Оглавление

Содержание

Предисловие

В современной разработке программного обеспечения тестирование управляет проектированием, реализацией, развитием, обеспечением качества и развертыванием программных систем. Чтобы быть эффективным разработчиком, вы должны стать успешным тестировщиком программного обеспечения, и эта книга поможет вам в этом.

Тестирование – это не что иное, как выполнение части программного обеспечения с целью увидеть, соответствует ли его поведение ожидаемому. Но тестирование – непростая задача. Его сложность становится очевидной, стоит только подумать о полном наборе тестов, которые необходимо спроектировать и выполнить. Какие из бесконечного множества возможных тестов следует написать? Достаточно ли полно протестирована система, чтобы ее можно было передать в промышленную эксплуатацию? Какие дополнительные тесты нужны и зачем? И если нужно изменить систему, то как настроить набор тестов, чтобы он способствовал, а не препятствовал будущим изменениям?

Книга не увиливает от таких сложных вопросов. Она охватывает такие ключевые методы тестирования, как проектирование по контрактам, тестирование на основе свойств, граничное тестирование, критерии достаточности тестирования, мутационное тестирование и правильное использование фиктивных объектов. А там, где это уместно, дает ссылки на исследовательские работы по теме.

В то же время эта книга наглядно показывает, что сами тесты и процесс тестирования остаются настолько простыми, насколько это возможно. Эта простота достигается за счет того, что всегда принимается точка зрения разработчика, который фактически разрабатывает и запускает тесты. Книга полна примеров, которые помогут читателю сразу приступить к применению приемов в своих проектах.

Эта книга появилась на основе курса, преподававшегося в Делфтском техническом университете в течение многих лет. Этот курс по тестированию программного обеспечения я ввел в учебную программу

бакалавриата в 2003 году. В 2016-м ко мне присоединился Маурисио Аниче, а в 2019 году он полностью взял этот курс на себя. Маурисио – превосходный лектор, и в 2021 году студенты избрали его учителем года факультета электротехники, математики и компьютерных наук.

В Делфтском техническом университете мы обучаем тестированию на самом первом курсе нашей программы бакалавриата по информатике и вычислительной технике. Было трудно найти книгу, соответствующую нашему представлению о том, что эффективный инженер-программист должен быть эффективным тестировщиком программного обеспечения. Многие академические учебники сосредоточены на результатах исследований. Многие книги, ориентированные на разработчиков, посвящены конкретным инструментам или процессам.

Книга «Эффективное тестирование программного обеспечения» Маурисио Аниче (Maurício Aniche) заполняет этот пробел между теорией и практикой. Она написана для действующих разработчиков и освещает самые современные методы тестирования программного обеспечения. В то же время она идеально подходит для университетских курсов бакалавриата и учит следующее поколение информатиков, как стать эффективными тестировщиками программного обеспечения.

– *Доктор Ари ван Дерсен* (Dr. Arie van Deursen),
профессор факультета программной инженерии,
Делфтский технический университет, Нидерланды

Книга «Эффективное тестирование программного обеспечения» Маурисио Аниче (Maurício Aniche) – это практическое введение, которое поможет разработчикам тестировать свой код. Это компактный обзор основ тестирования программного обеспечения, охватывающий основные темы, о которых должен знать каждый разработчик. Сочетание теории и практики в книге показывает глубину опыта Маурисио как ученого и действующего программиста.

Мой собственный путь в информатике был довольно случайным: несколько курсов программирования в университете, обучение на рабочем месте и, наконец, курс переквалификации, ведущий к получению докторской степени. Это заставило меня завидовать программистам, которые прошли нужные курсы в нужное время и обладали теоретической базой, которой мне не хватало. Я периодически обнаруживал, что та или иная моя идея, обычно с недоработанной реализацией, оказывалась устоявшейся концепцией, о которой я не слышал. Вот почему я считаю важным читать вводные материалы, такие как эта книга.

На протяжении большей части моей профессиональной карьеры я рассматривал тестирование как необходимое зло, которое в основном связано с утомительным выполнением текстовых инструкций вручную. В настоящее время стало совершенно очевидно, что автоматизация тестирования лучше всего выполняется с помощью компью-

теров, но потребовались десятилетия, чтобы это понимание нашло широкое распространение. Вот почему разработка через тестирование, когда я впервые столкнулся с ней, сначала показалась мне безумием, а затем необходимостью.

