ожидает аргумента `email` перед параметром `name`. Но, поскольку оба параметра объявлены как `string`, компилятор не будет выдавать ошибку, если вы случайно поменяете их местами. Это еще один пример кода со строковой типизацией [3], которого нужно избегать[1].

Листинг 12.2. Фрагмент кода, вызывающий ошибку из листинга 12.1. Можете ли вы найти ее? (Restaurant/d7b74f1/Restaurant.RestApi/SqlReservationsRepository.cs)

```
using var rdr =
    await cmd.ExecuteReaderAsync().ConfigureAwait(false);
if (!rdr.Read())
    return null;

return new Reservation(
    id,
    (DateTime)rdr["At"],
    (string)rdr["Name"],
    (string)rdr["Email"],
    (int)rdr["Quantity"]);
```

Устранить дефект легко, но если я ошибся единожды, то могу сделать это снова. Поэтому я хочу предотвратить регрессию. Перед исправлением кода нужно написать тест, который сможет воспроизвести ошибку. В листинге 12.3 приведен мой пример. Это интеграционный тест, который проверяет, получите ли вы резервирование, равное сохраненному, если обновите его в БД и впоследствии прочитаете.

[1] Один из способов избежать кода со строковой типизацией — ввести классы Email и Name, которые обертывают соответствующие строковые значения. Это предотвращает случайный обмен этих двух аргументов местами, но, как оказалось, способ не полностью надежный. Для более подробной информации вы можете обратиться к репозиторию Git с примерами кода. Если вкратце, то мне казалось, что интеграционный тест оправдан.

Конечно же, он воспроизводит ошибку, так как метод `ReadReservation` меняет местами `name` и `email` (см. листинг 12.2).

`PutAndReadRoundTrip` — интеграционный тест, в котором задействована база данных. Он новый. До сих пор в этой книге все тесты выполнялись без внешних зависимостей, а добавление БД требует обхода.

12.2.2. Медленные тесты

Язык программирования и реляционная БД имеют различные подходы к работе с данными, и преодоление разрыва между ними может привести к появлению ошибок[1], так почему бы не протестировать такой код?

В этом разделе представлена схема того, как это сделать, но есть проблема: такие тесты обычно медленные — на несколько порядков медленнее, чем внутрипроцессные тесты. Давайте проанализируем код, представленный в листинге 12.3.

Листинг 12.3. Интеграционный тест SqlReservationsRepository (Restaurant/645186b/Restaurant.RestApi.SqlIntegrationTests/SqlReservationsRepositoryTests.cs)

```
[Theory]
[InlineData("2032-01-01 01:12", "z@example.net", "z", "Zet", 4)]
[InlineData("2084-04-21 23:21", "q@example.gov", "q", "Quu", 9)]
public async Task PutAndReadRoundTrip(
    string date,
    string email,
    string name,
    string newName,
    int quantity)
{
    var r = new Reservation(
        Guid.NewGuid(),
        DateTime.Parse(date, CultureInfo.InvariantCulture),
```

[1] Сторонники объектно-реляционного отображения (ORM) могут убеждать в полезности данной технологии. Однако я уже писал, что считаю ORM пустой тратой времени: эта технология создает больше проблем, чем решает. Если вы не согласны, то можете пропустить этот раздел.

```
            new Email(email),
            new Name(name),
            quantity);
        var connectionString = ConnectionStrings.Reservations;
        var sut = new SqlReservationsRepository(connectionString);
        await sut.Create(r);

        var expected = r.WithName(new Name(newName));
        await sut.Update(expected);
        var actual = await sut.ReadReservation(expected.Id);

        Assert.Equal(expected, actual);
}
```

Время, затрачиваемое на выполнение набора тестов, имеет значение, особенно для тестов разработчиков, которые вы постоянно запускаете. У вас не получится выполнить рефакторинг с набором тестов в качестве подстраховки, если на все тесты уходит по полчаса. Цикл «красный, зеленый, рефакторинг» для разработки через тестирование не будет работать, если выполнение тестов занимает пять минут.

Максимальное время для такого набора тестов — десять секунд. Если его будет больше, вы можете просто что-либо упустить. У вас возникнет соблазн проверить почту или зайти в соцсети.

Вы можете легко превысить это время, если воспользуетесь БД. Поэтому перенесите такие испытания на второй этап. Есть много способов, но самый эффективный — просто создать второе решение Visual Studio, которое будет существовать параллельно с повседневным. Перед запуском нового решения не забудьте обновить скрипт сборки (листинг 12.4).

Листинг 12.4. Скрипт сборки, выполняющий все тесты. Файл Build.sln содержит как модульные, так и интеграционные тесты, использующие базу данных. Сравните с листингом 4.2 (Restaurant/645186b/build.sh)

```
#!/usr/bin/env bash
dotnet test Build.sln --configuration Release
```

Файл `Build.sln` содержит продакшен-код, код модульного теста и интеграционные тесты, использующие БД. Я выполняю повседневную

работу, не связанную с базой данных, в другом решении Visual Studio под именем `Restaurant.sln`.

Это решение содержит только продакшен-код и модульные тесты, поэтому выполнение всех тестов в этой среде происходит намного быстрее.

Тест в листинге 12.3 — часть кода интеграционного теста, поэтому он запускается только при запуске сценария сборки или если я явно выбираю работу в `Build.sln`, а не в `Restaurant.sln`. Иногда это разумно, если нужно выполнить рефакторинг, затрагивающий код базы данных.

Я не буду подробно описывать работу теста из листинга 12.3, так как он специфичен для взаимодействия .NET с SQL Server. Дополнительную информацию вы найдете в сопроводительной кодовой базе примеров. Если коротко: все интеграционные тесты дополнены атрибутом `[UseDatabase]`. Это пользовательский атрибут, который подключается к платформе модульного тестирования xUnit.net для запуска некоторого кода до и после каждого тест-кейса. Таким образом, каждый тест-кейс окружен таким поведением.

1. Создайте новую БД и запустите для нее все скрипты DDL[1].
2. Запустите тест.
3. Удалите БД.

Каждый тест *создает новую базу данных*, только чтобы снова удалить ее через несколько миллисекунд[2]. Это *медленно*, вот почему вам не следует выполнять такие тесты часто.

---

[1] Язык определения данных, обычно подмножество SQL. Пример см. в листинге 4.18.

[2] Всякий раз, когда я объясняю такой подход к тестированию интеграции с базой данных, я неизменно сталкиваюсь с возражением, что вместо этого можно провести тестирование путем отката транзакций. Да, только это будет значить, что вы не можете тестировать поведение транзакций базы данных. Откат транзакций может быть быстрее, но замеряли ли вы это? Я — да, и так и не нашел существенной разницы. В разделе 15.1 изложена моя общая позиция по оптимизации производительности.

Отложите медленные тесты до второго этапа конвейера сборки. Вы можете сделать их, как описано выше, или определить новые шаги, которые выполняются только на вашем сервере непрерывной интеграции.

12.2.3. Недетерминированные дефекты

Через некоторое время после запуска системы резервирования столиков в ресторане метрдотель сообщает о баге: иногда система допускает избыточное резервирование. Она не может преднамеренно воспроизвести проблему, но состояние базы данных резервирований нельзя отрицать. В некоторые дни бронирований больше, чем позволяет бизнес-логика из листинга 12.5. Так в чем же дело?

Листинг 12.5. Видимо, в этом коде есть ошибка, допускающая избыточное резервирование. В чем может быть проблема? (Restaurant/dd05589/Restaurant.RestApi/ReservationsController.cs)

```
[HttpPost]
public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));

    var id = dto.ParseId() ?? Guid.NewGuid();
    Reservation? r = dto.Validate(id);
    if (r is null)
        return new BadRequestResult();

    var reservations = await Repository
        .ReadReservations(r.At)
        .ConfigureAwait(false);
    if (!MaitreD.WillAccept(DateTime.Now, reservations, r))
        return NoTables500InternalServerError();

    await Repository.Create(r).ConfigureAwait(false);
    await PostOffice.EmailReservationCreated(r).ConfigureAwait(false);

    return Reservation201Created(r);
}
```

Вы анализируете логи приложения[1] и наконец понимаете: переполнение может возникать из-за состояния гонки. Если ближе к концу дня вместимость приближается к пределу, а два резервирования поступают одновременно, метод `ReadReservations` может возвращать один и тот же набор строк в оба потока, указывая, что резервирование возможно. Как показано на рис. 12.2, каждый поток определяет, что он может принять резервирование, поэтому добавляет новую строку в таблицу бронирований.

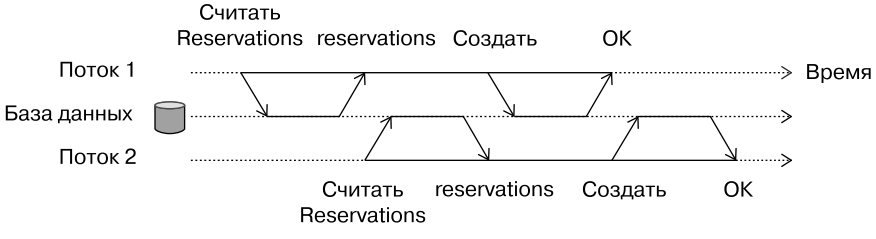


Рис. 12.2. Состояние гонки между двумя потоками (например, двумя HTTP-клиентами), одновременно пытающимися выполнить резервирование

Это явный дефект, и вам нужно воспроизвести его с помощью теста. Но проблема в том, что такое поведение недетерминированно. Автотесты должны быть определяемыми, не так ли?

Действительно, лучше, если тесты будут детерминированными, но все же допускайте, что недетерминизм тоже может быть приемлем. Как в итоге поступить?

Тесты могут дать сбой двумя способами: могут указать на сбой там, где его нет (ложноположительный результат), или могут не указать на фактическую ошибку (ложноотрицательный результат).

Ложноположительные результаты проблематичны, так как, внося шум, они снижают отношение «сигнал/шум» набора тестов. Если у вас

[1] См. подраздел 13.2.1.

есть тестовый набор, который часто дает сбои без видимых причин, вы перестаете обращать на него внимание [31].

Ложноотрицательные результаты не так страшны. Слишком много ложноотрицательных результатов может снизить ваше доверие к набору тестов, но они не вносят шума. Так вы по крайней мере знаете, что, если набор тестов дает сбой, значит, проблема *есть*.

Один из способов справиться с состоянием гонки в системе резервирования — воспроизвести его в виде недетерминированного теста (листинг 12.6).

Листинг 12.6. Недетерминированный тест, воспроизводящий состояние гонки (Restaurant/98ab6b5/Restaurant.RestApi.SqlIntegrationTests/ConcurrencyTests.cs)

```
[Fact]
public async Task NoOverbookingRace()
{
    var start = DateTimeOffset.UtcNow;
    var timeOut = TimeSpan.FromSeconds(30);
    var i = 0;
    while (DateTimeOffset.UtcNow - start < timeOut)
        await PostTwoConcurrentLiminalReservations(
            start.DateTime.AddDays(++i));
}
```

Такой метод тестирования — лишь дополнение фактического модульного теста. Он продолжает запускать метод `PostTwoConcurrentLiminalReservations` из листинга 12.7 в течение 30 секунд снова и снова, чтобы проверить, не сработает ли он. Предполагается, что если система может работать в течение 30 секунд без сбоев, то она способна вести себя корректно.

Но опять же нет никакой гарантии. Если состояние гонки возникает крайне редко, этот тест может дать ложноотрицательный результат. Но я с таким не сталкивался.

Когда я писал этот тест, он выполнялся всего несколько секунд, прежде чем закончился неудачей. Это позволяет мне думать, что 30-секундный тайм-аут достаточно безопасен. Но я признаю, что это лишь предположение; это еще один пример искусства разработки ПО.

Еще выяснилось, что в системе была такая же ошибка при обновлении существующих резервирований (в отличие от создания новых), поэтому я написал аналогичный тест для такого случая.

Листинг 12.7. Фактический метод тестирования, организованный кодом из листинга 12.6, который пытается опубликовать два конкурирующих резервирования. Практически все столики в системе уже зарезервированы (забронированы девять столиков из десяти возможных), поэтому можно сделать лишь одно резервирование (Restaurant/98ab6b5/Restaurant.RestApi.SqlIntegrationTests/ConcurrencyTests.cs)

```
private static async Task PostTwoConcurrentLiminalReservations(
    DateTime date)
{
    date = date.Date.AddHours(18.5);
    using var service = new RestaurantService();
    var initialResp =
        await service.PostReservation(new ReservationDtoBuilder()
            .WithDate(date)
            .WithQuantity(9)
            .Build());
    initialResp.EnsureSuccessStatusCode();

    var task1 = service.PostReservation(new ReservationDtoBuilder()
        .WithDate(date)
        .WithQuantity(1)
        .Build());
    var task2 = service.PostReservation(new ReservationDtoBuilder()
        .WithDate(date)
        .WithQuantity(1)
        .Build());
    var actual = await Task.WhenAll(task1, task2);

    Assert.Single(actual, msg => msg.IsSuccessStatusCode);
    Assert.Single(
        actual,
        msg => msg.StatusCode == HttpStatusCode.InternalServerError);
}
```

Приведенные листинги — примеры медленных тестов, которые лучше включать во второй этап тестирования, как описано в подразделе 12.2.2.

Есть разные способы устранения дефекта. Вы можете воспользоваться паттерном проектирования Unit of Work («Единица работы») [33] или решить эту проблему на архитектурном уровне, введя устойчивую очередь с однопоточным модулем записи, принимающим сообщения из нее. В любом случае вам нужно сериализовать операции чтения и записи, связанные с операцией.

Я выбрал прагматичное решение: использовать легковесные транзакции .NET (листинг 12.8). Окружение `TransactionScope` критической части метода `Post` эффективно сериализует[1] операции чтения и записи, что решает проблему.

Листинг 12.8. Критическая часть метода Post теперь окружена TransactionScope, которая сериализует методы чтения и записи. Новый по сравнению с листингом 12.5 код выделен (Restaurant/98ab6b5/Restaurant.RestApi/ReservationsController.cs)

```
using var scope = new TransactionScope(
    TransactionScopeAsyncFlowOption.Enabled);
var reservations = await Repository
    .ReadReservations(r.At)
    .ConfigureAwait(false);
if (!MaitreD.WillAccept(DateTime.Now, reservations, r))
    return NoTables500InternalServerError();

await Repository.Create(r).ConfigureAwait(false);
await PostOffice.EmailReservationCreated(r).ConfigureAwait(false);
scope.Complete();
```

Большинство дефектов можно воспроизвести как детерминированные тесты, но есть некоторые исключения. Одно из них — многопоточный код. Поэтому я предпочитаю недетерминированные тесты полному отсутствию тестового покрытия. Такие тесты часто приходится выполнять до тех пор, пока не истечет их время. Это даст вам уверенность в том, что они покрыли достаточно тестовых случаев. Поэтому следует поместить их на второй этап тестирования, который запускается только по запросу и как часть конвейера развертывания.

---

[1] Сериализуемость здесь означает, что транзакции БД ведут себя так, как если бы они были сериализованы одна за другой [55]. Это не имеет ничего общего с преобразованием объектов в JSON или XML и обратно.

12.3. МЕТОД БИСЕКЦИИ

Иногда некоторые дефекты сложно определить. При разработке системы резервирования столиков я столкнулся с одной проблемой, с которой я разбирался целый день. После нескольких часов неудачных попыток я наконец понял, что долгий анализ кода не поможет мне решить эту проблему. Поэтому пришлось использовать подходящий *метод*.

К счастью, он существует. Назовем его методом *бисекции*. Он прост в обращении, и этапы работы с ним следующие.

1. Найдите способ обнаружить или воспроизвести проблему.
2. Удалите половину кода.
3. Если проблема не устранена, повторите шаг 2. Если проблема исчезнет, восстановите удаленный код и удалите вторую половину. Снова повторите шаг 2.
4. Продолжайте, пока не сократите код с дефектом до размера, достаточного для понимания происходящего.

Для обнаружения проблемы вы можете использовать автотест или любой другой способ определения наличия или отсутствия проблемы. Неважно, как именно вы это сделаете, но я считаю, что автотест — самый простой способ из-за необходимости повторения.

Я часто пользуюсь им, когда «общаюсь» с резиновым утенком путем написания вопросов на Stack Overflow, так как они должны сопровождаться минимальным *рабочим примером*. В большинстве случаев процесс создания рабочего примера помогает вам найти ответ прежде, чем вы опубликуете вопрос.

12.3.1. Метод бисекции с Git

Для определения причины дефекта вы можете применить метод бисекции с Git, чтобы отыскать коммит, привнесший дефект. Я использовал его для решения проблемы, с которой столкнулся сам.

Я добавил защищенный ресурс в REST API для отображения расписания на определенный день. Метрдотель может сделать GET-запрос к этому ресурсу, чтобы увидеть расписание на день со всеми бронированиями и информацией о времени прибытия. Расписание включает имена и адреса электронной почты гостей, поэтому оно не должно быть доступно без проверки подлинности и авторизации[1].

Этот ресурс требует, чтобы клиент предоставил действительный вебтокен JSON (JWT). Я разработал эту функцию безопасности с помощью TDD. У меня было достаточно тестового кода, чтобы чувствовать себя уверенно.

Позже, в один прекрасный день, когда я взаимодействовал с развернутым REST API, я больше не мог получить доступ к этому ресурсу! Сначала я подумал, что указал недопустимый JWT, поэтому потратил несколько часов на устранение неполадок. Как итог — тупик.

Наконец на меня снизошло понимание того, что эта функция безопасности *работала*. В прошлом. А сейчас — нет. Между этими двумя известными состояниями коммит должен был внести дефект. Если бы я мог определить это конкретное изменение кода, у меня было бы больше шансов понять проблему.

К сожалению, между этими двумя крайностями было около 130 коммитов.

К счастью, с помощью коммита я нашел простой способ обнаружить дефект. Это означало, что я мог использовать функцию `bisect` в Git, чтобы точно определить коммит, вызвавший проблему.

Если у вас есть автоматизированный способ обнаружения проблемы, Git может запустить автоматический метод бисекции. Но обычно такого способа нет. Метод бисекции помогает вам найти коммит с дефектом, который *ранее оставался незамеченным*. Это значит, что, даже если у вас есть автоматизированный набор тестов, он не смог поймать эту ошибку.

---

[1] Пример см. в подразделе 15.2.5.

Поэтому Git может разделить ваши коммиты пополам в интерактивной сессии. Такую сессию можно запустить командой `git bisect start` (листинг 12.9).

Листинг 12.9. Начало сессии Git bisect. Я запустил ее с помощью Bash, но вы можете использовать любую оболочку, в которой используете Git. Чтобы все поместилось на странице, я отредактировал вывод терминала, удалив ненужные данные, которые обычно отображает Bash

```
~/Restaurant ((56a7092...))
$ git bisect start

~/Restaurant ((56a7092...)|BISECTING)
```

Этот код запускает интерактивную сессию, о чем вы можете судить по интеграции Git в Bash (`BISECTING`). Если в текущем коммите есть исследуемый дефект, вы помечаете его (листинг 12.10).

Листинг 12.10. Пометка коммита как плохого в сессии bisect

```
$ git bisect bad

~/Restaurant ((56a7092...)|BISECTING)
```

Если вы не укажете идентификатор коммита, Git предположит, что вы имели в виду текущий (здесь это `56a7092`).

В листинге 12.11 вы сообщаете ему об идентификаторе коммита, который вам точно известен как хороший. Это другая крайность диапазона коммитов, которые вы исследуете.

Листинг 12.11. Пометка коммита как хорошего в сессии bisect. Я немного отформатировал вывод, чтобы он поместился на странице

```
$ git bisect good 58fc950
Bisecting: 75 revisions left to test after this (roughly 6 steps)
[3035c14...] Use InMemoryRestaurantDatabase in a test

~/Restaurant ((3035c14...)|BISECTING)
```

Обратите внимание, что Git уже сообщает вам количество ожидаемых итераций. И вы можете видеть, что он проверил для вас новый коммит (`3035c14`). Это промежуточный коммит.

Теперь нужно проверить, есть ли в этом коммите дефект. Вы можете сделать это, запустив автотест, систему или любым другим способом, который вы определили для решения этого вопроса.

В моем случае промежуточный коммит не имеет дефекта, что я и сообщил Git (листинг 12.12).

Листинг 12.12. Пометка промежуточного коммита как хорошего в сессии bisect. Я немного отформатировал вывод, чтобы он поместился на странице

```
$ git bisect good
Bisecting: 37 revisions left to test after this (roughly 5 steps)
[aa69259...] Delete Either API

~/Restaurant ((aa69259...)|BISECTING)
```

Git снова оценивает, сколько еще шагов осталось, и проверяет новый коммит (`aa69259`) (листинг 12.13).

Листинг 12.13. Поиск коммита, ответственного за дефект, с помощью сессии Git bisect

```
$ git bisect bad
Bisecting: 18 revisions left to test after this (roughly 4 steps)
[75f3c56...] Delete redundant Test Data Builders

~/Restaurant ((75f3c56...)|BISECTING)
$ git bisect good
Bisecting: 9 revisions left to test after this (roughly 3 steps)
[8f93562...] Extract WillAcceptUpdate helper method

~/Restaurant ((8f93562...)|BISECTING)
$ git bisect good
Bisecting: 4 revisions left to test after this (roughly 2 steps)
[1c6fae1...] Extract ConfigureClock helper method

~/Restaurant ((1c6fae1...)|BISECTING)
$ git bisect good
```

```
Bisecting: 2 revisions left to test after this (roughly 1 step)
[8e1f1ce] Compact code

~/Restaurant ((8e1f1ce...)|BISECTING)
$ git bisect good
Bisecting: 0 revisions left to test after this (roughly 1 step)
[2563131] Extract CreateTokenValidationParameters method

~/Restaurant ((2563131...)|BISECTING)
$ git bisect bad
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[fa0caeb...] Move Configure method up

~/Restaurant ((fa0caeb...)|BISECTING)
$ git bisect good
2563131c2d06af8e48f1df2dccbf85e9fc8ddafc is the first bad commit
commit 2563131c2d06af8e48f1df2dccbf85e9fc8ddafc
Author: Mark Seemann <mark@example.com>
Date:  Wed Sep 16 07:15:12 2020 +0200

Extract CreateTokenValidationParameters method

Restaurant.RestApi/Startup.cs | 32 ++++++++++++++++++++-------------
1 file changed, 19 insertions(+), 13 deletions(-)

~/Restaurant ((fa0caeb...)|BISECTING)
```

Я повторил процесс для каждого шага, помечая коммит как хороший или плохой, в зависимости от того, прошел мой шаг проверки или нет (см. листинг 12.13).

Всего после восьми итераций Git нашел коммит, ответственный за дефект. Обратите внимание, что последний шаг сообщает, какой коммит был «первым плохим».

Увидев содержимое коммита, я сразу понял, в чем проблема, и смог легко ее исправить. Не буду здесь детально описывать ошибку и способ исправления. Чтобы узнать о процессе подробнее, можете посетить мой блог [101] или ознакомиться с репозиторием Git, который прилагается к книге.

Суть в том, что бисекция — это мощный метод поиска и изоляции источника ошибки. Вы можете использовать его с Git или без него.

12.4. ЗАКЛЮЧЕНИЕ

В поиске и устранении дефектов, как правило, помогает личный опыт. Однажды я работал в команде, где модульный тест закончился безуспешно на компьютере одного разработчика, но прошел успешно на ноутбуке другого. Точно такой же тест, тот же код, тот же коммит Git.

Мы могли бы просто пожать плечами и найти обходной путь, но все знали, что важно найти дефект. Два программиста работали вместе около получаса, чтобы свести проблему к минимальному рабочему примеру. По сути, все сводилось к сравнению строк.

На компьютере, где тест *не был пройден*, при сравнении строк будет считаться, что `aa` меньше, чем `bb`, а `bb` меньше, чем `cc`. Вроде все верно, не так ли?

Но на компьютере, где тест прошел *успешно*, `bb` все еще меньше, чем `cc`, но `aa` *больше, чем* `bb`. В чем же дело?

В этот момент я вмешался и спросил обоих разработчиков, какая у них «культура по умолчанию». В .NET «культура по умолчанию» — это окружающий контекст (ambient context) [25], который знает о правилах форматирования для конкретной культуры, о порядке сортировки и т. д.

Как и ожидалось, компьютер, который считал, что `aa` больше, чем `bb`, работал с датской «культурой по умолчанию», тогда как другой использовал английский язык США. В датском алфавите есть три дополнительные буквы (*Æ*, *Ø* и *Å*) после Z, но ранее буква Å записывалась как Aa, и, поскольку это написание все еще существует в именах собственных, комбинация aa считается равнозначной å. И, будучи последней буквой в алфавите, Å считается больше, чем B.

Для выявления проблемы мне понадобилось меньше минуты, так как в начале своей карьеры я часто сталкивался с подобными случаями. Это все еще зыбучие пески личного опыта — искусство программной инженерии.

Я бы никогда не смог определить проблему, если бы мои коллеги сначала не использовали методологию, подобную бисекции, чтобы свести проблему к простому симптому. Возможность создать минимальный рабочий пример — это суперсила для поиска и устранения дефектов в ПО.

Обратите внимание, что я *не упомянул* в этой главе отладку.

Слишком много разработчиков для устранения неполадок рассчитывают лишь на процесс отладки. Иногда я тоже использую отладчик, но все же сочетание научного метода, автоматизированного тестирования и бисекции кажется мне более эффективным. Изучите и используйте эти универсальные методы, ведь вы не сможете применять инструменты отладки в своей продакшен-среде.

13 РАЗДЕЛЕНИЕ ОТВЕТСТВЕННОСТИ

Представьте, что вы изменили схему БД вашего приложения, что привело к увеличению размера шрифта в электронных письмах, отправляемых системой.

Почему размер шрифта шаблона электронной почты зависит от схемы базы данных? Хороший вопрос. Так быть не должно.

Никогда не включайте бизнес-логику в свой пользовательский интерфейс. Не добавляйте код импорта и экспорта данных в свой защитный код. Этот принцип известен как разделение ответственности. Кент Бек выразился так:

> *«Те события, которые меняются с одинаковой скоростью, принадлежат друг другу, а те, которые изменяются с разной, должны быть отдельно друг от друга» [8].*

Основной посыл этой книги: код должен умещаться в голове. Как написано в подразделах 7.1.3 и 7.2.7, блоки кода должны быть небольшими и изолированными. Важно держать фрагменты отдельно друг от друга.

В главе 7 в основном была описана декомпозиция — почему и когда нужно разбивать большие блоки кода на более мелкие, но мало говорилось о том, *как* именно это делать.

Здесь я попытаюсь ответить на этот вопрос.

13.1. КОМПОЗИЦИЯ

Композиция и декомпозиция неразрывно связаны, ведь основная цель написания кода — создание рабочего ПО. Вы не можете произвольно делить что-то на части. Хотя декомпозиция важна (рис. 13.1), нужно уметь восстановить то, что вы декомпозировали.

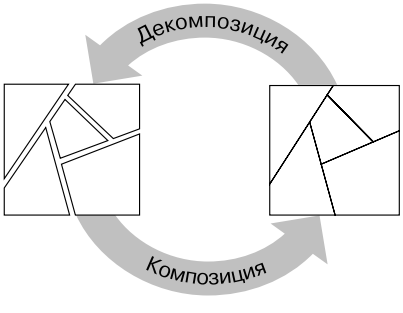


Рис. 13.1. Композиция и декомпозиция неразрывно связаны. Декомпозицию необходимо выполнять так, чтобы можно было вновь составить из частей рабочее ПО

Таким образом, модели композиции являются иллюстративными. Существует несколько способов композиции программных компонентов[1], и у всех есть как плюсы, так и минусы. И сразу же привнесу горькую истину — объектно-ориентированная композиция довольно проблемна.

[1] Я использую термин «компонент» в широком смысле. Он может означать объект, модуль, библиотеку, виджет или что-то еще. У некоторых языков программирования и платформ есть определенные представления об этом понятии, но обычно они несовместимы с представлениями другого языка. Как и модульный тест или mock, этот термин неоднозначен.

13.1.1. Вложенная композиция

Программное обеспечение взаимодействует с реальным миром. С его помощью можно рисовать на экране, сохранять данные в базах, отправлять электронные письма, публиковать сообщения в социальных сетях, управлять промышленными роботами и пр. Все это в контексте CQS (разделение команд и запросов) мы называем *побочными эффектами*.

Поскольку побочные эффекты — смысл существования ПО, кажется вполне естественным выполнять композицию вокруг них. Именно так большинство людей склонны подходить к ООП. Вы моделируете *действия*.

Объектно-ориентированная композиция, как правило, фокусируется на совместном составлении побочных эффектов. Паттерн проектирования «Компоновщик» [39] может быть образцом такого стиля композиции, но большинство шаблонов в «Паттернах проектирования» [39] сильно зависят от композиции побочных эффектов.

Как показано на рис. 13.2, этот стиль основан на вложении одних объектов в другие или одних побочных эффектов в другие. Но, так как наша цель — умещающийся в голове код, это может стать проблемой.

Чтобы показать, насколько это проблематично, я собираюсь сделать то, от чего до сих пор воздерживался: я покажу вам плохой код. Никогда не пишите такой код, как в листинге 13.1 или 13.3.

Листинг 13.1. Плохой код: контроллер, взаимодействующий с вложенной композицией. В листинге 13.6 приведен более подходящий пример (Restaurant/b3dd0fe/Restaurant.RestApi/ReservationsController.cs)

```
public IRestaurantManager Manager { get; }

public async Task<ActionResult> Post(ReservationDto dto)
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));
```

```
    Reservation? r = dto.Validate();
    if (r is null)
        return new BadRequestResult();

    var isAccepted =
        await Manager.Check(r).ConfigureAwait(false);
    if (!isAccepted)
        return new StatusCodeResult(
            StatusCodes.Status500InternalServerError);

  return new NoContentResult();
}
```

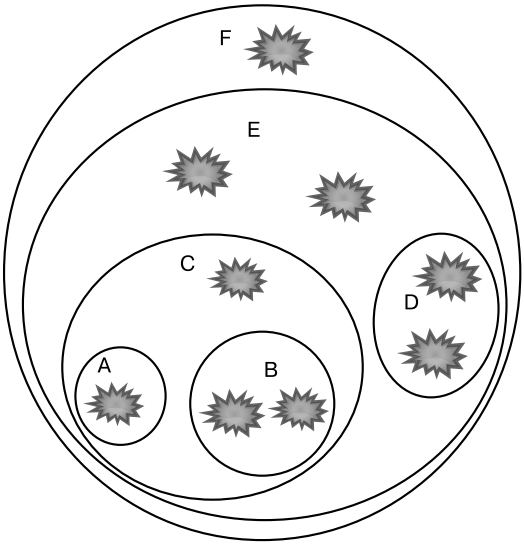


Рис. 13.2. Типичная композиция объектов (точнее, методов объектов) — вложенность. Чем больше модулей вы объединяете, тем сложнее становится композиция. Каждая звездочка на рисунке указывает на интересующий вас побочный эффект. Объект A инкапсулирует один побочный эффект, а B — два. Объект C объединяет A и B, но добавляет и четвертый побочный эффект. Итого уже четыре побочных эффекта, о которых нужно помнить, разбираясь с кодом. Ситуация легко выходит из-под контроля: объект E создает в общей сложности восемь побочных эффектов, а F — девять. Все это уже плохо умещается в вашей голове

Вы можете задаться вопросом, что же не так с этим кодом. Ведь его цикломатическая сложность всего 4, объектов — тоже, а строк 17. Проблема скрыта за одним из четырех объектов: `Manager` — внедренной зависимостью. Это интерфейс `IRestaurantManager` из кода листинга 13.2. Можете ли вы найти проблему?

Листинг 13.2. Интерфейс IRestaurantManager, использованный в листинге 13.1 и реализованный в листинге 13.3 (Restaurant/b3dd0fe/Restaurant.RestApi/IRestaurantManager.cs)

```
public interface IRestaurantManager
{
  Task<bool> Check(Reservation reservation);
}
```

Попробуйте скрыть имя метода за несколькими `x`. Если вы это сделаете, у вас останется `Task<bool> Xxx(Reservation reservation)`, который выглядит как асинхронный предикат. Это должен быть метод, который проверяет, является ли информация о бронировании `true` или `false`. Но если вы проанализируете код листинга 13.1 с этой точки зрения, метод `Post` использует только логическое значение, чтобы решить, какой код состояния HTTP нужно вернуть.

Программист забыл сохранить бронирование в базе данных?

Скорее всего, нет. Проанализируйте реализацию интерфейса `IRestaurantManager` из листинга 13.3. Интерфейс выполняет небольшую проверку, а затем вызывает `Manager.TrySave`.

Листинг 13.3. Плохой код: реализация интерфейса IRestaurantManager выглядит, будто в ней есть побочный эффект (Restaurant/b3dd0fe/Restaurant.RestApi/RestaurantManager.cs)

```
public async Task<bool> Check(Reservation reservation)
{
    if (reservation is null)
        throw new ArgumentNullException(nameof(reservation));

    if (reservation.At < DateTime.Now)
        return false;
    if (reservation.At.TimeOfDay < OpensAt)
        return false;
```

```
    if (LastSeating < reservation.At.TimeOfDay)
        return false;

    return await Manager.TrySave(reservation).ConfigureAwait(false);
}
```

Если вы продолжите «тянуть» этот спагетти-код, то в конечном итоге обнаружите, что `Manager.TrySave` сохраняет резервирование в базе данных *и* возвращает логическое значение. Основываясь на том, что вы уже узнали, можете сказать, что здесь не так?

Здесь нарушен принцип разделения команд и запросов. Хотя метод выглядит как запрос, у него есть побочный эффект. Почему это проблема?

Вспомните определение Роберта Мартина:

> *«Абстракция — это устранение неважного и усиление существенного» [60].*

Скрывая побочный эффект в запросе, я *избавился* от чего-то существенного. Другими словами, в листинге 13.1 происходит больше, чем кажется. Цикломатическая сложность может быть всего 4, но есть скрытое пятое действие, о котором нужно знать.

Конечно, пять фрагментов все еще умещаются в голове, но такое единственное скрытое взаимодействие — это дополнительные 14 % по сравнению с бюджетом в семь фрагментов. Не нужно много скрытых побочных эффектов, для того чтобы код перестал быть понятным.

13.1.2. Последовательная композиция

Хотя вложенная композиция проблематична, это не единственный способ компоновки. Вы можете скомпоновать поведение, объединив его в цепочку, как на рис. 13.3.

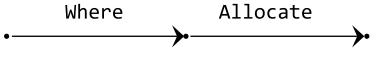


Рис. 13.3. Последовательная композиция двух функций. Выход из Where становится входом для Allocate

В контексте разделения команд и запросов команды вызывают проблемы, тогда как запросы — почти никаких. Они возвращают данные, которые вы можете использовать в качестве входных для других запросов.

Вся кодовая база примеров ресторана создана с учетом этого принципа. Рассмотрим метод `WillAccept` из листинга 8.13. После всех Guard Clauses [7] сначала создается новый экземпляр класса `Seating`. Вы можете рассматривать конструктор как запрос, при условии что он не имеет побочных эффектов[1].

Следующая строка кода фильтрует существующие данные о резервировании, используя в качестве предиката метод `Overlaps` из листинга 13.4. Встроенный метод `Where` — это запрос, как и метод `Overlaps`.

Коллекция `relevantReservations` — это выходные данные одного запроса. Но она становится входными данными для запроса `Allocate` из листинга 13.5.

Листинг 13.4. Метод Overlaps — это запрос, так как не имеет побочных эффектов и возвращает данные (Restaurant/e9a5587/Restaurant.RestApi/Seating.cs)

```
internal bool Overlaps(Reservation other)
{
    var otherSeating = new Seating(SeatingDuration, other);
    return Start < otherSeating.End && otherSeating.Start < End;
}
```

Листинг 13.5. Метод Allocate — другой запрос (Restaurant/e9a5587/Restaurant.RestApi/MaitreD.cs)

```
private IEnumerable<Table> Allocate(
    IEnumerable<Reservation> reservations)
{
    List<Table> availableTables = Tables.ToList();
    foreach (var r in reservations)
    {
```

[1] У конструктора не должно быть побочных эффектов!

```
            var table = availableTables.Find(t => t.Fits(r.Quantity));
            if (table is { })
            {
                availableTables.Remove(table);
                if (table.IsCommunal)
                    availableTables.Add(table.Reserve(r.Quantity));
            }
        }

        return availableTables;
    }
```

Наконец, метод `WillAccept` возвращает информацию о том, есть ли какая-либо таблица `Any` среди `availableTables`, где `Fits` соответствует `candidate.Quantity`. Метод `Any` — это еще один встроенный запрос, а `Fits` — это предикат.

По сравнению с рис. 13.3 можно сказать, что конструктор `Seating`, `seating.Overlaps`, `Allocate` и `Fits` составлены последовательно.

Ни один из этих методов не имеет побочных эффектов, значит, как только `WillAccept` вернет свое логическое значение, вы можете забыть о том, как именно он достиг этого результата. Он и правда устраняет ненужное и усиливает существенное.

13.1.3. Ссылочная прозрачность

Остается еще одна проблема, которую принцип разделения команд и запросов не может решить: предсказуемость. Хотя у запроса нет побочных эффектов, которые нужно контролировать, вас все равно может удивить, что вы будете получать новое возвращаемое значение каждый раз, когда его вызываете, даже с теми же входными данными.

Все не так плохо, как с побочными эффектами, но все равно будет нелегко. Что произойдет, если мы установим дополнительное правило для принципа разделения команд и запросов, согласно которому запросы должны быть детерминированными?

Это означает, что запрос не может полагаться на генераторы случайных чисел, создание GUID, время суток, день месяца или любые

другие данные из среды, а также содержимое файлов и БД. Звучит ограничительно, так в чем же польза?

Детерминированный метод *без побочных эффектов ссылочно прозрачен*. Это также известно как *чистая функция*. У таких функций есть несколько полезных качеств.

Одно из таких качеств — это то, что чистые функции легко компонуются. Если выходные данные одной функции подходят для ввода другой, их можно скомпоновать последовательно. Всегда. На то есть математические причины[1], но достаточно сказать, что композиция встроена в код, из которого сделаны чистые функции.

Другое качество — это то, что вы можете заменить вызов чистой функции ее результатом: вызов функции *равен* результату. Единственная разница между результатом и вызовом функции — это время для его получения.

Подумайте об этом с точки зрения определения *абстракции* Роберта Мартина. Как только чистая функция возвращается, результат — это все, о чем вам нужно беспокоиться. То, как функция пришла к нему, — деталь реализации. Ссылочно-прозрачные функции устраняют неважное и усиливают существенное. Как показано на рис. 13.4, они сводят произвольную сложность к одному результату — одному простому для понимания фрагменту.

Но если вы хотите узнать, как работает функция, вы можете рассмотреть ее реализацию с точки зрения фрактальной архитектуры. Это может быть метод `WillAccept` из листинга 8.13. На самом деле это не просто запрос — это чистая функция. Анализируя ее исходный код, вы видите его увеличенным, а окружающий контекст не имеет значения. Он работает только со своими входными аргументами и неизменяемыми полями класса.

---

[1] Одна точка зрения предлагается теорией категорий, на которую опираются такие языки функционального программирования, как Haskell. Отличное введение для программистов можно найти в публикациях Бартоша Милевски Category Theory for Programmers [68].

Когда вы снова уменьшаете масштаб, вся функция сводится к своему результату. Это единственное, за чем вы должны следить.

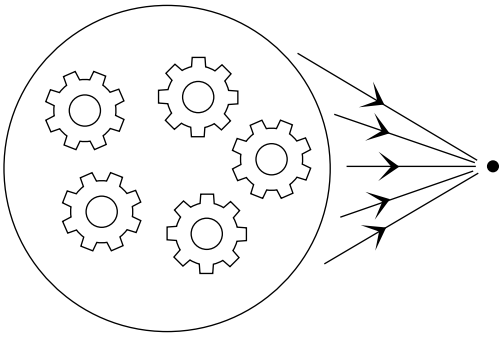


Рис. 13.4. Чистая функция *(слева)*, приводящая к результату *(справа)*. Независимо от сложности ссылочно-прозрачный вызов функции может быть заменен ее выходными данными. Таким образом, однажды узнав, какой вывод, это единственное, что вам нужно будет отслеживать, считывая и интерпретируя вызывающий код

Что насчет всего недетерминированного поведения и побочных эффектов?

Выдвиньте все на край системы: ваш метод `Main`, ваши контроллеры, обработчики сообщений и т. д. Рассмотрите листинг 13.6 как лучшую альтернативу листингу 13.1.

Для ясности: сам метод `Post` не прозрачен с точки зрения ссылок. Он создает новый GUID (недетерминированность), запрашивает БД (недетерминированность), получает текущие дату и время (недетерминированность) и условно сохраняет резервирование в базе данных (побочный эффект).

Листинг 13.6. Последовательная композиция метода Post (в сравнении с листингом 13.1) (Restaurant/e9a5587/Restaurant.RestApi/ReservationsController.cs)

```
[HttpPost]
public async Task<ActionResult> Post(ReservationDto dto)
```

```
{
    if (dto is null)
        throw new ArgumentNullException(nameof(dto));

    var id = dto.ParseId() ?? Guid.NewGuid();
    Reservation? r = dto.Validate(id);
    if (r is null)
        return new BadRequestResult();

    var reservations = await Repository
        .ReadReservations(r.At)
        .ConfigureAwait(false);
    if (!MaitreD.WillAccept(DateTime.Now, reservations, r))
        return NoTables500InternalServerError();

    await Repository.Create(r).ConfigureAwait(false);

    return Reservation201Created(r);
}
```

Собрав все данные, метод вызывает чистую функцию `WillAccept`. Только если `WillAccept` возвращает значение `true`, `Post` допускает возникновение побочного эффекта.

Держите недетерминированные запросы и поведение с побочными эффектами близко к краю системы и создавайте сложную логику в виде чистых функций. Такой стиль программирования известен как *функциональное ядро*, *императивная оболочка* [11], так как программирование с использованием в основном чистых функций — это область функционального программирования.

Чтобы с кодом было проще разобраться, рекомендую вам изучить функциональное программирование[1].

[1] Я рекомендую вам начать с изучения правильного языка функционального программирования Haskell. Но вы можете выбрать язык, который соответствует вашим предпочтениям. Большинство ваших знаний из функционального программирования вы можете использовать для улучшения ваших объектно-ориентированных кодовых баз. Вся кодовая база примеров написана в стиле функционального ядра в императивной оболочке, хотя она написана на якобы объектно-ориентированном языке C#.

13.2. СКВОЗНАЯ ФУНКЦИОНАЛЬНОСТЬ

Есть ряд проблем, которые пересекаются с разрозненными функциями. Неудивительно, что их называют *сквозными*. К ним относятся [25]:

- логирование;
- мониторинг производительности;
- аудиторская проверка;
- измерение;
- инструментирование;
- кэширование;
- отказоустойчивость;
- безопасность.

Все это может вам не пригодиться, но если что-то из этого вдруг понадобится использовать, то конкретная проблема распространится на многие функции.

Например, если вы обнаружите, что вам нужно добавить шаблон Circuit Breaker (прерыватель цепи) [73] к вызовам вашего веб-сервиса, придется делать это везде, где вы его вызываете. Или, если вам нужно кэшировать запросы к базе данных, вы должны делать это последовательно.

По моему опыту, сквозные задачи имеют одну общую черту: их лучше всего реализовать с помощью шаблона проектирования Decorator [39]. Рассмотрим следующий пример.

13.2.1. Логирование

Большинство элементов из списка выше — варианты ведения логов, в том смысле, что они включают запись данных в какой-либо лог. Мониторинг производительности записывает измерения производительности, аудиторская проверка — данные аудита, измерение — данные об использовании того, что можно измерить, а инструментирование — отладочную информацию.

Скорее всего, вам понадобится реализовать лишь часть из перечисленных выше сквозных задач. Нужны ли они вам, зависит от требований системы.

Но вам в любом случае *нужно* добавить минимум логирования в свою систему. Во время работы ваше ПО столкнется с непредвиденными обстоятельствами. Оно может аварийно закрываться или обнажать дефекты. Чтобы решить проблемы, в первую очередь их нужно понять — логи дадут вам бесценную информацию о работе системы.

Как минимум важно убедиться, что все необработанные исключения регистрируются. Для этого вам может не потребоваться предпринимать явных действий. Например, ASP.NET автоматически регистрирует необработанные исключения как в Windows, так и в Microsoft Azure.