В своей практике мне приходилось видеть много малопонятного тестового кода. Это особенно заметно задним числом, когда отсутствует давление сроков или когда модель предметной области устоялась. Я считаю, что этот тестовый код можно было бы улучшить, если бы методы структурирования и анализа задач, описанные в этой книге, имели большее распространение среди программистов. Это не означает, что все мы должны стать учеными, однако простота применения концепций может иметь большое значение. Например, я считаю проектирование по контрактам полезным при работе с компонентами, сохраняющими состояние. Возможно, я не всегда добавляю явные предварительные и заключительные проверки в свой код, но знание концепций помогает мне думать или обсуждать, что должен делать код.

Очевидно, что тестирование программного обеспечения – важная тема для разработчиков, и эта книга поможет им начать ее осваивать. Для тех из нас, кто занимается этой темой немного дольше, книга послужит хорошим напоминанием о методах, которыми мы пренебрегли или, возможно, упустили в первый раз. В книге также имеются замечательные разделы, посвященные тестированию программного обеспечения как практике, в частности краткое введение в крупномасштабное тестирование и, что мне особенно нравится, обсуждение вопросов поддержания высокого качества тестового кода. В реальном мире многие наборы тестов превращаются в источник разочарования, потому что они не поддерживаются.

Опыт Маурисио проявляется в практических рекомендациях и эвристиках, которые он включает в объяснение каждой методики. Он тщательно описывает инструменты, но позволяет читателю найти свой собственный путь (хотя обычно лучший выбор – последовать его совету). И конечно же, содержание самой книги было тщательно протестировано, так как изначально она разрабатывалась на основе его курса в Делфтском техническом университете.

Я сам часто встречался с Маурисио, когда читал лекции для его курса, после которых мы заходили в палатку в историческом центре города, чтобы отведать маринованной сельди (имеющей уникальный вкус для гурманов Северной Европы). Мы обсуждали методы программирования и тестирования, а также жизнь в Нидерландах. Я был впечатлен его стремлением дать все самое лучшее своим студентам и его идеями для исследований. Я с нетерпением жду, когда снова смогу сесть на поезд до Делфта.

– *Доктор Стив Фриман* (Dr. Steve Freeman),
автор книги «Growing Object-Oriented Software,
Guided by Tests» (Addison-Wesley Professional)

Вступление

Каждый разработчик программного обеспечения помнит конкретную ошибку, повлиявшую на его карьеру. Позвольте мне рассказать вам о моей ошибке. В 2006 году я занимал должность технического руководителя небольшой группы разработчиков, работавшей над созданием приложения для контроля платежей на заправочных станциях. В то время я заканчивал бакалавриат по информатике и начинал свою карьеру как разработчик программного обеспечения. До этого мне приходилось работать только над двумя серьезными веб-приложениями. И, как ведущий разработчик, я очень серьезно относился к своим обязанностям.

Система должна была напрямую связываться с подающими насосами. Как только клиент заканчивал заправку, бензоколонка уведомляла нашу систему и приложение запускало свой процесс: сбор информации о покупке (тип топлива, количество в литрах), расчет конечной стоимости, проведение пользователя через процесс оплаты и сохранение информации для будущей отчетности.

Программная система должна была работать на выделенном устройстве с процессором 200 МГц, 2 Мбайт ОЗУ и несколькими мегабайтами постоянной памяти. Это была первая попытка внедрения устройства в бизнес. То есть не было никакого предшествующего проекта, опыт разработки которого мы могли бы изучить или позаимствовать из него код. Мы также не могли повторно использовать какие-либо внешние библиотеки, и нам даже пришлось реализовать свою упрощенную базу данных.

Система обслуживала заправочные станции, поэтому моделирование их оборудования стало жизненно важной частью нашего процесса разработки. Мы писали новую функцию, запускали систему, запускали симулятор, моделировали несколько покупок топлива и вручную проверяли, правильно ли реагирует система.

Через несколько месяцев мы реализовали несколько важных функций. Наши тесты (выполнявшиеся вручную), включая тесты, проведенные компанией, прошли успешно. У нас появилась версия, которую можно было протестировать в реальных условиях! Но испытания в реальных условиях оказались непростыми: команде инженеров при-

шлось внести физические изменения на заправочной станции, чтобы насосы могли взаимодействовать с нашим программным обеспечением. К моему удивлению, компания решила запланировать первый пуск в Доминиканской Республике. Я был рад не только увидеть, как реализуется мой проект, но и посетить такую прекрасную страну.

Я был единственным разработчиком, который ездил в Доминиканскую Республику, поэтому я отвечал за исправление всех ошибок, обнаруженных на месте. Следил за установкой и за первым запуском программного обеспечения. Весь день наблюдал за системой, и все выглядело нормально.

А вечером мы пошли праздновать. Пиво было холодным, и гордость переполняла меня. Я лег спать рано, чтобы утром успеть подготовиться к встрече с заинтересованными сторонами и обсуждению следующих шагов проекта. Но в 6 утра в моем номере зазвонил телефон. Это был владелец опытной АЗС: «По всей видимости, ночью произошел сбой программного обеспечения. Ночная смена не знала, что делать, а бензоколонки не выдавали ни капли топлива, поэтому за всю ночь станция ничего не смогла продать!» Я был потрясен. Как такое могло случиться?

Я отправился прямо на станцию и начал отладку системы. Ошибка была вызвана непроверенной ситуацией, когда количество заправок превышало возможности системы. Мы знали, что используем встроенное устройство с ограниченным объемом памяти, поэтому приняли меры предосторожности. Но мы ни разу не проверили, что произойдет, если предел будет достигнут, – и это была наша ошибка!

Все наши тесты проводились вручную: для имитации заправки мы подходили к симулятору, нажимали кнопку, имитирующую включение насоса, качающего топливо, ждали несколько секунд (считалось, что чем дольше мы ждали, тем больше литров топлива «купили»), а затем останавливали процесс заправки. Чтобы сымитировать 100 заправок, нужно было 100 раз нажать кнопку на симуляторе. Это довольно медленная и утомительная процедура. Поэтому во время разработки мы пробовали выполнить по две-три заправки кряду. Однажды мы протестировали механизм обработки исключений, но этого было недостаточно.

Первая программная система, над которой я работал в роли ведущего разработчика, не проработала даже суток! Что я мог сделать, чтобы предотвратить ошибку? Пришло время изменить способ создания программного обеспечения, и это привело меня к тому, что я больше узнал о тестировании программного обеспечения. Конечно, в колледже нас познакомили с разными методами тестирования и рассказывали о важности тестирования программного обеспечения, но мы часто начинаем понимать ценность некоторых вещей, только когда они становятся нужны.

Сегодня я не могу представить создание системы без набора автоматизированных тестов. Такой набор может сказать мне за считанные секунды, правильный или неправильный код я написал, поэтому я работаю намного продуктивнее. Эта книга – моя попытка помочь разработчикам избежать моих ошибок.

О книге

Как и многое другое в мире разработки, тестирование программного обеспечения – это искусство. За последнее десятилетие мы узнали, что автоматизированные тесты – лучший способ тестирования программного обеспечения. Компьютеры могут выполнять сотни тестов за доли секунды, и такие наборы тестов позволяют компаниям уверенно обновлять и развертывать программное обеспечение десятки раз в день.

Автоматизации тестов посвящено огромное количество ресурсов (книг, руководств и онлайн-курсов). Независимо от используемого языка или вида разрабатываемого программного обеспечения вы сможете найти информацию, руководствуясь которой сумеете выбрать правильный инструмент. Но нам не хватает ресурсов, связанных с разработкой эффективных тестов. Автоматизация выполняет тесты, написанные разработчиком. Если они некачественные или не охватывают какие-то части кода, где могут скрываться ошибки, то такой набор тестов мало полезен.

Сообщество разработчиков относится к тестированию программного обеспечения как к виду искусства, когда вдохновленные разработчики с творческим подходом создают более эффективные наборы тестов, чем все остальные. Но в этой книге я бросаю вызов такому отношению и показываю, что тестирование программного обеспечения не обязательно должно зависеть от знаний, опыта или творческих способностей: его в значительной мере можно систематизировать.

Следуя эффективному систематическому подходу к тестированию программного обеспечения, мы больше не полагаемся на умение опытных разработчиков писать эффективные тесты. И если мы найдем способы автоматизировать большую часть процесса тестирования, то это позволит нам сосредоточиться на тестах, требующих творческого подхода.