Поглядывайте в логи. Идеальное количество необработанных исключений равно нулю. Если вы видите исключение в журнале, рассматривайте его как дефект. Подробнее об этом — в разделе 12.2.

Одни дефекты проявляются в виде аварийного закрытия приложения во время выполнения, другие же — как неправильное поведение. Система продолжает работать, но ведет себя некорректно. В разделе 12.2 было приведено несколько примеров. Система разрешила избыточное резервирование, а адрес электронной почты и имя были перепутаны местами. Чтобы найти причину сбоя, вам понадобится больше логов, чем просто лог необработанных исключений.

13.2.2. Паттерн проектирования Decorator («Декоратор»)

Декоратор иногда называют матрешкой в честь традиционных русских кукол, которые можно друг в друга вкладывать (рис. 13.5).

Как и матрешки, полиморфные объекты могут быть вложены друг в друга. Это отличный способ добавить несвязанную функциональность к существующей реализации. Давайте рассмотрим пример того, как добавить ведение журнала в интерфейс доступа к базе данных (листинг 13.7).

Кодовая база уже содержит класс, реализующий интерфейс. Он называется `SqlReservationsRepository` и выполняет чтение и запись в основную

БД SQL Server. Чтобы регистрировать действия этого класса, нужно разделить эти задачи. Не редактируйте `SqlReservationsRepository` только для добавления ведения лога. Добавьте декоратор. В листинге 13.8 приведен пример объявления класса и конструктора. Обратите внимание, что, хотя он реализует интерфейс `IReservationsRepository`, он служит и оболочкой для другого объекта `IReservationsRepository`.

Рис. 13.5. Матрешки, что могут вкладываться одна в другую, часто используются в качестве метафоры декоратора

Листинг 13.7. Еще одна версия интерфейса IReservationsRepository, на этот раз с поддержкой нескольких пользователей. Для других вариантов вы можете в качестве примера использовать код из листинга 10.11 или 8.3 (Restaurant/3bfaa4b/Restaurant.RestApi/IReservationsRepository.cs)

```
public interface IReservationsRepository
{
    Task Create(int restaurantId, Reservation reservation);

    Task<IReadOnlyCollection<Reservation>> ReadReservations(
        int restaurantId, DateTime min, DateTime max);

    Task<Reservation?> ReadReservation(Guid id);

    Task Update(Reservation reservation);

    Task Delete(Guid id);
}
```

Поскольку код реализует интерфейс, он должен реализовывать все методы. Это всегда возможно, так как он может просто вызвать один и тот же метод на `Inner`. Но каждый метод позволяет декоратору перехватить вызов метода. В листинге 13.9 приведен пример логирования метода `ReadReservation`.

Листинг 13.8. Объявление класса и конструктор для декоратора LoggingReservationsRepository (Restaurant/3bfaa4b/Restaurant.RestApi/LoggingReservationsRepository.cs)

```
public sealed class LoggingReservationsRepository :
IReservationsRepository
{
    public LoggingReservationsRepository(
        ILogger<LoggingReservationsRepository> logger,
        IReservationsRepository inner)
    {
        Logger = logger;
        Inner = inner;
    }

    public ILogger<LoggingReservationsRepository> Logger { get; }
    public IReservationsRepository Inner { get; }
```

Листинг 13.9. Декорированный метод ReadReservation (Restaurant/3bfaa4b/Restaurant.RestApi/LoggingReservationsRepository.cs)

```
public async Task<Reservation?> ReadReservation(Guid id)
{
    var output = await Inner.ReadReservation(id).
    ConfigureAwait(false);
    Logger.LogInformation(
        "{method}(id: {id}) => {output}",
        nameof(ReadReservation),
        id,
        JsonSerializer.Serialize(output?.ToDto()));
    return output;
}
```

Сначала метод вызывает `ReadReservation` для внутренней реализации, чтобы получить выходные данные. Перед возвратом вывода он регистрирует, что метод был вызван, используя внедренный `Logger`.

В листинге 13.10 приведен пример типичной записи лога, созданной этим кодом.

Другие методы `LoggingReservationsRepository` работают так же: вызывают внутреннюю реализацию, регистрируют результат и возвращают.

Вам нужно подготовить встроенный контейнер внедрения зависимостей ASP.NET, чтобы использовать декоратор вокруг фактической реализации (листинг 13.11). Некоторые контейнеры внедрения зависимостей изначально знают о шаблоне Decorator, а встроенный — нет. К счастью, чтобы обойти это ограничение, вы можете зарегистрировать службы с помощью лямбда-выражения.

Листинг 13.10. Пример записи в логе, созданной листингом 13.9. Фактическая запись представляет собой одну широкую строку. Я отредактировал ее для удобства чтения, добавив разрывы строк и небольшие отступы

```
2020-11-12 16:48:29.441 +00:00 [Information]
Ploeh.Samples.Restaurants.RestApi.LoggingReservationsRepository:
ReadReservation(id: 55a1957b-f85e-41a0-9f1f-6b052f8dcafd) =>
{
  "Id":"55a1957bf85e41a09f1f6b052f8dcafd",
  "At":"2021-05-14T20:30:00.0000000",
  "Email":"elboughs@example.org",
  "Name":"Elle Burroughs",
  "Quantity":5
}
```

Листинг 13.11. Настройка декоратора с помощью платформы ASP.NET (Restaurant/3bfaa4b/Restaurant.RestApi/Startup.cs)

```
var connStr = Configuration.GetConnectionString("Restaurant");
services.AddSingleton<IReservationsRepository>(sp =>
{
    var logger =
        sp.GetService<ILogger<LoggingReservationsRepository>>();
    return new LoggingReservationsRepository(
        logger,
        new SqlReservationsRepository(connStr));
});
```

В примере системы бронирования ресторанов есть другие зависимости, помимо `IReservationsRepository`. Например, она еще отправляет электронные письма с помощью интерфейса `IPostOffice`. Для регистрации этих взаимодействий используется декоратор `LoggingPostOffice`, эквивалентный `LoggingReservationsRepository`.

Используя декораторы, можно решить большинство сквозных проблем. Для кэширования вы можете реализовать декоратор, который сначала пытается читать из кэша. Только если значение не находится в кэше, он считывает базовое хранилище данных, и в этом случае обновляет кэш перед возвратом. Это называется *кэшем сквозного чтения*.

Что касается отказоустойчивости, в моей прошлой книге [25] есть пример шаблона Circuit Breaker [73]. С помощью декоратора можно решить и проблемы безопасности, но у большинства фреймворков встроенные функции безопасности, и лучше использовать их. См. пример в подразделе 15.2.5.

13.2.3. Что регистрировать

Однажды я работал с командой, которая точно определила необходимое количество логов. Мы разрабатывали и поддерживали набор REST API. Каждый API регистрировал в лог детали[1] каждого HTTP-запроса и возвращенного им HTTP-ответа. Он регистрировал и все взаимодействия с БД, включая входные аргументы и весь набор результатов, возвращаемый ею.

Не было ни одного дефекта, который мы не смогли бы отследить и понять. Вот в чем польза верного количества логов.

Большинство организаций-разработчиков регистрируют слишком много информации. Когда дело доходит до инструментов, я часто вижу примеры «перелогирования». Когда вы ведете логи для поддержки устранения будущих дефектов, вы не можете предсказать,

[1] За исключением конфиденциальной информации, такой как веб-токены JSON, которые мы отредактировали.

что вам понадобится, поэтому решаете регистрировать максимально много данных. Это и служит причиной «перелогирования».

Лучше всего регистрировать только то, что вам нужно. Не слишком мало, не слишком много, а ровно столько, сколько необходимо. Очевидно, мы должны назвать это *Goldilogs*.

Откуда вы знаете, что регистрировать? Откуда знаете, что записали все, что вам нужно, если вы не знаете, что вам может пригодиться?

Главное — повторяемость. Точно так же, как вы должны иметь возможность воспроизводить сборки и повторять развертывания, у вас должна быть возможность воспроизводить выполнение.

Если вы можете воспроизвести то, что произошло, когда проблема проявилась, вы можете устранить ее. Вам нужно зарегистрировать достаточно данных, чтобы вы могли повторить выполнение. Как вы идентифицируете эти данные?

Рассмотрим код в листинге 13.12. Вы бы зарегистрировали это?

Листинг 13.12. Будете ли вы регистрировать это утверждение?

```
int z = x + y;
```

Разумно регистрировать `x` и `y`, особенно если это значения времени выполнения (например, введенные пользователем или в результате вызова веб-сервиса и т. д.). Вы можете сделать что-то вроде кода из листинга 13.13.

Листинг 13.13. Логирование входных значений может быть разумным

```
Log.Debug($"Adding {x} and {y}.");
int z = x + y;
```

Но вы бы когда-нибудь зарегистрировали результат как в листинге 13.14?

Листинг 13.14. Есть ли смысл регистрировать результат сложения?

```
Log.Debug($"Adding {x} and {y}.");
int z = x + y;
Log.Debug($"Result of addition: {z}");
```

Нет причин контролировать и регистрировать результаты расчета. Сложение — это чистая функция, оно *детерминировано*. Если вы знаете входные данные, вы всегда можете повторить расчет, чтобы получить результат. Два плюс два — всегда четыре.

Чем больше ваш код состоит из чистых функций, тем меньше вам нужно регистрировать [103]. Это одна из многих причин, по которым ссылочная прозрачность полезна и желательна и почему нужно отдавать предпочтение стилю архитектуры функционального ядра в императивной оболочке.

> Регистрируйте все неявные события, но не более.

Записывайте все, что не можете воспроизвести: весь недетерминированный код (например, получение текущей даты, времени суток, генерация случайного числа, чтение из файла или базы данных и т. д.) и все, что имеет побочные эффекты. Все остальное регистрировать не нужно.

Конечно, если ваша кодовая база не отделяет чистые функции от неявных событий, вам придется регистрировать все данные.

13.3. ЗАКЛЮЧЕНИЕ

Разделяйте несвязанные проблемы. Изменения в пользовательском интерфейсе не должны затрагивать кодовую базу данных, и наоборот.

Разделение задач подразумевает, что вы должны разделять, то есть декомпозировать, разные фрагменты вашей кодовой базы. Декомпозиция полезна, но только если вы можете перекомпоновать разрозненные части кодовой базы.

Даже если вы сможете выполнить объектно-ориентированную декомпозицию, вам придется приложить много усилий, чтобы заставить ее работать. Большинство разработчиков не знают, как это сделать, поэтому обычно компонуют объекты, вкладывая их друг в друга.

Делая так, вы заметаете важное поведение под ковер, что затрудняет понимание кода.

Последовательная композиция, когда чистые функции возвращают данные, которые можно использовать в качестве входных для других чистых функций, дает более разумную альтернативу.

Хотя я совсем не ожидаю, что организации избавятся от своих так называемых объектно-ориентированных кодовых баз для Haskell, я все же настоятельно рекомендую перейти к *функциональному ядру, императивной оболочке*.

Это упрощает изолирование фрагментов базы кода, реализующих неявные события. Как правило, это и те фрагменты, где вам нужно применять сквозную функциональность, что лучше всего делать с помощью декораторов.

14 ОРГАНИЗАЦИЯ РАБОЧЕГО ПРОЦЕССА

Я сотрудничал со многими организациями по разработке ПО. Одни следуют одному процессу, другие — другому. В некоторых то, что делается, может отличаться от того, что говорят.

Некоторые команды говорят, что проводят стендапы ежедневно, но на самом деле делают это через день, или когда есть желание и необходимость.

Я работал в команде, где каждое утро мы устраивали стендапы. Но одному коллеге всегда удавалось сосредоточить встречу вокруг своего стола, за которым он оставался сидеть. Он был из тех людей, которые полностью игнорировали формат «что я сделал вчера, что я собираюсь делать сегодня» и формат каких-либо блокировок. В итоге всю планировку мы просто болтали, стоя рядом с его столом, пока не начинали болеть ноги.

Я работал в команде, которая придерживалась методологии канбан. Все было хорошо, за исключением того, что много времени тратилось на «тушение пожаров». Эти приятные рабочие моменты давали плохое представление о работе, которая на самом деле выполнялась.

Но одна из лучших команд, в которой я был, почти никогда не придерживалась какого-то определенного стиля работы. Все это было

неважным, так как они реализовали непрерывное развертывание. Эта команда предоставляла функции быстрее, чем заинтересованные стороны могли их принять. Вместо бесконечных вопросов о том, что готово, члены команды иногда спрашивали заинтересованных лиц, было ли у них время полюбоваться функциями, о которых они просили. Чаще всего на анализ новых функций у них не было времени.

Я не буду рассказывать вам, как организовать процесс работы. Независимо от того, работаете ли вы согласно Scrum, XP [5], PRINCE2 или «ежедневному хаосу», я надеюсь, что эта книга будет полезна вам. Я не буду настаивать на каком-либо конкретном процессе разработки ПО, так как пришел к выводу, что свободный ритм или структура дня могут быть полезными. Это относится к личному опыту и к командному.

14.1. ИНДИВИДУАЛЬНЫЙ ПРОЦЕСС РАБОТЫ

Каждый день разный, но я считаю, что лучше определить некоторую свободную структуру своих действий. Никакая ежедневная деятельность не должна быть обязательной — это вызовет лишь стресс из-за хотя бы одного упущенного дня. Структура же поможет вам завершить хотя бы часть запланированных дел.

Хотя моя жена, скорее всего, скажет вам, что я один из самых дисциплинированных людей, которых она знает, я тоже склонен откладывать дела на потом. Наличие ежедневного ритма помогает мне свести к минимуму бесполезную трату времени.

14.1.1. Тайм-боксинг

Работайте через определенные промежутки времени. Например, каждые 25 минут делайте пятиминутный перерыв. Вы, наверное, знакомы с методом помидора, но это не то. Он более сложный [18], а я считаю дополнительные действия неуместными.

У работы каждые 25 минут есть ряд преимуществ. Одни из них более очевидны, другие — менее.

Двадцать пять минут непрерывной работы могут помочь вам смотреть на объемную задачу как на более выполнимую. Даже если рабочий процесс кажется пугающим или непривлекательным, проще сказать себе, что вы можете смотреть на него хотя бы в течение 25 минут. Мой опыт показывает, что самое сложное — это начать.

Обязательно засеките время с помощью таймера (рис. 14.1). Я использую программу, отображающую оставшееся количество минут в системном трее[1] моего экрана. Преимущество видимого обратного отсчета в том, что автоматически отпадает необходимость проверять соцсети, электронную почту и т. д. Всякий раз, когда у меня возникает такое желание, я смотрю на обратный отсчет и думаю: «Хорошо, у меня осталось еще 16 минут. Я сначала доделаю задачу, а потом у меня будет перерыв».

Рис. 14.1. Вы можете спутать работу каждые 25 минут с методом помидора. На рисунке изображен кухонный таймер-помидор, от которого эта техника и получила свое название

Перерывы дают менее очевидное преимущество. Когда вы делаете перерыв, делайте его правильно: встаньте, пройдитесь, выйдите из комнаты или хотя бы отойдите от компьютера. Удивительно, как часто смена обстановки позволяет нам сменить свой взгляд на ситуацию.

[1] В Windows системный трей обычно находится в правом нижнем углу экрана. Его еще называют областью уведомлений.

Даже если вы не чувствуете, что застряли, перерыв поможет вам осознать, что вы потратили впустую последние 15 минут. Звучит неприятно, но я лучше потрачу 15 минут, чем три часа.

У меня часто бывало так, что необходимость остановиться и отойти от компьютера становилась почти мучительной. Но я обнаружил: это иногда помогает мне понять, что то, что я делал, могло никогда не заработать, потому что некоторые проблемы проявили бы себя позже.

Если бы я не сделал перерыв, то потратил бы часы работы впустую.

Программистам больше нравится работать непрерывно, но нет никакой гарантии того, что этот метод работы лучше. Вы можете написать много строк кода, но не факт, что они будут полезны.

Самый любопытный аспект в следующем: если то, что вы делаете, действительно полезно, пятиминутный перерыв будет неважен. Я часто осознавал, что даже после пары минут отсутствия за компьютером я могу сразу же вернуться к работе, если чувствую, что двигаюсь в верном направлении.

14.1.2. Делайте перерывы

Однажды я разработал какую-то часть ПО с открытым исходным кодом, которое стало настолько популярным, что пользователи начали предлагать разные функции и возможности, которые я изначально не планировал. Первая версия работала адекватно, но я понимал, что придется переписать бóльшую часть кода, чтобы он был более гибким.

Разработка нового дизайна предполагала много размышлений. К счастью, тогда я работал в организации, дорога до которой занимала около получаса, и я сделал бóльшую часть этой работы, просто катаясь на велосипеде[1] туда и обратно.

[1] Копенгаген — город велосипедистов, и я при любой возможности всегда езжу на велосипеде. Это быстрее и полезнее. Но всегда есть опасность погрузиться в свои мысли.

Отходить от компьютера весьма полезно. Я регулярно занимаюсь спортом, и многие идеи возникают во время бега, принятия душа или мытья посуды. Я не припомню, чтобы такое было, сиди я постоянно перед компьютером.

Думаю, так происходит потому, что моя Система 1 [51] (или какой-то другой подсознательный процесс) продолжает работать над проблемой, даже когда я не осознаю ее. Но это действует, только если я уже провел некоторое время перед компьютером, работая над проблемой. Вы не можете просто лежать на диване и ждать озарения. Старайтесь чередовать разные виды деятельности.

Если вы работаете в офисе, вам может быть сложно отправиться на прогулку. Но я думаю, что это может быть более продуктивно, чем сидеть весь день перед компьютером.

По возможности делайте перерывы в работе за компьютером. Займитесь чем-нибудь другим в течение 20–30 минут. Попробуйте совместить это с физической активностью. Это не обязательно должны быть тяжелые физические упражнения. Попробуйте просто прогуляться. Например, если у вас поблизости есть продуктовый магазин, наведайтесь туда. Я делаю это через день. Так я могу с пользой провести свой обеденный перерыв — шопинг становится эффективным, так как в это время всегда мало людей.

Помните: интеллектуальная работа отличается от физической. Вы не можете измерить продуктивность временем, проведенным за компьютером. На самом деле чем дольше вы работаете, тем ниже становится ваша продуктивность, поскольку вы будете делать ошибки, на исправление которых потом придется тратить время. Никогда не работайте без перерывов.

14.1.3. Используйте время разумно

Я не хочу превращать эту книгу в лекцию о личной продуктивности и самоорганизованности. Таких написали уже много. Но я советую вам использовать свое время осознанно. И дам для этого несколько рабочих техник.

Книга «Программист-прагматик» предлагает каждый год изучать новый язык программирования [50]. Я не уверен, что согласен с этим правилом. Знать более одного языка — хорошо, но изучать каждый год по одному — это уже перебор. Есть и другие навыки, которые нужно прокачивать: разработка через тестирование, алгоритмы, специальные библиотеки или фреймворки, паттерны проектирования, тестирование на основе свойств и т. п.

Я не пытаюсь каждый год изучать новый язык программирования, но стараюсь расширять свои знания. Если у меня нет встреч, я начинаю свой день с двух 25-минутных тайм-боксов, посвященных самообразованию. В эти дни я обычно читаю специальную литературу и делаю упражнения. На заре своей карьеры каждое утро я начинал с ответов на вопросы в Usenet[1], а позже — в Stack Overflow. Многому можно научиться, обучая кого-то другого. А еще я считал, что код ката — это упражнение по программированию, которое помогает специалистам оттачивать свои навыки через практику и повторение.

Еще один совет по продуктивности: ограничьте количество созвонов и совещаний, в которых участвуете. Однажды я работал в компании, которая постоянно проводила разные встречи. Какое-то время я занимал важную должность, поэтому получал много приглашений.

Я заметил, что многие из встреч на самом деле были просто обычными запросами на информацию. Заинтересованные стороны могли слышать, что я был на совещании без них, и просили узнать, что обсуждалось. Это понятно, но неэффективно, поэтому я просто начал записывать.

Когда меня звали на встречу, я запрашивал повестку дня. Часто этого было достаточно, чтобы отменить встречу. В других случаях, как только я видел повестку дня, я отправлял им то, что уже записал. Тогда сразу все получат необходимую им информацию, вместо того чтобы часами или днями ждать встречи.

---

[1] Да, это было давно!

14.1.4. Метод слепой печати

В 2013 году возник конфликт между профсоюзом учителей Дании и их государственными работодателями, в результате чего школы закрылись на неопределенный срок. Конфликт длился 25 дней, но, когда он начался, никто не знал, сколько он продлится.

Моей дочери тогда было десять лет. Я не хотел, чтобы она бездельничала, поэтому составил ей учебный план. Одним из навыков, который ей нужно было развивать, был метод слепой печати. Посвящать этому она должна была один час в день. Когда конфликт закончился, она уже легко печатала вслепую.

Когда в 2020 году из-за эпидемии COVID-19 были закрыты школы, мой тринадцатилетний сын получил такое же задание. Теперь он тоже легко печатает вслепую. Как итог: обучение моих детей слепому методу печати заняло несколько недель по часу в день.

Я работал с программистами, которые не умеют печатать вслепую, и заметил, насколько это неэффективно. Нет, дело не в скорости. В конце концов, набор текста не самый важный навык в разработке ПО. Вы тратите больше времени на чтение кода, чем на его ввод, поэтому производительность тесно связана с удобочитаемостью кода. Но не скорость печати влияет на *эффективность набора текста двумя пальцами*. Проблема в том, что, находясь в постоянных поисках следующей клавиши, вы не замечаете происходящего на экране.

В современных IDE есть много функций, которые сообщают об ошибках, но я начал печатать вслепую, когда всего этого еще не было. Тем не менее я не особенно аккуратен, и довольно часто мне приходится что-то удалять.

И хоть я склонен ошибаться в написании текста, при наборе кода я делаю меньше ошибок. Это связано с тем, что завершение операторов и другие функции IDE «выполняют набор» вместо меня, автоматически.

Я знаю программистов, которые настолько заняты поиском следующей клавиши, что пропускают все подсказки и функции, которые

предлагает IDE. Хуже того: если они опечатаются, то не увидят ошибку, пока не попытаются скомпилировать или запустить код. И когда они наконец смотрят на экран, то не понимают, что именно не так.

Работая в паре с такими людьми, я смотрю на опечатку десятки секунд, поэтому мне становится болезненно понятно, что не так, тогда как для человека это новый контекст, который еще нужно разобрать.

Обязательно научитесь печатать вслепую. IDE — это аббревиатура от Integrated Development Environment, но для современных инструментов термин «интерактивная среда разработки» будет более описателен. А если вы не смотрите на экран, интерактивности будет довольно мало.

14.2. РАБОЧИЙ ПРОЦЕСС В КОМАНДЕ

Во время работы в группе вам придется согласовывать свой индивидуальный ритм с командным. И, скорее всего, у команды будут какие-то повторяющиеся действия: ежедневные стендапы, ретроспективы спринтов каждые две недели или обед в определенное время.

Как я уже говорил, я не буду навязывать какой-либо конкретный процесс, но есть моменты, о которых нужно подумать заранее. Можно даже отметить их в виде чек-листа.

14.2.1. Регулярное обновление зависимостей

Кодовые базы имеют зависимости. Когда вы читаете их, то используете SDK для конкретного типа БД. Когда пишете модульный тест, вы используете фреймворк модульного тестирования. Если вы хотите аутентифицировать пользователей с помощью веб-токена JSON, то используете для этого библиотеку.

Такие зависимости обычно входят в пакеты, доставляемые через диспетчер. В .NET есть NuGet, в JavaScript — NPM, в Ruby — RubyGems

и т. д. Такой тип распределения означает, что пакеты могут часто обновляться. Разработчик пакета может легко выполнить непрерывное развертывание, поэтому каждый раз при исправлении ошибки или добавлении новой функции вы можете получить новую версию пакета.

Нет необходимости обновлять пакет до последней версии каждый раз, когда она выходит. Если вам не нужна новая функция, можете эту версию пропустить.

С другой стороны, пропускать все версии небезопасно. Некоторые разработчики пакетов добросовестно относятся к критическим изменениям, тогда как другие не так ответственны. Чем больше вы пропускаете обновлений, тем больше накапливается критических изменений. Двигаться вперед будет все труднее, и вы можете оказаться в ситуации, когда больше не сможете обновлять зависимости.

Это касается как языков, так и платформ. В конечном счете вы можете застрять на настолько старой версии своего языка, что вам будет сложно нанимать новых сотрудников. Такое бывает.

Поэтому регулярно загружайте обновления — это упростит вам работу. В логе репозитория Git вы можете ознакомиться с примером кода, где я время от времени обновлял зависимости. В листинге 14.1 представлен фрагмент лога.

Листинг 14.1. Фрагмент лога Git, отображающий обновления пакетов и несколько сопутствующих коммитов для наглядности

```
0964099 (HEAD) Add a schedule link to each day
2295752 Rename test classes
fdf2a2f Update Microsoft.CodeAnalysis.FxCopAnalyzers NuGet
9e5d33a Update Microsoft.AspNetCore.Mvc.Testing NuGet pkg
f04e6eb Update coverlet.collector NuGet package
3bfc64f Update Microsoft.NET.Test.Sdk NuGet package
a2bebea Update System.Data.SqlClient NuGet package
34b818f Update xunit.runner.visualstudio NuGet package
ff5314f Add cache header on year calendar
df8652f Delete calendar flag
```

Как часто нужно обновлять зависимости? Зависит от их количества и стабильности. Что касается примера кодовой базы, я решил, что

можно проверять наличие обновлений примерно раз в два месяца. В большой кодовой базе может содержаться на порядок больше пакетов — это потребует более частого обновления.

Другой фактор — частота изменения конкретных зависимостей. Одни меняются редко, тогда как другие постоянно пересматриваются. Вам придется поэкспериментировать, чтобы найти наиболее подходящий вариант для вашей кодовой базы. Возможно, имеет смысл привязать этот рабочий элемент к другому обычному действию. Если, например, вы используете Scrum с двухнедельными спринтами, можете запланировать обновление пакета как первое действие, которое вы будете делать в новом спринте[1].

14.2.2. Планирование других действий

Обновления зависимостей нужно планировать, так как об этом этапе легко забыть. Ежедневно проверять обновления бессмысленно, поэтому вряд ли это станет частью чьего-то рабочего ритма.

Это тот тип проблемы, которую замечают, когда становится уже слишком поздно. И в эту категорию входит не только обновление зависимостей.

Срок действия сертификатов[2] истекает только спустя несколько лет, поэтому можно легко забыть об их обновлении. Но если вы так сделаете, программа перестанет работать. К этому моменту в команде уже может не остаться ни одного из первоначальных разработчиков. Поэтому очень важно заблаговременно обновлять сертификаты. Запланируйте это действие заранее.

То же касается доменных имен. Срок их службы тоже составляет несколько лет. Убедитесь, что они обновляются.

Другой пример — резервное копирование базы данных. Его легко автоматизировать, но знаете ли вы, будет ли оно потом работать?

---

[1] Не откладывайте эту задачу на потом, иначе она будет выполнена некорректно.

[2] Например, сертификаты X.509.

Реально ли восстановить систему из резервной копии? Рассмотрите вопрос о проверке на регулярной основе. Случайно узнав, что резервные копии не работают, когда они действительно нужны, вы будете разочарованы.

14.2.3. Закон Конвея

На первой работе у меня был свой кабинет. Это было в 1994 году, когда открытые офисные пространства не были так распространены. С тех пор у меня никогда не было такого кабинета[1]. Работодатели узнали, что офисы открытого типа дешевле, а гибкие процессы, такие как XP (экстремальное программирование), требуют частого взаимодействия сотрудников [5].

Я не люблю такие варианты планировки офисных помещений, хотя допускаю, что общение лицом к лицу укрепляет сотрудничество. Если вы когда-нибудь пробовали вести письменное обсуждение в чат-форуме проблемы GitHub или спецификации функции, то знаете, что это может затянуться на дни или недели. Часто вы можете решить то, что выглядит как конфликт, *поговорив* с другим человеком около 15 минут.

Личное взаимодействие всегда поможет разрешить недопонимание.

С другой стороны, если вы постоянно общаетесь и сидите вместе, то рискуете установить в команде культуру устного общения, при которой ничего не записывается, и вы должны каждый раз отвечать на одни и те же вопросы, а с уходом людей теряются и знания.

Рассмотрим это сквозь призму закона Конвея:

> *«Организации проектируют системы [...] дизайн которых отражает структуру коммуникаций, сложившуюся в этих организациях» [21].*

[1] Это не совсем так. Я много лет работаю не в найме, и, когда нет необходимости работать напрямую с клиентом, я работаю дома. Даже сейчас я пишу эту книгу в своем домашнем офисе.

Если все сидят вместе и могут произвольно общаться с кем угодно, в результате может выстроиться система без четкой структуры, но с большим количеством связей. Другими словами, спагетти-код [15].

Подумайте над тем, как способ организации работы влияет на кодовую базу.

Хотя мне не нравятся открытые офисы и пустая болтовня, я не рекомендую переходить и в другую крайность. Жесткие иерархии и цепочки подчинения вряд ли поспособствуют продуктивности. Лично мне нравится организовывать работу (даже корпоративную) так, как обычно организуют ПО с открытым исходным кодом: с пул-реквестами, код-ревью и в основном письменным общением. Все это дает возможность асинхронной разработки ПО [96].

Вам не обязательно повторять за мной, но организуйте свою команду так, чтобы способствовать как общению, так и программной архитектуре, которая вам нравится. Суть в том, чтобы знать, что организация команды и архитектура связаны.

14.3. ЗАКЛЮЧЕНИЕ

Книг о личной продуктивности довольно много, поэтому я не включал большинство тем, которые обычно обсуждаются в них. Ваш способ работы зависит от вас, и организация работы команды может быть разнообразной.

Я хотел обсудить несколько правил, которые я вывел для себя за многие годы работы. Делайте перерывы, отойдите от компьютера — лучшие идеи могут возникнуть, когда вы занимаетесь чем-то отвлеченным.

15 ОЧЕВИДНЫЕ АСПЕКТЫ

Что насчет производительности? Безопасности? Анализа зависимости? Алгоритмов? Архитектуры? Информатики?

Все это — базовые понятия программной инженерии. До сих пор я делал вид, что их не существует. Это не потому, что я считаю их неактуальными. Я просто думаю, что эти понятия уже давно знакомы каждому программисту.

Когда я консультирую команды разработчиков, мне редко приходится учить их насчет производительности. Я часто сталкиваюсь с ситуацией, когда участник команды знает об алгоритмах и информатике больше, чем я. Несложно будет найти и кого-то, кто осведомлен о безопасности лучше, чем я.

Главная тема моей книги — практики, которым я обучаю. Надеюсь, что она поможет заполнить некоторые пробелы в знаниях (восклицательный знак на рис. 15.1), даже если это будут базовые знания.

Мое желание сосредоточиться на других вещах не означает, что я игнорировал самые очевидные понятия. В предпоследней главе мне бы хотелось обсудить мой подход к производительности, безопасности и некоторым другим аспектам.

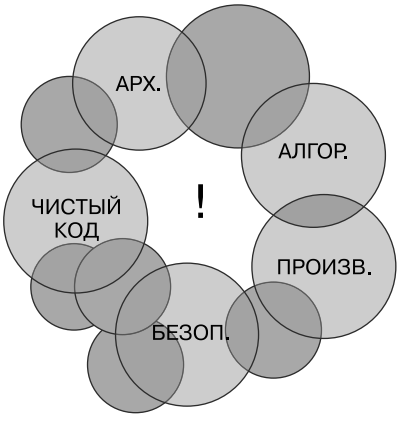


Рис. 15.1. Самые очевидные понятия в программной инженерии: архитектура, алгоритмы, производительность, безопасность и подход к коду, описанные в таких книгах, как «Чистый код» [61] и «Совершенный код» [65]. Эти темы раскрыты и в других источниках, но я думаю, что без комплексного подхода можно что-то упустить. В своей книге я пытаюсь восполнить такой пробел

15.1. ПРОИЗВОДИТЕЛЬНОСТЬ

Я заметил одну интересную закономерность. Когда я предлагаю идею, которая кому-то не нравится, то иногда по выражению лиц могу сказать, что люди изо всех сил пытаются придумать контраргумент. Но спустя некоторое время я слышу: «А как же производительность?»

Действительно, что с производительностью? Я признаю, что узкая направленность на производительность некоторых специалистов меня раздражает, но думаю, что понимаю, откуда она взялась. Полагаю, что все дело в устаревших знаниях.

15.1.1. Устаревшие знания

Десятки лет компьютеры были очень медленными. Они могли считать быстрее, чем человек, но по сравнению с современными машинами их скорость была совсем невысока. Когда индустрия начала

превращаться в академическую дисциплину Computer Science, производительность была приоритетным вопросом. При применении неэффективных алгоритмов программа могла стать непригодной для использования.

Неудивительно, что типичная учебная программа по информатике будет включать алгоритмы, теорию вычислительной сложности с О-нотацией и внимание к объему памяти. Но дело в том, что эта учебная программа устарела.

Производительность все еще важна, но современные компьютеры настолько быстрые, что иногда сложно уловить разницу между ними. Имеет ли значение, что тот или иной метод возвращает результат через 10 или 100 наносекунд? С точки зрения узкого цикла, может быть, и важно, но, как правило, не сильно.

Я знаком с разработчиками, которые тратили часы на то, чтобы сократить вызов метода на несколько микросекунд, только чтобы запросить БД и получить результат[1]. Нет смысла оптимизировать операцию, если вы объедините ее с другой операцией на несколько порядков медленнее. Если вы должны сосредоточиться на производительности, по крайней мере оптимизируйте критические места.

Производительность никогда не должна быть в центре внимания. Важнее всего *правильность* работы программы. Джеральд Вайнберг рассказывает о том, «как донести эту мысль до тех, чьи умы запутались в вопросах эффективности и других второстепенных вещах» [115]. Это история о сорванном программном проекте и разработчике, нанятом, чтобы исправить ошибки. ПО, о котором идет речь, безнадежно сложное, содержит много ошибок и находится на грани удаления. Главный герой истории переписывает код так, чтобы он работал, и представляет его изначальным авторам.

Главный разработчик оригинального ПО интересуется, сколько времени нужно на запуск программы. Услышав ответ, он отвергает новую

[1] На всякий случай поясню: запрос к базе данных обычно занимает миллисекунды. Со временем все начинает работать быстрее, так что на момент чтения этой книги информация может стать уже неактуальной.

идею, так как дефектная программа работает в десять раз быстрее. На что главный герой отвечает: «Но она ведь не работает. Если программа не должна работать, я могу написать такую, запуск которой занимает одну миллисекунду на каждую карту»[1] [115].

Сначала заставьте программу работать, а уже потом, возможно, думайте о производительности. Может быть, безопасность тоже важна. Может быть, вам нужно посоветоваться с другими заинтересованными сторонами о том, как расставить приоритеты.

И если выяснится, что для них важна производительность, так тому и быть! Современные компиляторы сложны. Они могут встраивать вызовы методов, оптимизировать плохо структурированные циклы и пр. Генерируемый ими машинный код может выглядеть совсем не так, как вы себе представляете. Кроме того, производительность очень чувствительна к таким вещам, как используемое оборудование, установленное ПО, действия других процессов, и ко многим другим факторам [59]. Не рассуждайте о производительности слишком долго. Но если вы считаете, что это важно, — хорошо.

15.1.2. Удобочитаемость

Многие люди фокусируются на производительности, чтобы улучшить удобочитаемость. Я заимствовал эту идею из книги на совершенно другую тему *Seeing Like a State* [90].

Автор утверждает, что некоторые схемы создаются, чтобы сделать непонятное понятным. В качестве примера объясняется, как введение кадастровых карт (рис. 15.2) стало решением проблемы такого рода. Культура средневековых деревень была по большей части устной: только местные жители знали, кто и когда имеет право использовать какой участок земли. Это лишило власти возможности напрямую взимать налоги. Только местные дворяне обладали достаточными знаниями, чтобы облагать крестьян налогом [90].

---

[1] Это было еще во времена перфокарт.

Поскольку феодализм уступил место централизованным государствам, королям нужны были способы обойти свою зависимость от местной знати. Кадастровая карта была способом внести ясность в непонятный им мир [90].

Но при переводе многое может быть утеряно. Например, в средневековых деревнях право пользования участком земли могло быть привязано к другим критериям, а не только к положению в обществе. Люди могли, например, выращивать урожай на определенном участке земли в течение вегетационного периода. После сбора урожая вся земля становилась общим достоянием без каких-либо индивидуальных прав до следующего периода возделывания. Кадастровые карты не могли зафиксировать такие сложные «бизнес-правила», поэтому они установили и кодифицировали упрощенную форму собственности. Такие карты не фиксировали текущее положение дел — они изменили реальность.

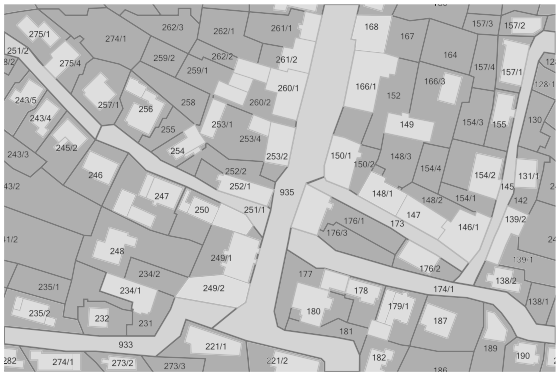


Рис. 15.2. Кадастровые карты были введены королями, чтобы обойти свою зависимость от местной знати. Они вносили ясность за счет деталей. Будьте внимательны, чтобы не перепутать карту с местностью

В разработке ПО выполняется много событий. Большинство из них мы не можем увидеть, поэтому в попытках понять их вносим всевозможные измерения и процессы. После того как мы вводим такие

устройства, они начинают формировать наше восприятие. Как говорится, тому, кто держит молоток, мир кажется гвоздем.

Однажды я помогал одной компании разобраться в теме непрерывного развертывания. После того как я провел несколько недель с разными разработчиками, один из руководителей поинтересовался: «Кто из моих разработчиков хорош?»

Он не был «технарем». Он никогда не программировал, поэтому не мог определить сам. Но я воспринял его вопрос как неэтичный, поскольку разработчики доверяли мне. Я так и не ответил.

Руководителям сложно управлять разработкой ПО. Как же можно измерить что-то настолько нематериальное? Обычно они вводят косвенные показатели, например отработанные часы. Если вы когда-либо получали почасовую оплату, то знаете, насколько это неэффективно.

Я думаю, что для некоторых зацикленность на производительности — это на самом деле попытка осознать неосязаемую природу своей профессии. Будучи измеримой, производительность становится кадастровой картой разработки ПО.

Для некоторых искусство программной инженерии — источник большого дискомфорта, а постановка вопроса о производительности вносит толику ясности.

15.2. БЕЗОПАСНОСТЬ

Безопасность ПО как страховка: вы действительно не хотите платить за это, но, не сделав этого, вы пожалеете.

Как и во многих других аспектах программной инженерии, безопасность заключается в нахождении баланса. Нет такого понятия, как полностью безопасная система. Даже если вы отключите интернет и поставите у компьютера вооруженную охрану, все равно найдется способ пробраться к нему — допустим, подкупом или грубой силой.

Чтобы определить риски и принять соответствующие меры по их устранению, вам придется работать с другими заинтересованными сторонами.

15.2.1. Модель угроз STRIDE

Для выявления потенциальных проблем безопасности вы можете использовать модель угроз STRIDE [48]. Чтобы обнаружить возможные проблемы, вы можете включить ее в свой чек-лист.

- **Спуфинг (spoofing).** Злоумышленники пытаются выдать себя за тех, кем не являются, чтобы получить несанкционированный доступ к системе.
- **Незаконное изменение (tampering).** Злоумышленники пытаются подделать данные, например путем SQL-инъекции.
- **Отказ от авторства (repudiation).** Злоумышленники отрицают, что выполнили какое-либо действие, за которое им заплатили.
- **Раскрытие информации (information disclosure).** Злоумышленники считывают данные, которые не должны считывать. Как пример: атака с применением технологии «незаконный посредник» (man-in-the-middle) и SQL-инъекции.
- **Отказ в обслуживании (denial of service).** Злоумышленники пытаются сделать систему недоступной для обычных пользователей.
- **Повышение привилегий (elevation of privilege).** Злоумышленники пытаются получить больше разрешений, чем у них есть.

Над моделированием угроз обычно работают как программисты, ИТ-специалисты, так и другие заинтересованные лица, например владельцы бизнеса. Одни проблемы лучше решать в коде, другие — в конфигурации сети, а с третьими вы действительно ничего не сможете сделать.

Например, нельзя полностью предотвратить отказ в обслуживании для онлайн-системы. Когда корпорация Microsoft разработала модель STRIDE, бóльшая часть их сетевого кода была написана на C и C++. Эти языки чувствительны к переполнению буфера [4], поэтому отправка злонамеренных данных может вызвать в них сбой или зависание системы.

Хотя управляемый код, например написанный на C# и Java, предотвращает многие такие проблемы, вы не можете гарантировать, что распределенная атака типа «отказ в обслуживании» не «убьет» вашу систему. Вы можете попытаться обеспечить достаточную мощность, чтобы справиться с увеличением трафика, но если атака достаточно массовая, это не поможет.

У разных систем разные профили угроз. Мобильные или настольные приложения подвержены бóльшим видам атак, чем веб-сервис. Давайте создадим модель угроз для системы бронирования столиков в ресторане. Как вы помните, это REST API, позволяющий клиентам выполнять и редактировать резервирование. Кроме того, метрдотель может сделать GET-запрос к ресурсу, чтобы увидеть расписание на день, включая все заказы и информацию о посетителях. Расписание включает имена и адреса электронной почты гостей.

Я пройдусь по каждому из пунктов STRIDE, как если бы это был чек-лист, но я буду делать это только неформально, чтобы дать вам базовое представление. Возможно, вы захотите рассмотреть более систематический подход.

15.2.2. Спуфинг

Уязвима ли система для спуфинга? Да, при резервировании вы можете представиться кем угодно. Вы можете просто назвать имя Киану Ривза, и система его примет. Это проблема? Возможно, но нам придется уточнить у владельцев заведений, создаст ли им это проблемы.

Текущая реализация системы не принимает никаких решений на основе имени, поэтому спуфинг не изменит ее поведения.

15.2.3. Незаконное изменение

Чувствительна ли система к незаконному изменению? Система содержит таблицу резервирований в БД SQL Server. Может ли кто-то редактировать эти данные без доступа к ним?

Здесь есть несколько сценариев.

Сам REST API позволяет редактировать резервирование с помощью HTTP-запросов `PUT` и `DELETE`. Точно так же, как вы можете добавить новое резервирование без аутентификации, вы можете отредактировать его, если у вас есть адрес ресурса (то есть URL-адрес). Стоит ли нам беспокоиться? И да и нет. Каждый адрес ресурса однозначно идентифицирует одно резервирование. Одна часть адреса — это идентификатор резервирования, который представляет собой GUID. Злоумышленник не может угадать GUID, так что это должно нас немного успокоить[1]. С другой стороны, при добавлении нового резервирования ответ на запрос `POST` включает заголовок `Location` с адресом ресурса. «Незаконный посредник» сможет перехватить ответ и увидеть адрес.

Есть простое решение для этой угрозы: HTTPS. Безопасное соединение не должно быть опциональным — оно должно быть обязательным. Это хороший пример смягчения последствий, с которым лучше справится ИТ-специалист.

Обычно это вопрос правильной настройки сервиса, а не написания кода.

Еще один сценарий несанкционированного доступа, который нужно учитывать, — это прямой доступ к БД. Можно ли его получить? Зависит от обеспечения безопасности развертывания базы данных или

[1] Возможно, это звучит как «безопасность от неизвестности». Но это не так. GUID так же сложно угадать, как и любой другой 128-битный криптографический ключ. В конце концов, это всего лишь 128-битное число.

уверенности в том, что облачная БД достаточно защищена. Опять же, требуемые компетенции указывают на ИТ-специалистов, а не на узконаправленных программистов.

Злоумышленник может получить доступ к базе с помощью SQL-инъекции. Ответственность за устранение такой угрозы полностью ложится на программистов. Кодовая база резервирования столиков в ресторане использует именованные параметры (листинг 15.1). При использовании ADO.NET это рекомендуемая защита от SQL-инъекций.