Кому адресована книга

Эта книга написана для разработчиков, желающих больше узнать о тестировании или отточить свои навыки. Если вы имеете многолет-

ний опыт разработки программного обеспечения и написали множество автоматических тестов, но при этом всегда полагаетесь только на свою интуицию, решая, каким должен быть следующий тест, то эта книга поможет вам структурировать ваш мыслительный процесс.

Книга будет полезна разработчикам с разным уровнем знаний: начинающие смогут следовать всем примерам кода и методам, которые я представляю; опытные познакомятся с методами, которые им, возможно, еще не знакомы, и в каждой главе будут учиться на обсуждениях практических приемов.

Методы тестирования, которые я описываю, предназначены для разработчиков, пишущих код. Тестировщики программного обеспечения, преданные своему делу, которые рассматривают программы как черные ящики, тоже могут читать эту книгу, но должны иметь в виду, что она написана с точки зрения разработчика, написавшего тестируемый код.

Примеры в этой книге написаны на Java, но я сделал все возможное, чтобы избежать причудливых конструкций, которые могут быть незнакомы разработчикам, использующим другие языки программирования. Я также постарался обобщить приемы, чтобы, даже если код не переводится непосредственно в ваш контекст, вы могли заимствовать хотя бы идеи.

В главе 7 я обсуждаю проектирование систем с учетом простоты их тестирования. Эти идеи в большей степени ориентированы на разработку объектно ориентированных программных систем, чем функциональных. Однако это единственная глава, которая может не иметь прямого применения к функциональному программированию.

Организация книги

Эта книга состоит из 11 глав. В главе 1 я привожу аргументы в пользу систематического и эффективного тестирования программного обеспечения. Здесь вы познакомитесь с примером, когда два разработчика реализуют одну и ту же функцию – один небрежно, а другой систематически, – подчеркивающим различия между их подходами. Затем мы обсудим различия между модульными, интеграционными и системными тестами и я обосную, почему разработчики должны в первую очередь сосредоточиться на быстрых модульных и интеграционных тестах (согласно известной пирамиде тестирования).

Глава 2 знакомит с тестированием предметной области. Эта практика тестирования фокусируется на инженерных тестах, основанных на требованиях. Когда речь идет о требованиях, команды разработчиков программного обеспечения используют разные методы для тестирования – пользовательские истории, унифицированный язык моделирования (Unified Modeling Language, UML) или внутренние форматы. Каждый сеанс тестирования должен начинаться с формулирования требований к разрабатываемой функции.

В главе 3 показано, как использовать исходный код и структуру программы для расширения тестов, которые мы разрабатываем с приме-

нением методов тестирования предметной области. Мы можем запускать инструменты оценки покрытия кода тестами и, анализируя результаты, выявлять и исследовать те части кода, которые не были охвачены первоначальным набором тестов. Некоторые разработчики не считают оценку покрытия кода полезным показателем, но в этой главе я надеюсь убедить вас, что при правильном применении эта оценка может быть важнейшей частью процесса тестирования.

В главе 4 мы обсудим идею о том, что качество кода зависит не только от полноты тестирования, но также от организации кода и надежности классов и методов, на которые могут опираться другие классы и методы системы. Проектирование по контрактам позволяет сделать пред- и постусловия в коде явными, благодаря чему, если что-то пойдет не так, программа остановится, не вызывая других проблем.

Глава 5 знакомит с тестированием на основе свойств. Согласно этой методике тесты пишутся не на одном конкретном примере, а проверяют все свойства программы, причем за создание входных данных, соответствующих свойствам, отвечает платформа тестирования. Освоить эту технику непросто, потому что порой довольно сложно выразить свойства, и для этого требуется практика. Кроме того, тестирование на основе свойств подходит не для всех фрагментов кода, однако в этой главе вы найдете множество примеров, демонстрирующих данную идею, которые помогут вам лучше понять ее.

В главе 6 обсуждаются практические вопросы, выходящие за рамки разработки хороших тестов. В сложных системах одни классы зависят от других, и разработка тестов может стать весьма утомительным занятием. Здесь я познакомлю вас с фиктивными объектами и заглушками, которые позволят игнорировать некоторые зависимости во время тестирования. Мы также обсудим важный компромисс: фиктивные объекты упрощают тестирование, но они делают тесты более тесно привязанными к производственному коду, что может усложнить дальнейшее их развитие. В главе обсуждаются плюсы и минусы фиктивных объектов, а также перечисляются ситуации, когда их можно или нельзя использовать.