Листинг 15.1. Использование именованного параметра SQL @id (Restaurant/e89b0c2/Restaurant.RestApi/SqlReservationsRepository.cs)

```
public async Task Delete(Guid id)
{
    const string deleteSql = @"
        DELETE [dbo].[Reservations]
        WHERE [PublicId] = @id";

    using var conn = new SqlConnection(ConnectionString);
    using var cmd = new SqlCommand(deleteSql, conn);
    cmd.Parameters.AddWithValue("@id", id);

    await conn.OpenAsync().ConfigureAwait(false);
    await cmd.ExecuteNonQueryAsync().ConfigureAwait(false);
}
```

Поскольку защита от атак через внедрение SQL-кода — обязанность разработчиков, обратите на нее внимание во время код-ревью и при парном программировании.

15.2.4. Отказ от авторства

Могут ли пользователи системы отрицать, что они совершили действие? Да, что еще хуже, они могут сделать бронь и впоследствии никогда ею не воспользоваться. Это проблема не только ресторанов, но и больниц, парикмахерских и многих других мест, где осуществляется предзапись.

Можем ли мы снизить этот риск? Мы могли бы потребовать аутентификацию или цифровую подпись при регистрации. Можно попросить пользователей внести предоплату по карте. Но важно узнать у владельцев ресторанов, как они к этому относятся.

Большинство заведений могут быть обеспокоены тем, что такие жесткие меры спугнут клиентов. Это еще один пример того, когда безопасность предполагает поиск приемлемого баланса. Вы можете сделать систему настолько безопасной, что она перестанет выполнять свое назначение.

15.2.5. Раскрытие информации

Чувствительна ли система бронирования к раскрытию информации? Она не хранит пароли, но хранит адреса электронной почты посетителей, которые мы должны рассматривать как личную информацию, которая не должна попасть на глаза злоумышленникам.

Важно учитывать и адрес ресурса (URL) каждого резервирования. Если у вас есть такой, вы можете удалить ресурс (с помощью `DELETE`) для получения доступа к уже забронированному месту, удалив чье-то резервирование.

Как злоумышленник может получить доступ к такой информации? Например, с помощью «незаконного посредника». Но мы уже решили использовать HTTPS, так что бояться не стоит. SQL-инъекция может быть еще одним вектором атаки, но мы уже знаем, как решить эту проблему. Я не думаю, что нам стоит волноваться.

Но остается еще одна проблема. Метрдотель ресторана может сделать GET-запрос к ресурсу, чтобы увидеть расписание на день, включая все заказы, время прибытия, имена и адреса электронной почты посетителей, чтобы они могли идентифицировать себя по прибытии.

В листинге 15.2 приведен пример такого взаимодействия с уже установленным средством защиты.

Листинг 15.2. Пример расписания GET-запроса и соответствующего ему ответа. По сравнению с тем, что выдает реальная система из примера, я упростил как запрос, так и ответ, чтобы выделить важные моменты

```
GET /restaurants/2112/schedule/2021/2/23 HTTP/1.1
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInCI6IkpXVCJ9.eyJ...

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
{
  "name": "Nono",
  "year": 2021,
  "month": 2,
  "day": 23,
  "days": [{
    "date": "2021-02-23",
    "entries": [{
      "time": "19:45:00",
      "reservations": [{
        "id": "2c7ace4bbee94553950afd60a86c530c",
        "at": "2021-02-23T19:45:00.0000000",
        "email": "anarchi@example.net",
        "name": "Ann Archie",
        "quantity": 2
      }]
    }]
  }]
}
```

Упрощение здесь в том, чтобы потребовать от метрдотеля подтверждения аутентификации. Для механизма аутентификации я выбрал веб-токен JSON. Если клиент не представляет верный токен с подтверждением, то получает ответ `403 Forbidden`.

Для проверки правильности поведения вы можете написать интеграционные тесты, как в листинге 15.3.

Проверка подлинности нужна только для ресурса *расписания*, поскольку лишь он содержит конфиденциальную информацию. Так как владельцы не хотят отпугивать клиентов требованием их аутентификации, лучше просить сотрудников делать это.

Листинг 15.3. Тест, проверяющий, не представляет ли клиент действительный веб-токен JSON с подтверждением роли MaitreD. API отклоняет запрос с ответом 403 Forbidden. В этом тесте данные о роли неверны — Foo и Bar (Restaurant/0e649c4/Restaurant.RestApi.Tests/ScheduleTests.cs)

```
[Theory]
[InlineData(    1, "Hipgnosta")]
[InlineData( 2112, "Nono")]
[InlineData(90125, "The Vatican Cellar")]
public async Task GetScheduleWithoutRequiredRole(
    int restaurantId,
    string name)
{
    using var api = new SelfHostedApi();
    var token =
        new JwtTokenGenerator(new[] { restaurantId }, "Foo", "Bar")
            .GenerateJwtToken();
    var client = api.CreateClient().Authorize(token);

    var actual = await client.GetSchedule(name, 2021, 12, 6);

    Assert.Equal(HttpStatusCode.Forbidden, actual.StatusCode);
}
```

15.2.6. Отказ в обслуживании

Может ли злоумышленник передать поток байтов в REST API, чтобы вызвать сбой? Если да, то я думаю, что мы ничего не сможем с этим сделать.

API, написанный на высокоуровневом языке, например C#, Java или JavaScript, не работает, управляя указателями. Переполнение буфера, вызывающее сбой системы, не может произойти с управляемым кодом. Вернее, если это так, то это не ошибка в пользовательском коде, а ошибка самой платформы. Мы ничего не сможем сделать, чтобы устранить такую угрозу, кроме как поддерживать рабочую систему в актуальном состоянии.

Будет ли проблемой распределенная атака типа «отказ в обслуживании»? Скорее всего, да. Мы поинтересовались у наших ИТ-специалистов, смогут ли они решить проблему.

Мы можем подумать о том, как сделать систему более устойчивой к неожиданным большим объемам трафика. Для некоторых систем это может быть хорошим решением. Например, система продажи билетов на концерты тесно связана с системой резервирования мест в ресторанах. Популярный артист, дающий концерт на стадионе, может легко нагрузить систему тысячами запросов в секунду после поступления билетов в продажу.

Один из способов сделать такую систему устойчивой к нагрузкам — это соответствующе ее спроектировать. Например, можно поместить все потенциальные операции записи в долговременную очередь, а операции чтения будут основаны на материализованных представлениях. Это предполагает архитектуру вроде CQRS (в этой книге не описана).

CQRS-архитектура сложнее обработки записей по мере их возникновения. Можно было бы спроектировать систему бронирования столиков таким образом, но мы (мои воображаемые заинтересованные стороны и я) решили, что она не обеспечивает хорошей окупаемости инвестиций.

При моделировании угроз можно определить угрозу, только чтобы решить не реагировать на нее. В конце концов, это бизнес-решение. Просто убедитесь, что остальные сотрудники вашей организации понимают и принимают все риски.

15.2.7. Повышение привилегий

Возможно ли, что злоумышленник представится обычным пользователем, а затем каким-то хитрым способом получит права администратора?

Опять же SQL-инъекция — самая распространенная проблема. Если злоумышленники могут выполнять произвольные SQL-запросы в базе данных, они могут и запускать внешние процессы в ОС[1].

[1] Например, в SQL Server можно запустить хранимую процедуру xp_cmdshell. Но, начиная с SQL Server 2005, эта функция по умолчанию отключена. Не включайте ее.

Эффективное средство — запуск БД и всех других служб с максимально ограниченными разрешениями. Не запускайте базу от имени администратора.

Поскольку мы уже решили при написании кода учитывать атаки с помощью SQL-атаки, меня не слишком беспокоят такие угрозы.

Итак, мы рассмотрели пример модели угроз STRIDE для системы резервирования столиков. Разумеется, что проектирование безопасности имеет гораздо большее значение, но, как отдаленный от области безопасности человек, я склонен подходить к этому именно так. Если во время моделирования угроз я обнаружу проблему, с которой я не буду знать, как бороться, я обращусь к другу-специалисту.

15.3. ПРОЧИЕ ТЕХНИКИ

Производительность и безопасность — это два важнейших аспекта классической разработки ПО. Но есть много других методик, о которых нужно знать. Темы, которые я раскрываю, основаны на моем опыте и на вопросах, которые обычно возникают во время проведения консультаций для команд. Остальные темы я опускаю не потому, что они неважны.

Вам могут пригодиться методы, включающие канареечные релизы и A/B-тестирование [49], отказоустойчивость и способность быстро восстанавливаться [73], анализ зависимостей, лидерство, алгоритмы распределенных систем [55], архитектуру, конечные автоматы, паттерны проектирования [39; 33; 66; 46], непрерывное развертывание [49], принципы SOLID [60] и пр. Поле не просто обширно, оно продолжает расти.

Но мне бы хотелось кратко объяснить две другие темы.

15.3.1. Тестирование на основе свойств

Разработчикам, незнакомым с автоматизированным тестированием, часто сложно определять тестовые значения. Одна из причин в том, что иногда в тест нужно включить определенные значения,

даже если они не имеют отношения к тест-кейсу. Давайте рассмотрим листинг 15.4, в котором проверяется, генерирует ли конструктор `Reservation` исключение `ArgumentOutOfRangeException`, если предоставленное количество не является натуральным числом.

Этот параметризованный тест использует значения `0` и `-1` как примеры недопустимых величин. Значение `0` — граничное [66], поэтому должно быть включено, но точно отрицательное число неважно. Значение `-42` было бы так же полезно, как `-1`.

Листинг 15.4. Параметризованный тест, проверяющий, генерирует ли конструктор Reservation исключение ArgumentOutOfRangeException для недопустимого количества (Restaurant/812b148/Restaurant.RestApi.Tests/ReservationTests.cs)

```
[Theory]
[InlineData( 0)]
[InlineData(-1)]
public void QuantityMustBePositive(int invalidQuantity)
{
    Assert.Throws<ArgumentOutOfRangeException>(
        () => new Reservation(
            Guid.NewGuid(),
            new DateTime(2024, 8, 19, 11, 30, 0),
            new Email("vandal@example.com"),
            new Name("Ann da Lucia"),
            invalidQuantity));
}
```

Зачем придумывать числа, если подойдет любое отрицательное? Что, если бы был фреймворк, способный создавать произвольные отрицательные числа?

Есть несколько таких многопользовательских программных пакетов. Это основная идея *тестирования на основе свойств*[1]. В дальнейшем

[1] Термин «свойство» здесь означает черту, качество или признак. Так, тестирование на основе свойств включает проверку свойства тестируемой системы, например того, что конструктор Reservation выдает исключение для всех неположительных величин. В этом контексте свойство не имеет ничего общего со свойствами языков программирования C# или Visual Basic (геттеры или сеттеры).

я буду использовать библиотеку FsCheck, но есть и другие[1]. FsCheck интегрируется как с xUnit.net, так и с NUnit, поэтому вы можете легко объединять свои тесты на основе свойств с более традиционными. Это упрощает и рефакторинг существующих тестов в тесты на основе свойств (листинг 15.5).

Атрибут `[Property]` помечает метод как тест на основе свойств, управляемый библиотекой FsCheck. Он похож на параметризованный тест, но все аргументы метода теперь генерируются FsCheck, а не предоставляются атрибутами `[InlineData]`.

Значения генерируются случайно, обычно смещены в сторону типичных граничных значений, таких как 0, 1, –1 и т. д. По умолчанию каждое свойство выполняется 100 раз. Представьте, что это 100 атрибутов `[InlineData]`, дополняющих один тест, но каждое из значений случайно регенерируется при каждом его выполнении.

Листинг 15.5. Тест из листинга 15.4 после рефакторинга в тест на основе свойств (Restaurant/05e64f5/Restaurant.RestApi.Tests/ReservationTests.cs)

```
[Property]
public void QuantityMustBePositive(NonNegativeInt i)
{
    var invalidQuantity = -i?.Item ?? 0;
    Assert.Throws<ArgumentOutOfRangeException>(
        () => new Reservation(
            Guid.NewGuid(),
            new DateTime(2024, 8, 19, 11, 30, 0),
            new Email("vandal@example.com"),
            new Name("Ann da Lucia"),
            invalidQuantity));
}
```

Библиотека FsCheck поддерживает такие встроенные типы оболочек, как `PositiveInt`, `NonNegativeInt` и `NegativeInt`. Это просто оболочки для целых чисел, но с гарантией того, что FsCheck будет генерировать

[1] Исходная библиотека тестирования на основе свойств — это пакет Haskell QuickCheck. Он был впервые выпущен в 1999 году и до сих пор актуален. Есть много портированных пакетов для других языков.

только соответствующие описанию значения: только неотрицательные целые числа[1] для `NonNegativeInt` и т. д.

Для теста `QuantityMustBePositive` нам действительно нужны произвольные неположительные целые числа, но такого типа обертки не существует. Один из способов получения значений в желаемом диапазоне — обратиться к библиотеке FsCheck для создания значений `NonNegativeInt`, а затем их инвертировать.

Свойство `Item`[2] возвращает целое число, обернутое в значение `NonNegativeInt`. Один из анализаторов статического языка, который я включил, указывает на то, что параметр `i` может принимать значение `null`. Все символы вопросительных знаков — это способ C# обрабатывать возможные нулевые ссылки, заканчивающиеся резервным значением `0`. Я считаю это шумом. Важная операция — унарный оператор минуса перед `i`. Он инвертирует неотрицательное целое число в неположительное.

Как только вы поймете, что можете позволить библиотеке вроде FsCheck создавать произвольные тестовые значения, вы посмотрите на другие тестовые данные в новом свете. А что насчет `Guid.NewGuid()`? Почему бы вам вместо этого не позволить библиотеке FsCheck производить это значение?

Действительно, как показано в коде листинга 15.6, это возможно.

Листинг 15.6. Свойство из листинга 15.5 изменено, чтобы позволить библиотеке FsCheck тоже создавать идентификатор резервирования (Restaurant/87fefaa/Restaurant.RestApi.Tests/ReservationTests.cs)

```
[Property]
public void QuantityMustBePositive(Guid id, NonNegativeInt i)
{
    var invalidQuantity = -i?.Item ?? 0;
    Assert.Throws<ArgumentOutOfRangeException>(
```

[1] То есть числа больше нуля или равные ему.

[2] Здесь приведено свойство языка C#. Это не относится к свойству тестирования на основе свойств. Действительно, перегруженная терминология может легко вас запутать.

```
        () => new Reservation(
            id,
            new DateTime(2024, 8, 19, 11, 30, 0),
            new Email("vandal@example.com"),
            new Name("Ann da Lucia"),
            invalidQuantity));
}
```

На самом деле ни одно из жестко запрограммированных значений не влияет на результат теста. Вместо записи `vandal@example.com` вы можете использовать любую строку адреса электронной почты. То же касается и имени: вместо `Ann da Lucia` вы можете использовать любую строку. Библиотека FsCheck в результате выдаст вам значения как в листинге 15.7.

Листинг 15.7. Свойство из листинга 15.6 изменено, чтобы разрешить библиотеке FsCheck производить все параметры (Restaurant/af31e63/Restaurant.RestApi.Tests/ReservationTests.cs)

```
[Property]
public void QuantityMustBePositive(
    Guid id,
    DateTime at,
    Email email,
    Name name,
    NonNegativeInt i)
{
    var invalidQuantity = -i?.Item ?? 0;
    Assert.Throws<ArgumentOutOfRangeException>(
        () => new Reservation(id, at, email, name, invalidQuantity));
}
```

Вы можете зайти с этой концепцией на удивление далеко. Рано или поздно вы столкнетесь с особыми требованиями к входным данным, которые не сможете просто смоделировать с помощью одного из встроенных типов-оболочек, например `NonNegativeInt`. У хорошей библиотеки тестирования на основе свойств, такой как FsCheck, есть API для таких ситуаций.

Я часто обнаруживаю, что мне труднее придумать исчерпывающие тест-кейсы, чем описать общие свойства тестируемой системы.

Это произошло дважды, пока я разрабатывал пример системы резервирования столиков.

Для сложной логики, учитывая мнение метрдотеля, касающееся расписания, я настойчиво пытался придумать конкретные тест-кейсы. Когда я осознал ситуацию, я переключился на определение поведения, используя последовательности все более и более конкретных свойств[1] (листинг 15.8).

Листинг 15.8. Базовая реализация расширенного теста на основе свойств. Этот тестовый метод настраивается и вызывается кодом из листинга 15.9 (Restaurant/af31e63/Restaurant.RestApi.Tests/MaitreDScheduleTests.cs)

```
private static void ScheduleImp(
    MaitreD sut,
    Reservation[] reservations)
{
    var actual = sut.Schedule(reservations);

    Assert.Equal(
        reservations.Select(r => r.At).Distinct().Count(),
        actual.Count());
    Assert.Equal(
        actual.Select(ts => ts.At).OrderBy(d => d),
        actual.Select(ts => ts.At));
    Assert.All(actual, ts => AssertTables(sut.Tables, ts.Tables));
    Assert.All(
        actual,
        ts => AssertRelevance(reservations, sut.SeatingDuration, ts));
}
```

На самом деле это реализация теста, который получает аргумент `MaitreD` и массив резервирований (`reservations`), чтобы вызвать метод `Schedule`.

[1] Прогрессию коммитов и окончательный результат можно найти в репозитории Git, который прилагается к книге. Я считаю этот пример кода слишком специфичным, чтобы рассматривать пошагово здесь, но я подробно описал его в своем блоге [108].

Есть еще один метод, использующий FsCheck API для правильной настройки аргументов `sut` и `reservation`, вызывающий `ScheduleImp`, — метод, который запускается фреймворком модульного тестирования (см. листинг 15.9).

Листинг 15.9. Конфигурация и выполнение основного свойства из листинга 15.8 (Restaurant/af31e63/Restaurant.RestApi.Tests/MaitreDScheduleTests.cs)

```
[Property]
public Property Schedule()
{
    return Prop.ForAll(
        (from rs in Gens.Reservations
         from m in Gens.MaitreD(rs)
         select (m, rs)).ToArbitrary(),
        t => ScheduleImp(t.m, t.rs));
}
```

Это свойство использует продвинутые возможности библиотеки FsCheck (не описано в книге). Если вы не знакомы с FsCheck API, подробности будут для вас малопонятными. Это нормально. Я не приводил пример кода, для того чтобы обучить вас FsCheck. Я добавил его, чтобы показать более широкие возможности программной инженерии по сравнению с теми, что описаны здесь.

15.3.2. Поведенческий анализ кода

В этой книге я в основном подробно рассматривал код. Вы можете и должны учитывать влияние и ценность каждой строки. Но это не значит, что общая ситуация неважна. В подразделе 7.2.6 я говорил о фрактальной архитектуре, но также и о важности общей картины.

Но это статический взгляд на кодовую базу. Когда вы анализируете код, даже высокоуровневый, вы видите его таким, какой он есть на данный момент. С другой стороны, у вас есть система контроля версий, чтобы

проанализировать его для получения дополнительного понимания. Какие файлы изменяются чаще всего? Какие изменяются одновременно? Работают ли определенные программисты только с определенными файлами?

Анализ данных контроля версий был академической дисциплиной [44], но в своих книгах [111; 112] Адам Торнхилл проделал большую работу, чтобы сделать его практическим. Вы можете сделать поведенческий анализ кода частью конвейера непрерывного развертывания.

Поведенческий анализ кода извлекает информацию из Git для выявления шаблонов и проблем, которые могут быть обнаружены только со временем. Даже файл с низкой цикломатической сложностью и малым размером может быть проблематичным по другим причинам. Например, он может быть связан с другими более сложными файлами.

Некоторую связанность можно определить с помощью анализа зависимостей, но найти другие виды связанности может быть труднее. Особенно это касается копирования и вставки кода. Анализируя, какие файлы и части файлов изменяются одновременно, вы можете обнаружить зависимости, которые в иной ситуации могли быть неочевидными [112]. На карте связанности на рис. 15.3 показано, какие файлы чаще всего изменяются одновременно.

Вы можете проанализировать ее, чтобы увидеть «рентген-снимок» одного файла [112]. Какие методы вызывают больше всего проблем?

С помощью правильных инструментов вы можете создавать и карты «горячих точек» в своем коде (рис. 15.4). На таких интерактивных *диаграммах вложенности* каждый файл представлен окружностью, размер которой указывает на размер файла или сложность, а цвет — на частоту изменений. Чем больше коммитов в файле, тем интенсивнее цвет [112].

Поведенческий анализ кода можно использовать как активный инструмент программной инженерии. Вы можете не только создавать привлекательные диаграммы, но и количественно определять связан-

ность изменений и «горячие точки», так чтобы использовать числа в качестве пороговых значений для дальнейшего исследования.

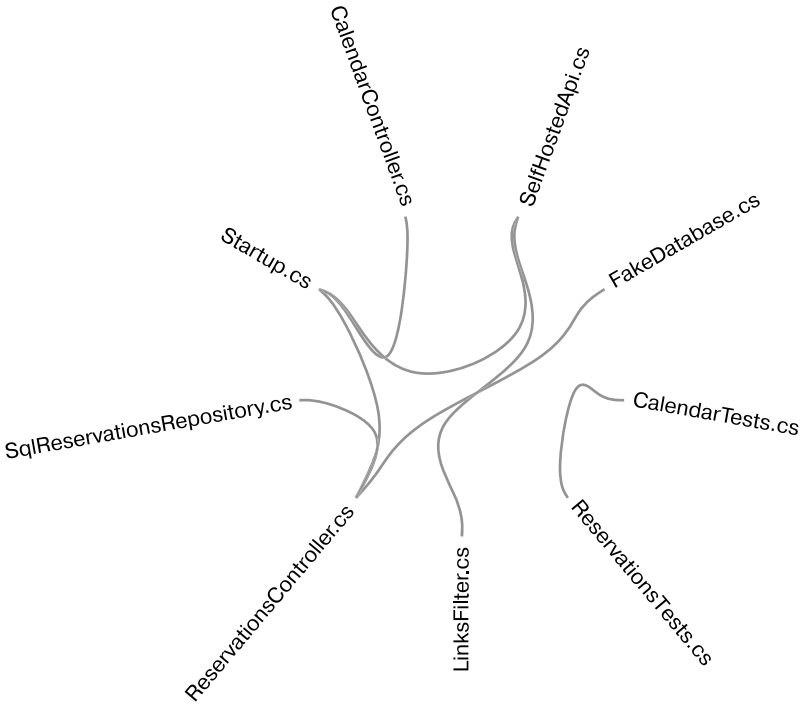


Рис. 15.3. Карта связанности изменений. Файлы, связанные линией, — это файлы, которые изменяются одновременно. В анализируемой кодовой базе больше файлов, но в диаграмму включены лишь те, что изменяются одновременно выше определенного порога

Вы можете следить за тенденциями. Они также поддаются воздействию. Если вы не начинаете с разработки новой кодовой базы, ваши цифры могут выглядеть не очень хорошо, но по крайней мере вы можете немедленно начать улучшать ситуацию.

Если вы часть большой команды, то можете использовать поведенческий анализ кода, чтобы определить распределение знаний

и взаимосвязь команды. Вариант диаграммы вложенности «горячих точек» — это карта знаний, на которой «главный автор» каждого файла определен разными цветами. Это приближает к реальной количественной оценке «фактора автобуса» для команды.

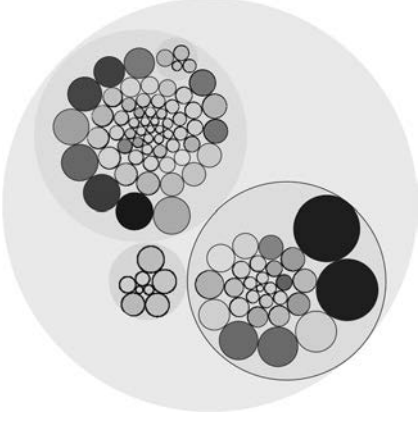


Рис. 15.4. Диаграмма вложенности «горячих точек». Чем больше окружность, тем сложнее файл. Чем интенсивнее цвет, тем чаще он меняется. Эти диаграммы обычно напоминают бактерии в чашке Петри

15.4. ЗАКЛЮЧЕНИЕ

Когда вы слышите термин *«программная инженерия»*, вы, скорее всего, думаете о классических практиках и дисциплинах, таких как проектирование производительности и безопасности, формальные код-ревью, анализ сложности и пр.

Программная инженерия объединяет все эти дисциплины, методы и эвристики, представленные в этой книге. В других книгах [48; 55] обсуждаются более традиционные представления о программной инженерии, поэтому я затронул их лишь поверхностно.

Производительность важна, но вряд ли это определяющий фактор для ПО. Правильность его работы гораздо важнее, чем его скорость.

После разработки рабочего ПО можно подумать и о производительности. Но помните, что ваши ресурсы не безграничны.

Важнее всего, когда программа работает лучше или безопаснее? Что важнее: чтобы кодовая база была в состоянии, которое может поддерживать организацию в ближайшие годы, или чтобы она работала немного быстрее?

Как у разработчика, у вас может быть свое мнение по этому поводу, но это проблемные вопросы, для решения которых нужно привлечь и другие заинтересованные стороны.

16 КРАТКИЙ ОБЗОР

Я надеюсь, следование рекомендациям из этой книги повысит ваши шансы на создание простого и понятного для понимания кода, который будет помогать бизнес-процессам вашей организации. Как он будет выглядеть?

В последней главе я познакомлю вас с кодовой базой примеров, которая прилагается к книге, и укажу на моменты, которые, как мне кажется, особенно нуждаются во внимании.

16.1. НАВИГАЦИЯ

Если вы не писали код, как же вам найти свой путь сквозь него? Все зависит от вашей мотивации. Если вы программист, поддерживающий систему, и вас попросили исправить дефект с помощью прикрепленной трассировки стека, вы можете сразу перейти к верхнему фрейму трассировки.

Но если у вас нет определенной цели и вы просто хотите получить представление о приложении, то лучше всего начать с точки входа программы. В кодовой базе .NET это будет метод `Main`.

В целом, я считаю разумным, если читатель кода будет знаком с основными принципами работы используемого языка, платформы и фреймворка.

Для ясности: я не предполагаю, что вы, читатель книги, знакомы с .NET или ASP.NET, но, когда я программирую, я ожидаю, что участник команды знает основные правила. Например, что он знает особое значение метода `Main` в .NET.

В листинге 16.1 приведен пример метода `Main`, который встречался вам в кодовой базе. Он не изменился со времени написания кода для листинга 2.4.

Листинг 16.1. Точка входа в систему резервирования столиков. Листинг 16.1 идентичен листингу 2.4 (Restaurant/af31e63/Restaurant.RestApi/Program.cs)

```
public static class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

В кодовых базах ASP.NET Core метод `Main` — это шаблонная модель, которая редко меняется. Я полагаю, что другие программисты, которые столкнутся с этим кодом, будут знать основы фреймворка, поэтому считаю, что лучше написать код как можно менее неожиданным. Но в листинге 16.1 слишком мало информации.

Разработчики, поверхностно знакомые с ASP.NET, знают, что оператор `webBuilder.UseStartup<Startup>()` определяет класс `Startup` как место, где происходят реальные события. Вот что нужно начать анализировать, чтобы понять кодовую базу.

16.1.1. Общее представление

Для перехода к классу `Startup` используйте IDE. В листинге 16.2 приведен пример объявления класса и конструктор. Он использует внедрение через конструктор (constructor injection [25]) для получения объекта `IConfiguration` из среды ASP.NET. Это обычный способ сделать хоть что-то, и он должен быть знаком всем, кто работал с фреймворком. Хотя и неудивительно, что до сих пор было получено мало информации.

По соглашению класс `Startup` должен определять два метода: `Configure` и `ConfigureServices`, которые следуют сразу после листинга 16.2. В листинге 16.3 приведен пример метода `Configure`.

Листинг 16.2. Объявление класса Startup и конструктор (Restaurant/af31e63/Restaurant.RestApi/Startup.cs)

```
public sealed class Startup
{
    public IConfiguration Configuration { get; }

    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }
```

Листинг 16.3. Метод Configure в классе Startup (объявленный в листинге 16.2). (Restaurant/af31e63/Restaurant.RestApi/Startup.cs)

```
public static void Configure(
    IApplicationBuilder app,
    IWebHostEnvironment env)
{
    if (env.IsDevelopment())
        app.UseDeveloperExceptionPage();

    app.UseAuthentication();
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints => { endpoints.MapControllers(); });
}
```

Здесь мы узнаем, что система использует аутентификацию, маршрутизацию, авторизацию и стандартную реализацию фреймворка паттерна Model-View-Controller [33] (MVC). Уровень абстракции высокий, но код умещается в голове. Цикломатическая сложность равна 2, активируемых объектов всего 3, а в самом коде 12 строк. На рис. 16.1 приведен один из способов изобразить его на диаграмме гексагонального цветка. Рисунок показывает, как именно код вписывается в концептуальную модель фрактальной архитектуры.

Рис. 16.1. Диаграмма гексагонального цветка метода Configure из листинга 16.3. Ее можно заполнить несколькими способами. В отличие от примера из главы 7, где показано заполнение каждой ячейки ветвью в соответствии с анализом цикломатической сложности, он заполняет каждую ячейку активированным объектом

По сути, это просто подробный список насущных проблем. Все методы, вызываемые в листинге 16.3, — это методы фреймворка. Единственная цель метода `Configure` — включить эти конкретные встроенные функции. Читая его, вы немного понимаете, чего ожидать от кода. Например, что каждый HTTP-запрос будет обрабатываться методом класса `Controller`.

Возможно, из метода `ConfigureServices` в листинге 16.4 можно получить больше информации?

В коде листинга 16.4 есть немного больше информации, но она все еще находится на высоком уровне абстракции. Это все еще просто для понимания: цикломатическая сложность равна 1, есть шесть активированных объектов (`services`, `urlSigningKey`, новый объект `UrlIntegrityFilter`, две переменные `opts` и свойство объекта `Configuration`) и 21 строка кода. Опять же, вы можете изобразить этот метод в виде диаграммы гексагонального цветка (рис. 16.2), чтобы показать, насколько он соответствует концепции фрактальной архитектуры. Пока вы можете отобразить каждую часть метода в ячейке диаграммы, код, скорее всего, будет умещаться в голове.

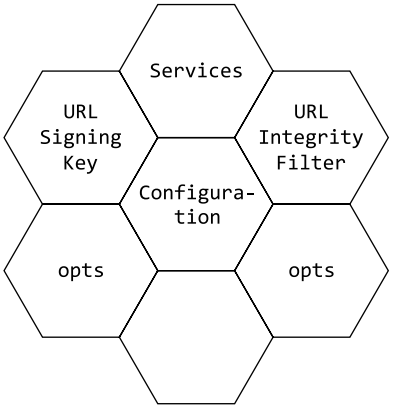


Рис. 16.2. Диаграмма гексагонального цветка метода ConfigureServices, приведенная в листинге 16.4. Как и на рис. 16.1, на этой диаграмме каждая ячейка заполнена активированным объектом

Листинг 16.4. Метод ConfigureServices в классе Startup (объявлен в листинге 16.2) (Restaurant/af31e63/Restaurant.RestApi/Startup.cs)

```
public void ConfigureServices(IServiceCollection services)
{
    var urlSigningKey = Encoding.ASCII.GetBytes(
        Configuration.GetValue<string>("UrlSigningKey"));

    services
        .AddControllers(opts =>
        {
```

```
            opts.Filters.Add<LinksFilter>();
            opts.Filters.Add(new UrlIntegrityFilter(urlSigningKey));
        })
        .AddJsonOptions(opts =>
            opts.JsonSerializerOptions.IgnoreNullValues = true);

    ConfigureUrSigning(services, urlSigningKey);
    ConfigureAuthorization(services);
    ConfigureRepository(services);
    ConfigureRestaurants(services);
    ConfigureClock(services);
    ConfigurePostOffice(services);
}
```

В этом методе мало деталей. Он больше похож на оглавление кодовой базы. Хотите узнать об авторизации? Перейдите к методу `ConfigureAuthorization`. Нужно изучить реализацию доступа к данным кодовой базы? Тогда к методу `ConfigureRepository`.

Переходя к дополнительным сведениям, вы увеличиваете масштаб. Это пример фрактальной архитектуры (см. подраздел 7.2.6), где код прост на каждом уровне. Для понимания кода при углублении в детализацию уровнем ниже более высокий уровень требоваться не должен.

Однако, прежде чем углубиться в детали, давайте обсудим, как перемещаться по кодовой базе.

16.1.2. Организация файлов

Меня часто спрашивают: «Как организовать файлы в кодовой базе?» Нужно ли создать один подкаталог для контроллеров, еще один для моделей, еще один для фильтров и т. д.? А может, стоит создать подкаталог для каждой функциональной возможности?

Мало кому нравится мой ответ: «Просто поместите все файлы в один каталог». Остерегайтесь создания подкаталогов только ради организации кода.

Файловые системы представлены в виде *иерархий*. Это деревья: особый вид ациклического графа, в котором любые две вершины

соединены ровно одним путем. Иначе говоря, у каждой вершины может быть не более одного родителя. Если еще проще: если вы поместите файл в гипотетический каталог `Controllers`, то не сможете поместить его и в каталог `Calendar`.

Как отмечается в анализе кодовой базы Firefox,

> *«системные архитекторы поняли, что для разделения системы на части есть несколько способов, что указывает на возможную сквозную функциональность, и что выбор одного разделения на модули приведет к разделению на модули других связных частей системы. Так, решение разделить компоненты браузера и инструментария Firefox привело к разделению компонентов мест и тем» [110].*

В этом проблема иерархии: любая попытка упорядочения автоматически исключает остальные способы организации. Та же проблема с иерархиями наследования в языках с одиночным наследованием, таких как C# и Java. Приняв решение унаследовать один базовый класс, вы исключаете все остальные классы как потенциальные базовые.

> *«Отдавайте предпочтение композиции объектов перед наследованием классов» [39].*

Точно так же, как вы должны избегать наследования, нужно воздерживаться от использования структуры каталогов для организации кода.

Но здесь, как и везде, есть исключения. Рассмотрим следующий пример кодовой базы. Каталог `Restaurant.RestApi` содержит 65 файлов кода: контроллеры, объекты передачи данных, доменные модели, фильтры, сценарии SQL, интерфейсы, адаптеры и т. д. Эти файлы реализуют различную функциональность, например резервирование и календарь, а также сквозную — как ведение логов.

Единственное исключение из этого правила — подкаталог `Options`. Его четыре файла существуют только для того, чтобы исключить раз-

рыв между файлами конфигурации на основе JSON и кодом. Классы в них предназначены для адаптации к системе параметров ASP.NET.

Это объекты передачи данных, которые существуют только для этой цели. Я совершенно уверен, что их нельзя использовать ни для чего другого, поэтому решил убрать их из поля зрения.

Когда я говорю специалистам, что организовывать файлы кода в сложные иерархии — плохая идея, они недоверчиво возражают: «Как мы тогда будем находить файлы?»

Используйте свою IDE. Она имеет функцию навигации. Когда ранее я писал, что вы должны использовать свою IDE для перехода к классу `Startup`, я не имел в виду: «Найдите файл `Startup.cs` в каталоге `Restaurant.RestApi` и откройте его».

Подразумевалось, что вы используете вашу IDE, чтобы перейти к определению символа. Например, в Visual Studio эта команда называется *Go To Definition* (Перейти к определению) и по умолчанию привязана к клавише `F12`. Другие команды позволяют перейти к реализаци интерфейсов, найти все ссылки или символ.

В вашем редакторе есть *вкладки*, и вы можете переключаться между ними с помощью стандартных сочетаний клавиш[1].

Я занимался моб-программированием, чтобы научить своих коллег разработке через тестирование. Мы анализировали тест, и я говорил что-то вроде: «Хорошо, мы можем переключиться на тестируемую систему?»

Затем драйвер находил имя этого класса, переходил к представлению, прокручивал его, чтобы найти файл, и дважды щелкал, чтобы открыть его.

Все это время файл был открыт в другой вкладке. Мы работали с ним три минуты назад, и это было просто сочетание клавиш.

---

[1] В Windows это будет сочетание клавиш Ctrl+Tab.

Попробуйте скрыть представление файлов в вашей среде IDE. Научитесь перемещаться по кодовым базам с помощью расширенной интеграции кода от IDE.

16.1.3. Поиск деталей

Метод как в листинге 16.4 позволяет вам получить общее представление, но иногда нужно увидеть детали реализации. Если, например, вы хотите узнать, как работает доступ к данным, перейдите к методу `ConfigureRepository` в листинге 16.5.

Листинг 16.5. Метод ConfigureRepository. Здесь вы можете узнать, как устроены компоненты доступа к данным (Restaurant/af31e63/Restaurant.RestApi/Startup.cs)

```
private void ConfigureRepository(IServiceCollection services)
{
    var connStr = Configuration.GetConnectionString("Restaurant");
    services.AddSingleton<IReservationsRepository>(sp =>
    {
        var logger =
            sp.GetService<ILogger<LoggingReservationsRepository>>();
        var postOffice = sp.GetService<IPostOffice>();
        return new EmailingReservationsRepository(
            postOffice,
            new LoggingReservationsRepository(
                logger,
                new SqlReservationsRepository(connStr)));
    });
}
```

Из метода `ConfigureRepository` вы можете узнать, что он регистрирует экземпляр `IReservationsRepository` во встроенном контейнере внедрения зависимостей. Код снова стал умещаться в голове: цикломатическая сложность равна 1, она активирует шесть объектов и имеет 15 строк кода. На рис. 16.3 с помощью диаграммы гексагонального цветка показано возможное отображение.

Поскольку вы углубились в детализацию, окружающий контекст не должен иметь значения. Что вам нужно отслеживать, так это параметр `services`, свойство `Configuration` и переменные, которые создает метод.

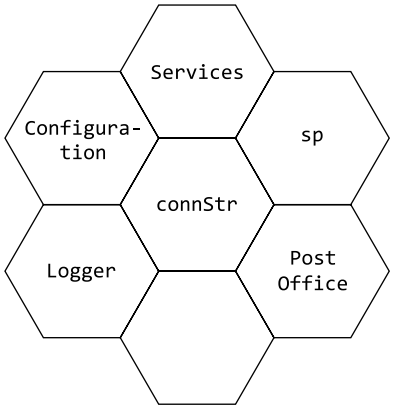


Рис. 16.3. Диаграмма гексагонального цветка метода ConfigureRepository из листинга 16.5. Как и на рис. 16.1, на этой диаграмме каждая ячейка заполнена активированным объектом

Из кода вы можете узнать несколько полезных моментов:

- чтобы отредактировать строку подключения приложения, нужно использовать стандартную систему конфигурации ASP.NET;
- служба `IReservationsRepository` — это декоратор с тремя уровнями глубины, который включает в себя логирование и отправку электронной почты;
- самая глубинная реализация — это класс `SqlReservationsRepository`.

В зависимости от того, что вас интересует, вы можете перейти к соответствующему типу. Чтобы узнать больше об интерфейсе `IPostOffice`, можете *перейти к определению* или *к реализации*. Чтобы взглянуть на `SqlReservationsRepository`, перейдите к нему. Делая это, вы погружаетесь в детализацию еще на уровень глубже.

Вы можете найти листинги кода из `SqlReservationsRepository` по всей книге, например листинги 4.19, 12.2 и 15.1. Как уже обсуждалось, все они умещаются в вашей голове.

Весь код в базе следует этим принципам.

16.2. АРХИТЕКТУРА

Я мало что мог сказать об архитектуре. Не то чтобы я считаю ее неважной, но на эту тему уже есть много хороших книг. Большинство представленных здесь практик работают с разными архитектурами: многоуровневой [33], монолитной, с портами и адаптерами [19], вертикальными срезами, моделью акторов, микросервисами, функциональным ядром в императивной оболочке [11] и т. д.

Очевидно, что архитектура ПО влияет на вашу организацию кода, так что вряд ли это неважно. Вы должны четко учитывать архитектуру для каждой кодовой базы, с которой работаете. Нет ни одной универсальной архитектуры, поэтому вы не должны рассматривать что-либо из того, что я скажу далее, как истину. Это описание одной архитектуры, которая хорошо работает для текущей поставленной задачи. Но оно подходит не для всех ситуаций.

16.2.1. Монолитная архитектура

Просматривая кодовую базу примеров из книги, вы могли заметить, что она выглядит монолитной. Если вы рассмотрите всю кодовую базу, включая интеграционные тесты, как на рис. 16.4, то увидите все три пакета[1], и только один из них — это продакшен-код.

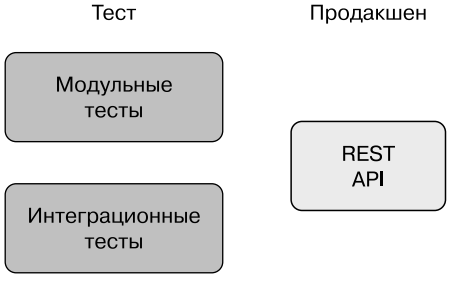


Рис. 16.4. Пакеты, из которых состоит кодовая база примеров. Только с одним продакшен-пакетом ее можно считать монолитной

[1] В Visual Studio это называется проектами.

Весь продакшен-код компилируется в один исполняемый файл. Этот процесс включает в себя доступ к БД, спецификацию HTTP, доменную модель, логирование, функциональность электронной почты, аутентификацию и авторизацию. Все в одном пакете, разве это не монолит?

В каком-то смысле да. Например, с точки зрения развертывания вы не можете разделить части, чтобы разместить их на разных машинах. Для нашего примера приложения я решил, что это не было бизнес-целью.

Вы также не можете повторно использовать фрагменты кода по-новому. Что, если бы мы захотели повторно использовать доменную модель для запуска запланированного задания пакетной обработки? Если вы попытаетесь сделать это, то обнаружите, что специфичный для HTTP код будет сопровождать вас, как и функциональность электронной почты.

Но это лишь артефакт того, как я решил упаковать код. Один пакет проще, чем, например, четыре.

Внутри этого единственного пакета я применил архитектуру *функционального ядра и императивной оболочки* [11], которая ведет к архитектуре в стиле портов и адаптеров [102].

Меня не очень беспокоит возможность разделения этой кодовой базы на несколько пакетов при необходимости.

16.2.2. Циклы

У монолитов плохая репутация — они быстро превращаются в спагетти-код. Основная причина в том, что внутри одного пакета весь код может легко[1] вызвать любой другой код.

Это часто приводит к зависимости одной части кода от другой, которая, в свою очередь, зависит от первой. Пример, который я часто

[1] Важно отметить, что в таком языке, как C#, вы можете использовать модификатор доступа private для предотвращения вызова метода другими классами. Для разработчика, который хочет срочно завершить задачу, это не помеха: просто измените модификатор доступа на internal и двигайтесь дальше.

встречаю, показан на рис. 16.5: интерфейсы доступа к данным, которые возвращают или принимают в качестве параметров объекты, определенные объектно-реляционным отображением. Интерфейс может быть определен как часть доменной модели кодовой базы, поэтому реализация связана с ней. Пока все хорошо, но интерфейс определяется в терминах классов объектно-реляционного отображения, поэтому абстракция будет зависеть и от деталей реализации. Это нарушает принцип инверсии зависимостей [60] и приводит к связанности.

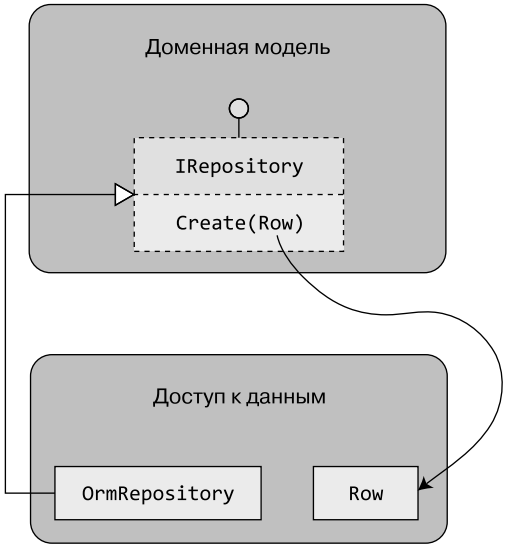


Рис. 16.5. Типичный цикл доступа к данным. Доменная модель определяет интерфейс доступа к данным IRepository. Члены определяются с возвращаемыми типами или параметрами, взятыми из уровня доступа к данным. Например, класс Row может быть определен с помощью объектно-реляционного отображения (ORM). Так, доменная модель будет зависеть от уровня доступа к данным. С другой стороны, класс OrmRepository — это основанная на ORM реализация интерфейса IRepository, которая не может реализовать интерфейс без ссылки на него. Поэтому уровень доступа к данным зависит и от доменной модели. Иначе говоря, зависимости образуют цикл

В таких случаях связанность проявляется в виде циклов. Как показано на рис. 16.6, объект A зависит от объекта B, который зависит от объекта C, который, в свою очередь, зависит от объекта A. Никакие основные языки не предотвращают циклы, поэтому вы должны быть предельно внимательны, чтобы избежать их.

Но есть лайфхак, который вы можете использовать. В то время как основные языки программирования допускают циклы в коде, они запрещают их в зависимостях пакетов. Если, например, вы пытаетесь определить интерфейс доступа к данным в пакете доменной модели и хотите использовать некоторые классы объектно-реляционного отображения для параметров или возвращаемых значений, вам придется добавить зависимость к вашему пакету доступа к данным.

На рис. 16.7 показано, что происходит дальше. Чтобы реализовать интерфейс в пакете доступа к данным, нужно добавить зависимость в пакет доменной модели. Но, так как ваша IDE отказывается нарушать принцип ациклической зависимости [60], вы не можете этого сделать.

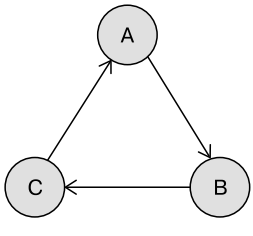


Рис. 16.6. Простой цикл. Объект A зависит от объекта B, который зависит от объекта C, который зависит от объекта A

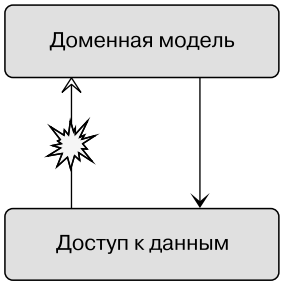


Рис. 16.7. Сорванный цикл. Если пакет доменной модели уже ссылается на пакет доступа к данным, последний не может ссылаться на пакет доменной модели. Вы не можете создать цикл зависимости между пакетами

Это должно мотивировать вас на разбиение кодовой базы на несколько пакетов. Вы получаете свою IDE для обеспечения соблюдения архитектурного принципа, даже если это только на уровне грубой детализации. Это модель Poka-Yoke применительно к архитектуре: она пассивно предотвращает крупномасштабные циклы.

Известный нам способ разделить систему на более мелкие компоненты: распределение поведения по доменной модели, доступу к данным, портам или пользовательскому интерфейсу и пакету Composition Root [25], чтобы составить остальные три вместе.

Как видно из рис. 16.8, вы можете модульно протестировать каждый пакет отдельно. Теперь вместо трех пакетов у вас семь.

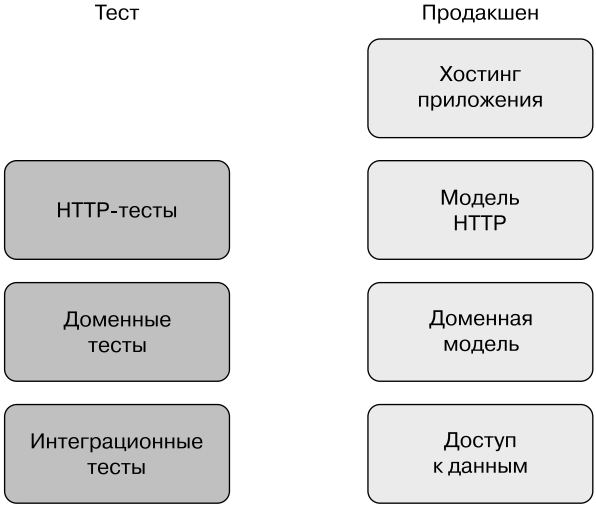


Рис. 16.8. Гипотетическая декомпозиция примера кодовой базы резервирования столиков. Модель HTTP будет содержать всю логику и конфигурацию, связанную с HTTP и REST, доменная модель — бизнес-логику, а пакет доступа к данным — код, взаимодействующий с БД. Пакет хостинга приложения будет содержать корень композиции [25], из которого состоят три других пакета. Они предназначены для трех продакшен-пакетов, содержащих сложную логику

Пассивное предотвращение циклов вносит дополнительную сложность. Если у участников команды нет большого опыта работы с языком, предотвращающим циклы, я рекомендую именно этот стиль архитектуры.

Такие языки есть. Например, F#. В нем вы не можете использовать фрагмент кода, если он уже не определен выше. Новички в языке программирования считают это ужасным недостатком, но на самом деле это одно из его самых больших преимуществ [117; 37].

Haskell использует другой подход, но в конечном счете его явная обработка побочных эффектов на уровне типов направляет вас к архитектуре в стиле портов и адаптеров. Иначе ваш код просто не скомпилируется [102]!

Я уже давно работаю с языками F# и Haskell и всегда следую их полезным функциям и правилам. Я уверен, что пример кода понятен, даже несмотря на то, что упакован как монолит. Но если у вас нет такого опыта, советую вам разделить кодовую базу на несколько пакетов.

16.3. ИСПОЛЬЗОВАНИЕ

Анализируя незнакомую кодовую базу, вы наверняка хотели бы увидеть ее в действии. У REST API нет пользовательского интерфейса, поэтому вы не можете просто запустить его и начать нажимать на кнопки.

Или в определенной степени можете. Если вы запустите приложение, то сможете просмотреть его домашний ресурс в браузере. Представления JSON, обслуживаемые API, содержат ссылки, по которым можно перейти. Но это ограниченный способ взаимодействия с системой.

Из своего браузера вы можете отправлять только запросы `GET`. Но, чтобы добавить новое резервирование, нужно будет сделать запрос `POST`.

16.3.1. Обучение на тестах

Если у кодовой базы есть исчерпывающий набор тестов, можно узнать о предполагаемом использовании именно из них. Например, вы можете разобраться, как сделать новое бронирование.

В листинге 16.6 приведен тест, который я написал, расширяя кодовую базу для многопользовательской системы.

Это все еще умещающийся в голове код: цикломатическая сложность равна 1, шесть активируемых объектов и 14 строк кода. Уровень абстракции высок в том смысле, что не сообщает подробностей о том, как он делает утверждения или как реализуется `PostReservation`.

Листинг 16.6. Модульный тест, выполняющий резервирование столика в ресторане Nono (Restaurant/af31e63/Restaurant.RestApi.Tests/ReservationsTests.cs)

```
[Fact]
public async Task ReserveTableAtNono()
{
    using var api = new SelfHostedApi();
    var client = api.CreateClient();
    var dto = Some.Reservation.ToDto();
    dto.Quantity = 6;

    var response = await client.PostReservation("Nono", dto);

    var at = Some.Reservation.At;
    await AssertRemainingCapacity(client, at, "Nono", 4);
    await AssertRemainingCapacity(client, at, "Hipgnosta", 10);
}
```

Если вам это интересно, можете перейти к реализации `PostReservation`, чтобы проанализировать код листинга 16.7.

Листинг 16.7. Метод Test Utility, выполняющий резервирование столика [66] (Restaurant/af31e63/Restaurant.RestApi.Tests/RestaurantApiClient.cs)

```
internal static async Task<HttpResponseMessage> PostReservation(
    this HttpClient client,
```

```
    string name,
    object reservation)
{
    string json = JsonSerializer.Serialize(reservation);
    using var content = new StringContent(json);
    content.Headers.ContentType.MediaType = "application/json";

    var resp = await client.GetRestaurant(name);
    resp.EnsureSuccessStatusCode();
    var rest = await resp.ParseJsonContent<RestaurantDto>();
    var address = rest.Links.FindAddress("urn:reservations");

    return await client.PostAsync(address, content);
}
```

Для взаимодействия с REST API метод Test Utility [66] использует `HttpClient`. Вы можете вспомнить из листинга 16.6, что рассматриваемый клиент взаимодействует с собственным экземпляром службы. Но при углублении в метод `PostReservation` вам больше не нужно отслеживать его. Единственное, что следует знать, — это то, что у вас есть работающий клиент.

Это еще один пример принципа работы фрактальной архитектуры: когда вы углубляетесь в детализацию, окружающий контекст становится неважным. Вам больше не нужно о нем думать.

В частности, вы можете видеть, что вспомогательный метод сериализует `reservation` в JSON, а затем находит подходящий адрес для отправки POST-запроса.

Теперь это выглядит более детально, чем раньше. Возможно, вы узнали, что хотели. Если вам интересно, как форматировать POST-запрос, какие HTTP-заголовки использовать и пр., вам не нужно продолжать поиск. Если же вы хотите знать, как перейти к определенному ресторану, придется углубиться в метод `GetRestaurant`. Или, если вы хотите узнать, как найти конкретный адрес в представлении JSON, можете углубиться в `FindAddress`.

Хорошо написанные тесты могут быть отличным учебным ресурсом.

16.3.2. Прислушивайтесь к своим тестам

Если бы у книги *Growing Object-Oriented Software, Guided by Tests* [36] был слоган, он бы звучал так: «Прислушивайтесь к своим тестам». Хорошие тесты могут научить вас большему, чем простое взаимодействие с тестируемой системой.

Помните, что тестовый код — это тоже код. Вам придется поддерживать его так же, как и продакшен. Вы должны провести рефакторинг тестового кода, когда он начинает деградировать, точно так же, как и в случае с продакшеном.

Можно добавить методы Test Utility [66], как в листинге 16.7 или 16.8. Оказывается, `GetRestaurant` в листинге 16.8 служит общей точкой входа для любого `HttpClient`, который хочет взаимодействовать с этим конкретным REST API. Поскольку это многопользовательская система, первый шаг любого клиента — переход к нужному ресторану.

Если вы внимательно посмотрите на листинги 16.7 и 16.8, то увидите, что в них нет ничего специфичного для тестов. Могут ли они быть полезны в других контекстах?

Листинг 16.8. Метод Test Utility [66], который находит ресурс ресторана по его названию (Restaurant/af31e63/Restaurant.RestApi.Tests/RestaurantApiClient.cs)

```
internal static async Task<HttpResponseMessage> GetRestaurant(
    this HttpClient client,
    string name)
{
    var homeResponse =
        await client.GetAsync(new Uri("", UriKind.Relative));
    homeResponse.EnsureSuccessStatusCode();
    var homeRepresentation =
        await homeResponse.ParseJsonContent<HomeDto>();
    var restaurant =
        homeRepresentation.Restaurants.First(r => r.Name == name);
    var address = restaurant.Links.FindAddress("urn:restaurant");

    return await client.GetAsync(address);
}
```

Преимущество REST API в том, что он поддерживает любой клиент, который знаком с HTTP и может анализировать JSON[1]. Но если вы только и делаете, что публикуете API, всем сторонним программистам придется разрабатывать свой клиентский код. Если значительная часть ваших клиентов находится на той же платформе, что и ваш тестовый код, вы можете повысить эти методы Test Utility до официального клиентского SDK.

Такое происходит со мной регулярно. Проводя рефакторинг тестового кода, я понимаю, что часть его может быть полезна и в качестве продакшен-кода. Это всегда приятное открытие. Если так произойдет, переместите код. Готово.

16.4. ЗАКЛЮЧЕНИЕ

«Настоящая» инженерия — это смесь детерминированных процессов и решений, которые принимает человек. Если вам нужно построить мост, у вас есть формулы для расчета несущей способности, но вам все равно нужно привлекать людей для решения бесчисленных сложностей, связанных с этой задачей. Какой трафик должна поддерживать конструкция? Какова желаемая пропускная способность? Каковы предельные значения температуры? Как устроен фундамент? Есть ли опасения по поводу окружающей среды?

Если бы проектирование было полностью детерминированным процессом, вам бы не понадобились человеческие ресурсы, ведь все, что для этого нужно, — компьютеры и промышленные роботы.

Вполне возможно, что некоторые инженерные дисциплины перейдут в эту область в будущем, но тогда этот процесс перестанет быть инженерией и станет производством.

Вы можете думать, что это пустые размышления, но я считаю, что это различие относится к искусству разработки ПО. Сколько бы ни было методологий в вашем арсенале, это не освобождает вас от обязанности использовать свой мозг.

---

[1] Или в данном случае XML.

Задача в том, чтобы объединить навыки с соответствующими процессами, эвристикой и технологиями для успешного процесса разработки. В этой книге я поделился несколькими техниками, которые вы можете использовать уже сегодня. Один из первых моих читателей сказал, что некоторые из предложенных здесь идей довольно *передовые*. Может и так, но реализовать их все еще *возможно*. Уильям Гибсон:

> *«Будущее уже наступило. Просто оно еще неравномерно распределено»*.

Методы из этой книги — это не что-то абстрактное. Некоторые организации уже используют их. Попробуйте и вы.

П ПЕРЕЧЕНЬ МЕТОДОВ

В этом приложении вы найдете перечень методов и эвристик из книги и ссылки на необходимые разделы.

П.1. ПРАВИЛО 50/72

Пишите простые сообщения коммитов Git:

- пишите заголовок в повелительном наклонении (не более 50 символов);
- при добавлении дополнительного текста оставляйте следующую строку пустой;
- можно добавить любое количество дополнительного текста, но желательно отформатировать его так, чтобы его объем был не более 72 символов.

После заголовка сосредоточьтесь и на объяснении *причины* внесения изменений, поскольку то, *какое* именно это изменение, уже видно через представление различий Git (`diff`). См. подраздел 9.1.1.

П.2. ПРАВИЛО 80/24

Пишите небольшие блоки кода.

В языках программирования, основанных на C, таких как C#, Java, C++ или JavaScript, постарайтесь всегда придерживаться размера кода 80 × 24 символа. Этот размер соответствует старому окну терминала.

Не воспринимайте пороговые значения 80 и 24 слишком буквально. Я выбрал их по трем причинам:

- они хорошо работают на практике;
- они отдают дань традициям;
- это правило подобно принципу Парето (принцип 80/20).

Вы можете выбрать другие пороговые значения. Я думаю, что самая важная часть этого правила — выбрать набор пороговых значений и постоянно оставаться в этих пределах.

Подробнее об этом — в подразделе 7.1.3.

П.3. ШАБЛОН ARRANGE-ACT-ASSERT (AAA)

Структурируйте автотесты в соответствии с шаблоном Arrange-Act-Assert. Подробную информацию вы можете найти в подразделах 4.2.2 и 4.3.3.

П.4. БИСЕКЦИЯ

Метод бисекции поможет вам понять причину возникновения неполадки. Удалите половину вашего кода и проверьте, остался ли дефект. Как минимум вы уже будете знать, где именно искать причину.

Продолжайте сокращать код, пока не уменьшите его до минимального рабочего примера. На этом этапе вы удалили столько неважного кода, что причина проблемы наверняка станет ясна. Подробнее об этом — в разделе 12.3.

П.5. ЧЕК-ЛИСТ ДЛЯ НОВОЙ КОДОВОЙ БАЗЫ

При создании новой кодовой базы или добавлении нового проекта в существующую составьте чек-лист, например:

- использовать Git;
- автоматизировать сборку;
- включить все сообщения об ошибках.

Можете изменить чек-лист, чтобы он соответствовал вашему конкретному контексту. Но он должен оставаться кратким и простым. Подробную информацию вы можете найти в разделе 2.2.

П.6. РАЗДЕЛЕНИЕ КОМАНД И ЗАПРОСОВ (CQS)

Отделяйте команды от запросов. Команды — это процедуры, имеющие побочные эффекты, а запросы — функции, которые возвращают данные. Каждый метод должен быть либо командой, либо запросом, но не тем и другим одновременно. Подробнее об этом — в подразделе 8.1.6.

П.7. ПОДСЧЕТ ПЕРЕМЕННЫХ

Подсчитывайте все переменные, участвующие в реализации метода. Включая локальные переменные, параметры метода и поля класса. Следите за тем, чтобы это число было небольшим. Подробную информацию вы можете найти в подразделе 7.2.7.

П.8. ЦИКЛОМАТИЧЕСКАЯ СЛОЖНОСТЬ

Цикломатическая сложность — одна из немногих действительно полезных метрик кода. С ее помощью можно подсчитать количество путей прохождения через фрагмент кода, тем самым получив представление о сложности метода.

Я считаю, что пороговое значение 7 хорошо работает на практике. Вы можете выполнять полезную работу с цикломатической сложностью, равной 7. Такой порог достаточно велик, чтобы не приходилось все время проводить рефакторинг, но все еще достаточно низок, чтобы метод умещался в вашей голове. Подробнее об этом — в подразделе 7.1.2.

Еще метрика предоставляет минимальное количество тестовых случаев, чтобы полностью охватить метод.

П.9. ПАТТЕРН ПРОЕКТИРОВАНИЯ DECORATOR ДЛЯ СКВОЗНОЙ ФУНКЦИОНАЛЬНОСТИ

Не внедряйте зависимости логирования в свою бизнес-логику. Это не разделяет функциональность, а перемешивает все вместе. Это же касается кэширования, отказоустойчивости и большинства других сквозных задач.

Вместо этого используйте паттерн декоратора, описанный в разделе 13.2.

П.10. МЕТОД «АДВОКАТ ДЬЯВОЛА»

Метод «Адвокат дьявола» — это эвристика, которую вы можете использовать, чтобы оценить, укрепит ли большее количество тестов

уверенность в тестовом наборе. Вы можете применять его для просмотра существующего (тестового) кода или использовать его в качестве вдохновения для новых тестовых случаев, которые нужно добавить.

Суть в том, чтобы преднамеренно неправильно реализовать тестируемую систему. Чем более неверным вы можете сделать ответ, тем больше тестовых случаев вы должны добавить. Подробнее об этом — в подразделе 6.2.2.

П.11. ФУНКЦИОНАЛЬНЫЙ ФЛАГ

Если вы не можете внести последовательный набор изменений за полдня, скройте функцию за флагом и продолжайте интегрировать свои изменения с работой других специалистов.

Подробную информацию вы можете найти в разделе 10.1.

П.12. ФУНКЦИОНАЛЬНОЕ ЯДРО, ИМПЕРАТИВНАЯ ОБОЛОЧКА

Отдайте предпочтение чистым функциям.

Ссылочная прозрачность означает, что вы можете заменить вызов функции ее результатом без изменения поведения программы. Это предельная абстракция.

Вывод инкапсулирует суть функции, в то время как все детали реализации остаются скрытыми (если они вам не нужны).

Чистые функции хорошо компонуются, и их легко модульно тестировать.

Подробную информацию вы можете найти в подразделе 13.1.3.

П.13. ИЕРАРХИЯ ОТНОШЕНИЙ

Пишите код для разработчиков, которые будут иметь с ним дело. Когда-нибудь одним из таких можете оказаться и вы. Отдавайте предпочтение коммуникативному поведению и намерениям в соответствии с этим списком приоритетов.

1. Указывайте разные типы API.
2. Присваивайте методам полезные и понятные имена.
3. Пишите полезные комментарии.
4. Предоставляйте наглядные примеры в виде автотестов.
5. Пишите полезные сообщения коммитов в Git.
6. Создавайте полезную и понятную документацию.

Правила приведены в порядке убывания приоритета. Подробнее об этом — в подразделе 8.1.7.

П.14. ОБОСНОВАНИЕ ИСКЛЮЧЕНИЙ ИЗ ПРАВИЛ

Хорошие правила работают в большинстве случаев, но всегда есть исключения, когда правило только мешает. При необходимости можно отклониться от правила, но лишь обосновав и задокументировав причину. Подробную информацию вы можете найти в подразделе 4.2.3.

Это отличная возможность получить еще одно мнение, прежде чем вы решите отклониться от правила. Иногда вы можете не обнаружить подходящего способа получить желаемое *и* следовать правилу, но ваш коллега может.

П.15. АНАЛИЗИРОВАТЬ, А НЕ ПРОВЕРЯТЬ

Ваш код взаимодействует со остальным миром, а он не является объектно-ориентированным. Вместо этого вы получаете данные в виде

JSON, XML, значений, разделенных запятыми, буферов протоколов или в других форматах, которые дают мало гарантий относительно целостности данных.

По возможности всегда старайтесь преобразовывать менее структурированные данные в более структурированные. Вы можете думать об этом как о *парсинге*, даже если вы не анализируете обычный текст. Подробнее об этом — в подразделе 7.2.5.

П.16. ЗАКОН ПОСТЕЛА

Помните о законе Постела для пред- и постусловий.

> *«Будьте консервативны в том, что отправляете, и либеральны в том, что принимаете».*

Методы должны принимать входные данные, до тех пор пока они могут их осмыслить, но не более того. Возвращаемые значения должны быть максимально достоверными. Подробную информацию вы можете найти в подразделе 5.2.4.

П.17. ЦИКЛ «КРАСНЫЙ, ЗЕЛЕНЫЙ, РЕФАКТОРИНГ»

При разработке через тестирование следуйте циклу «красный, зеленый, рефакторинг». Предполагайте, что это чек-лист [93].

1. Напишите провальный тест.
   - Вы провели тест?
   - Провалился ли он?
   - Провалился из-за утверждения?
   - Провалился из-за *последнего* утверждения?

2. Сделайте все тесты успешными, выполнив самые простые изменения, которые только смогут работать.
3. Проанализируйте итоговый код. Можно ли его улучшить? Если да, сделайте это, но убедитесь, что тесты все еще проходят.
4. Повторите.

Подробнее об этом — в подразделе 5.2.2.

П.18. РЕГУЛЯРНОЕ ОБНОВЛЕНИЕ ЗАВИСИМОСТЕЙ

Не позволяйте вашей кодовой базе отставать от зависимостей. Регулярно проверяйте наличие обновлений. Об этом легко забыть, и если пройдет слишком много времени с момента последнего обновления, вам может быть трудно наверстать упущенное. Подробнее об этом — в подразделе 14.2.1.

П.19. ВОСПРОИЗВЕДЕНИЕ ДЕФЕКТОВ В ВИДЕ ТЕСТОВ

По возможности воспроизведите ошибки в виде одного или нескольких автоматизированных тестов. Подробную информацию вы можете найти в подразделе 12.2.1.

П.20. КОД-РЕВЬЮ

При написании кода легко ошибиться. Попросите коллегу выполнить код-ревью. Оно не фиксирует все ошибки, но это один из самых эффективных методов обеспечения качества.

Вы можете выполнять код-ревью по-разному: постоянно, при парном/моб-программировании или асинхронно через пул-реквесты.

При каждом проведении код-ревью *отказ* должен быть потенциальной опцией. Ревью ничего не стоит, если является лишь формальностью и проводится для галочки.

Введите код-ревью в свою ежедневную рутину. Подробнее об этом — в разделе 9.2.

П.21. СЕМАНТИЧЕСКОЕ ВЕРСИОНИРОВАНИЕ

Рассмотрите возможность использования семантического версионирования. Подробную информацию вы можете найти в разделе 10.3.

П.22. РАЗДЕЛЬНЫЙ РЕФАКТОРИНГ ТЕСТОВОГО И ПРОДАКШЕН-КОДА

Автоматизированные тесты дают вам уверенность при рефакторинге продакшен-кода. Рефакторинг же тестового кода менее надежен из-за отсутствия у вас автотестов для тестов.

Это не значит, что вы вообще не можете выполнить рефакторинг тестового кода, но вы должны быть при этом предельно внимательны. Не проводите рефакторинг тестового и продакшен-кода одновременно.

При рефакторинге продакшен-кода не трогайте тестовый, и наоборот. Подробную информацию вы можете найти в подразделе 11.1.3.

П.23. СРЕЗ

Всегда разделяйте большие задачи на более мелкие. Каждое приращение должно улучшать рабочую систему. Начните с вертикального среза и добавьте к нему функциональность. Подробнее об этом — в главе 4.

Не рассматривайте этот процесс как исключительный. Я считаю, что это мой основной процесс продвижения вперед, но иногда нужно остановиться и заняться другими делами. Например, исправлением ошибок или работой над сквозной функциональностью.

П.24. ПАТТЕРН STRANGLER

Некоторые рефакторинги выполняются быстро. Переименование переменной, метода или класса встроено в большинство IDE и выполняется одним нажатием кнопки. На некоторые изменения уходит несколько минут или часов. Пока вы можете перейти от одного согласованного состояния кодовой базы к другому менее чем за полдня, вам не понадобится делать ничего особенного.

Но есть изменения, которые имеют большое потенциальное влияние. Я проводил рефакторинги, реализация которых занимала дни или даже недели. Это не лучший способ работы.

Для внедрения таких изменений используйте паттерн Strangler. Установите новый способ работы параллельно со старым и постепенно переносите код со старого на новый.

Это может занять часы, дни или даже недели, но в процессе миграции система всегда будет оставаться согласованной и интегрируемой. Когда никакой код не вызывает исходный API, его можно удалить.

Подробнее об этом — в разделе 10.2.

П.25. МОДЕЛЬ УГРОЗ STRIDE

Принимайте осознанные решения в области безопасности.

Модель STRIDE достаточно проста в понимании для неэкспертов в области безопасности, поэтому вы можете легко с ней работать.

- Спуфинг (spoofing).
- Незаконное изменение (tampering).

- Отказ от авторства (repudiation).
- Раскрытие информации (information disclosure).
- Отказ в обслуживании (denial of service).
- Повышение привилегий (elevation of privilege).

При моделировании угроз должны участвовать ИТ-специалисты и другие заинтересованные стороны, поскольку надлежащее смягчение последствий обычно включает взвешивание бизнес-проблем и рисков безопасности.

Подробную информацию вы можете найти в подразделе 15.2.1.

П.26. ПРЕДПОСЫЛКИ ПРИОРИТЕТА ТРАНСФОРМАЦИИ (TPP)

Старайтесь работать так, чтобы ваш код бóльшую часть времени находился в рабочем состоянии.

Преобразование одного допустимого состояния в другое обычно включает фазу, когда код недействителен, например когда он может не компилироваться.

TPP предлагают ряд небольших преобразований, которые минимизируют недопустимые этапы. Попытайтесь отредактировать свой код, внеся ряд этих небольших изменений. Подробнее об этом — в подразделе 5.1.1.

П.27. X-ОРИЕНТИРОВАННАЯ РАЗРАБОТКА

Используйте *драйвер* для кода, который пишете. Это может быть статический анализ кода, модульное тестирование, встроенные инструменты рефакторинга и т. д. Подробную информацию вы можете найти в разделе 4.2.

Можно отклоняться от этого правила, но чем чаще вы его придерживаетесь, тем меньше будет вероятность сбиться с верного пути.

П.28. ИСКЛЮЧЕНИЕ ИМЕН

Замените имена методов символами x, чтобы проверить количество передаваемой сигнатурой метода информации. Можно сделать это в уме. Это не обязательно делать в вашем редакторе, так как в статически типизированном языке типы могут нести очень много информации. Подробнее об этом — в подразделе 8.1.5.

БИБЛИОГРАФИЯ

1. *Adzic G.* The Poka-Yoke principle and how to write better software. Пост в блоге: https://gojko.net/2007/05/09/the-poka-yoke-principle-and-how-towrite-better-software, 2007.
2. *Allamaraju S.* RESTfulWeb Services Cookbook. O'Reilly, 2010.
3. *Atwood J.* New Programming Jargon. Пост в блоге: https://blog.codinghorror.com/new-programming-jargon, 2012.
4. *Barr A.* The Problem with Software.Why Smart EngineersWrite Bad Code. MIT Press, 2018.
5. *Бек Кент.* Экстремальное программирование. Разработка через тестирование. — СПб.: Питер, 2022. — 224 с.
6. *Beck K.* Твит: https://twitter.com/KentBeck/status/2507333-58307500032, 2012.
7. *Beck K.* Implementation Patterns. Addison-Wesley, 2007.
8. *Beck K.* Naming From the Outside In. Публикация по адресу https://www.facebook.com/notes/kent-beck/naming-from-the-outside-in/464270190272517 (доступна без регистрации), 2012.
9. *Beck K.* Test-Driven Development By Example. Addison-Wesley, 2002.
10. *Beck K.* Твит: https://twitter.com/KentBeck/status/1354418068869398538, 2021.
11. *Bernhardt G.* Functional Core, Imperative Shell. Онлайн-презентация по адресу https://www.destroyallsoftware.com/screencasts/catalog/functional-coreimperative-shell, 2012.
12. *Böckeler B., Siessegger N.* On Pair Programming. Пост в блоге: https://martinfowler.com/articles/on-pair-programming.html, 2020.

13. *Bossavit L.* The Leprechauns of Software Engineering. Laurent Bossavit, 2015.
14. *Brooks F. P., Jr.* No Silver Bullet — Essence and Accident in Software Engineering, 1986. Это эссе можно найти в разных источниках, в том числе в интернете. При написании этой книги я ссылаюсь на издание: *Брукс Ф.* Мифический человеко-месяц, или Как создаются программные системы. — Питер, 2022. Эссе — в главе 16.
15. *Brown W. J., Malveau R. C., McCormick III H. W. "Skip", Mowbray T. J.* AntiPatterns: Refactoring Software, Architectures and Projects in Crisis. Wiley Computer Publishing, 1998.
16. *Кейн С.* Тихая сила. Как достичь успеха если не любишь быть в центре внимания. — М.: Манн, Иванов и Фербер, 2022.
17. *Campidoglio E.* Твит: https://twitter.com/ecampidoglio/status/1194597766128963584, 2019.
18. *Чирилло Ф.* Метод Помидора. Управление временем, вдохновением и концентрацией. — М.: Бомбора, 2020.
19. *Cockburn A.* Hexagonal architecture. Статья по адресу https://alistair.cockburn.us/hexagonal-architecture/, 2005.
20. *Cohen J.* Modern Code Review в [75], 2010.
21. *Conway M. E.* How Do Committees Invent? Datamation, 1968. Признаюсь, у меня нет экземпляра журнала Datamation за апрель 1968 года. Вместо этого я использовал онлайн-переиздание, которое Мелвин Конвей размещает на сайте http://www.melconway.com/Home/Committees_Paper.html.
22. *Cunningham W., Venners B.* The Simplest Thing that Could Possibly Work. A Conversation with Ward Cunningham, Part V. Интервью на сайте: www.artima.com/intv/simplest.html, 2004.
23. *Цвалина К., Брэд А.* Непрерывное развертывание ПО. Инфраструктура программных проектов. Соглашения, идиомы и шаблоны. — М.: Вильямс, 2011.
24. *DeLine R.* Code Talkers в [75], 2010.
25. *Deursen S. van, Seemann M.* Dependency Injection Principles, Practices and Patterns. Manning, 2019.

26. *Эванс Э.* Непрерывное развертывание ПО. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем. — М.: Вильямс, 2018.
27. *Физерс М.* Ускоряйся! Наука DevOps. Эффективная работа с унаследованным кодом. — М.: Вильямс, 2017.
28. *Foote B., Yoder J.* The Selfish Class / B [62], 1998.
29. *Форсген Н., Хамбл Дж., Ким Дж.* Ускоряйся! Наука DevOps. Как создавать и масштабировать высокопроизводительные цифровые организации. — М.: Альпина Паблишер, 2020.
30. *Fowler M.* CodeOwnership. Пост в блоге: https://martinfowler.com/bliki/CodeOwnership.html, 2006.
31. *Fowler M.* Eradicating Non-Determinism in Tests. Пост в блоге: https://martinfowler.com/articles/nonDeterminism.html, 2011.
32. *Fowler M.* Is High Quality Software Worth the Cost? Пост в блоге: https://martinfowler.com/articles/is-quality-worth-cost.html, 2019.
33. *Фаулер М.* Шаблоны корпоративных приложений. — М.: Диалектика, 2018.
34. *Бек К., Брант Дж., Фаулер М.* Рефакторинг. Улучшение проекта существующего кода. — М.: Диалектика, 2017.
35. *Fowler M.* StranglerFigApplication. Пост в блоге: https://martinfowler.com/bliki/StranglerFigApplication.html, 2004.
36. *Freeman S., Pryce N.* Growing Object-Oriented Software, Guided by Tests. Addison-Wesley, 2009.
37. *Gabasova E.* Comparing F# and C# with dependency networks. Пост в блоге: http://evelinag.com/blog/2014/06-09-comparing-dependency-networks, 2014.
38. *Gabriel R. P.* Patterns of Software. Tales from the Software Community. Oxford University Press, 1996.
39. *Гамма Э., Джонсон Р., Хелм Р.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2021.
40. *Гаванде А.* Чек-лист. Как избежать глупых ошибок, ведущих к фатальным последствиям. — М.: Альпина Паблишер, 2014.

41. *Haack P.* I Knew How To Validate An Email Address Until I Read The RFC. Пост в блоге: https://haacked.com/archive/2007/08/21/i-knew-how-to-validatean-email-address-until-i.aspx, 2007.
42. *Henney K.* Твит: https://twitter.com/KevlinHenney/status/3361631527, 2009.
43. *Herraiz I., Hassan A. E.* Beyond Lines of Code: Do We Need More Complexity Metrics? / В [75], 2010.
44. *Herzig K. S., Zeller A.* Mining Your Own Evidence / В [75], 2010.
45. *Hickey R.* Simple Made Easy, Strange Loop conference talk, 2011. Запись доступна по адресу: https://www.infoq.com/presentations/Simple-Made-Easy.
46. *Хоп Г., Вульф Б.* Шаблоны интеграции корпоративных приложений. Проектирование, создание и развертывание решений. — М.: Диалектика, 2019.
47. *House C.* Твит: https://twitter.com/housecor/status/1115959687332159490, 2019.
48. *Ховард М., Лебланк Д.* Защищенный код для Windows Vista. — СПб.: Питер, 2008.
49. *Хамбл Дж., Фарли Д.* Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий. — М.: Вильямс, 2017.
50. *Хант Э., Томас Д.* Программист-прагматик. — М.: Диалектика, 2020.
51. *Канеман Д.* Думай медленно... Решай быстро. — М.: АСТ, 2013.
52. *Kay A., Binstock A.* Interview with Alan Kay, Dr. Dobb's, www.dr-dobbs.com/architecture-and-design/interview-with-alan-kay/240003442, July 10, 2012.
53. *Кериевски Дж.* Рефакторинг с использованием шаблонов. — М.: Диалектика, 2019.
54. *King A.* Parse, don't validate. Пост в блоге: https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate, 2019.
55. *Клеппман М.* Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2018.

56. *Lanza M., Marinescu R.* Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate and Improve the Design of Object-Oriented Systems. Springer, 2006.

57. *Левитт С., Дабнер С.* Фрикономика: Экономист-хулиган и журналист-сорвиголова исследуют скрытые причины всего на свете. — М.: Альпина Паблишер, 2018.

58. *Левитт С., Дабнер С.* Суперфрикономика. — М.: Эксмо, 2010.

59. *Lippert E.* Which is faster? Пост в блоге: https://ericlippert.com/2012/12/17/performance-rant, 2012.

60. *Мартин Р., Мартин М.* Принципы, паттерны и методики гибкой разработки на языке C#. — М.: Символ-Плюс, 2011.

61. *Мартин Р.* Чистый код: создание, анализ и рефакторинг. — СПб.: Питер, 2018.

62. *Martin R. C., Riehle D., Buschmann F.* (editors). Pattern Languages of Program Design 3. Addison-Wesley, 1998.

63. *Martin R. C.* The Sensitivity Problem. Пост в блоге: http://butunclebob.com/ArticleS.UncleBob.TheSensitivityProblem, 2005.

64. *Martin R. C.* The Transformation Priority Premise. Пост в блоге: https://blog.cleancoder.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html, 2013.

65. *Макконнелл С.* Совершенный код. — М.: БХВ, 2022.

66. *Месарош Дж.* Шаблоны тестирования xUnit. — М.: Диалектика; Вильямс, 2016.

67. *Мейер Б.* Объектно-ориентированное конструирование программных систем. — М.: Русская редакция, 2005.

68. *Milewski B.* Category Theory for Programmers. Первоначально серия публикаций в блоге: https://bartoszmilewski.com/2014/10/28/category-theory-forprogrammers-the-preface, 2014–2017. Доступна и в виде печатной книги, Blurb, 2019.

69. *Minsky Y.* Effective ML, запись лекции, прочитанной в Гарварде. Запись доступна на YouTube: https://youtu.be/-J8YyfrSwTk, но вместо нее вы можете посетить веб-страницу Ярона Мински,

которая содержит немного контекста: https://blog.janestreet.com/effective-ml-video, 2010.

70. *Neward T.* The Vietnam of Computer Science. Пост в блоге: http://blogs.tedneward.com/post/the-vietnam-of-computer-science, 2006.
71. *Норман Д.* Дизайн привычных вещей. — М.: Манн, Иванов и Фербер, 2021.
72. *North D.* Patterns of Effective Delivery, Roots opening keynote, 2011. Запись доступна по адресу https://vimeo.com/24681032.
73. *Nygard M. T.* Release It! Design and Deploy Production-Ready Software. Pragmatic Bookshelf, 2007.
74. *Nygard M. T.* DevOps: Tempo, Maneuverability and Initiative, DevOps Enterprise Summit conference talk, 2016. Запись доступна по адресу https://youtu.be/0rRWvsb8JOo.
75. Идеальная разработка ПО. Рецепты лучших программистов / под ред. Э. Орама, Г. Уилсона — СПб.: Питер, 2012.
76. *O'Toole G.* The Future Has Arrived — It's Just Not Evenly Distributed Yet. Статья по адресу https://quoteinvestigator.com/2012/01/24/future-hasarrived, 2012.
77. *Ottinger T.* Code is a Liability, 2007. Первоначально сообщение в блоге, но домен был передан другой организации. Сообщение в блоге по-прежнему доступно в интернет-архиве: http://web.archive.org/web/20070420113817/http://blog.objectmentor.com/articles/2007/04/16/code-is-aliability.
78. *Ottinger T.* What's this about Micro-commits? Пост в блоге: https://www.industriallogic.com/blog/whats-this-about-micro-commits, 2021.
79. *Peters T.* The Zen of Python, 1999. Первоначально это сообщение из списка рассылки, оно уже давно доступно по адресу https://www.python.org/dev/peps/pep-0020.
80. *Pinker S.* How the Mind Works, The Folio Society, 2013. Мое издание Folio Society, которое, согласно аннотации, «следует тексту издания Penguin 1998 года с незначительными изменениями».
81. *Pope T.* A Note About Git Commit Messages. Пост в блоге: https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html, 2008.

82. *Poppendieck M., Poppendieck T.* Implementing Lean Software Development: From Concept to Cash. Addison-Wesley, 2006.

83. *Preston-Werner T.* Semantic Versioning. Спецификация по адресу https://semver.org. В корневом каталоге сайта отображается последняя версия. На момент написания книги в октябре 2020 года последней версией была Semantic Versioning 2.0.0, выпущенная в 2013 году.

84. *Pyhäjärvi M.* Five Years of Mob Testing, Hello to Ensemble Testing. Пост в блоге: https://visible-quality.blogspot.com/2020/05/five-years-of-mob-testinghello-to.html, 2020.

85. *Rainsberger J. B.* Integration Tests Are a Scam, Agile 2009 conference talk, 2009. Запись доступна по адресу https://www.infoq.com/presentations/integration-tests-scam.

86. *Rainsberger J. B.* Твит: https://twitter.com/jbrains/status/167297606698008576, 2012.

87. *Reeves J.* What Is Software Design? // C++ Journal, 1992. Если у вас, как и у меня, нет журнала C++ Journal, вы можете найти статью по адресу https://www.developerdotstar.com/mag/articles/reeves_design.html (многие годы находится в общем доступе). Она есть и в виде приложения в [60].

88. *Рис Э.* Бизнес с нуля: Метод Lean Startup для быстрого тестирования идей и выбора бизнес-модели. — М.: Альпина Паблишер, 2022.

89. *Робинсон Я., Эифрем Э., Вебер Дж.* Графовые базы данных. Новые возможности для работы. — М.: ДМК-Пресс, 2016.

90. *Scott J. C.* Seeing Like a State: How Certain Schemes to Improve the Human Condition Have Failed. Yale University Press, 1998.

91. *Seemann M.* 10 tips for better Pull Requests. Пост в блоге: https://blog.ploeh.dk/2015/01/15/10-tips-for-better-pull-requests, 2015.

92. *Seemann M.* A heuristic for formatting code according to the AAA pattern. Пост в блоге: https://blog.ploeh.dk/2013/06/24/a-heuristic-for-formatting-codeaccording-to-the-aaa-pattern, 2013.

93. *Seemann M.* A red-green-refactor checklist. Пост в блоге: https://blog.ploeh.dk/2019/10/21/a-red-green-refactor-checklist, 2019.

94. *Seemann M.* Church-encoded Maybe. Пост в блоге: https://blog.ploeh.dk/2018/06/04/church-encoded-maybe, 2018.
95. *Seemann M.* CQS versus server generated Ids. Пост в блоге: https://blog.ploeh.dk/2014/08/11/cqs-versus-server-generated-ids, 2014.
96. *Seemann M.* Conway's Law: latency versus throughput. Пост в блоге: https://blog.ploeh.dk/2020/03/16/conways-law-latency-versus-throughput, 2020.
97. *Seemann M.* Curb code rot with thresholds. Пост в блоге: https://blog.ploeh.dk/2020/04/13/curb-code-rot-with-thresholds, 2020.
98. *Seemann M.* Devil's advocate. Пост в блоге: https://blog.ploeh.dk/2019/10/07/devils-advocate, 2019.
99. *Seemann M.* Feedback mechanisms and tradeoffs. Пост в блоге: https://blog.ploeh.dk/2011/04/29/Feedbackmechanismsandtradeoffs, 2011.
100. *Seemann M.* From interaction-based to state-based testing. Пост в блоге: https://blog.ploeh.dk/2019/02/18/from-interaction-based-to-state-basedtesting, 2019.
101. *Seemann M.* Fortunately, I don't squash my commits. Пост в блоге: https://blog.ploeh.dk/2020/10/05/fortunately-i-dont-squash-my-commits, 2020.
102. *Seemann M.* Functional architecture is Ports and Adapters. Пост в блоге: https://blog.ploeh.dk/2016/03/18/functional-architecture-is-ports-and-adapters, 2016.
103. *Seemann M.* Repeatable execution. Пост в блоге: https://blog.ploeh.dk/2020/03/23/repeatable-execution, 2020.
104. *Seemann M.* Structural equality for better tests. Пост в блоге: https://blog.ploeh.dk/2021/05/03/structural-equality-for-better-tests, 2021.
105. *Seemann M.* Tautological assertion. Пост в блоге: https://blog.ploeh.dk/2019/10/14/tautological-assertion, 2019.
106. *Seemann M.* Towards better abstractions. Пост в блоге: https://blog.ploeh.dk/2010/12/03/Towardsbetterabstractions, 2010.
107. *Seemann M.* Visitor as a sum type. Пост в блоге: https://blog.ploeh.dk/2018/06/25/visitor-as-a-sum-type, 2018.

108. *Seemann M.* When properties are easier than examples. Пост в блоге: https://blog.ploeh.dk/2021/02/15/when-properties-are-easier-than-examples, 2021.

109. *Шоу Дж.* Ложная память. Почему нельзя доверять воспоминаниям. — М.: Колибри, 2017.

110. *Thomas N., Murphy G.* How Effective Is Modularization? / В [75], 2010.

111. *Tornhill A.* Your Code as a Crime Scene: Use Forensic Techniques to Arrest Defects, Bottlenecks and Bad Design in Your Programs. Pragmatic Bookshelf, 2015.

112. *Tornhill A.* Software Design X-Rays: Fix Technical Debt with Behavioral Code Analysis. Pragmatic Bookshelf, 2018.

113. *Troy C.* Reviewing Pull Requests. Пост в блоге: https://chelseatroy.com/2019/12/18/reviewing-pull-requests, 2019.

114. *Webber J.* Savas Parastatidis and Ian Robinson. REST in Practice: Hypermedia and Systems Architecture. O'Reilly, 2010.

115. *Weinberg G. M.* The psychology of computer programming. Silver anniversary edition. Dorset House Publishing, 1998.

116. *Williams L.* Pair Programming в [75], 2010.

117. *Wlaschin S.* Cycles and modularity in the wild. Пост в блоге: https://fsharpforfunandprofit.com/posts/cycles-and-modularity-in-the-wild, 2013.

118. *Woolf B.* Null Object / В [62], 1997.

Марк Симан

Роберт Мартин рекомендует. Код, который умещается в голове: эвристики для разработчиков

Перевел с английского С.Черников

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Т. Сажина*
Художественный редактор	*В. Мостипан*
Корректоры	*М. Молчанова, Е. Павлович*
Верстка	*Г. Блинов*

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: 06.2023. Наименование: книжная продукция.
Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции
ОК 034-2014, 58.11.12 — Книги печатные
профессиональные, технические и научные.

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 19.04.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 32,250. Тираж 1200. Заказ 0000.