В главе 7 я объясняю разницу между системами, разработанными с учетом и без учета поддержки тестирования. Мы обсудим несколько простых шаблонов проектирования, которые помогут вам писать легко контролируемый и простой для наблюдения код (мечта любого разработчика, когда дело доходит до тестирования). Эта глава посвящена разработке программного обеспечения, а также тестированию – как вы увидите, между ними существует тесная связь.

В главе 8 обсуждается методика разработки через тестирование (Test-Driven Development, TDD), согласно которой тесты пишутся до основного кода, который они тестируют. TDD – чрезвычайно популярный метод, особенно среди практикующих методики гибкой разработки. Я рекомендую прочитать эту главу, даже если вы уже знакомы с TDD, потому что у меня несколько необычный взгляд на то, как

следует применять TDD, и, в частности, на случаи, в которых, как мне кажется, TDD не играет большой роли.

В главе 9 я выйду за рамки модульных тестов и уделю внимание интеграционным и системным тестам. Вы увидите, как методы, обсуждавшиеся в предыдущих главах (например, предметное и структурное тестирование), могут непосредственно применяться в этих тестах. Для написания интеграционных и системных тестов требуется гораздо больше кода, поэтому, если не организовать код должным образом, может получиться весьма сложный набор тестов. В этой главе будет представлено несколько передовых методов разработки надежных и простых в сопровождении наборов тестов.

В главе 10 обсуждается передовой опыт работы с тестовым кодом. Написание тестов в автоматическом режиме является фундаментальной частью нашего процесса. Для нас также желательно, чтобы тестовый код было легко понять и поддерживать. В этой главе будут представлены лучшие (чего мы хотим от наших тестов) и худшие практики (чего мы не хотим от наших тестов).

В главе 11 я вернусь к некоторым понятиям, затронутым в книге, повторю некоторые важные темы и дам несколько последних советов о том, что делать дальше.

О чем не рассказывается в книге

В этой книге не рассматривается тестирование программного обеспечения с использованием конкретных технологий и окружений, например здесь вы не найдете обсуждения выбора фреймворка или способов тестирования мобильных приложений, приложений React или распределенных систем.

Я уверен, что все практики и методы, о которых я рассказываю, применимы к любой программной системе. Эта книга может служить основой для любого вида тестирования. Однако в каждой области есть свои методы и инструменты тестирования; поэтому после прочтения вам следует поискать дополнительные ресурсы, касающиеся конкретного типа приложения, которое вы создаете.

Эта книга посвящена функциональному тестированию (здесь вы не найдете приемов тестирования производительности, масштабируемости и безопасности). Если ваше приложение требует такого типа тестирования, я предлагаю вам поискать специальные ресурсы по этой теме.

О примерах программного кода

Все идеи и концепции в этой книге иллюстрируются программным кодом на Java. Однако код написан так, чтобы разработчики на других языках могли читать его и понимать предлагаемые методы.

Ради экономии места листинги кода не включают все необходимые инструкции импорта. Однако вы можете найти полный исходный код

на веб-сайте книги (www.manning.com/books/Effective-software-testing) и на GitHub (https://github.com/efficient-software-testing/code). Код был протестирован с Java 11, и я не думаю, что возникнут проблемы с более новыми версиями Java.

У меня также есть специальный веб-сайт для этой книги по адресу www.efficient-software-testing.com, где я делюсь свежей информацией по тестированию программного обеспечения. Кроме того, вы можете подписаться на мою бесплатную рассылку новостей.

Живое обсуждение книги

Приобретая книгу «Эффективное тестирование программного обеспечения», вы получаете бесплатный доступ к частному веб-форуму, организованному издательством Manning Publications, где можно оставлять комментарии о книге, задавать технические вопросы, а также получать помощь от автора и других пользователей. Чтобы получить доступ к форуму и зарегистрироваться на нем, откройте в веб-браузере страницу https://livebook.manning.com/book/effective-software-testing/discussion. Узнать больше о форумах Manning и познакомиться с правилами поведения можно по адресу https://livebook.manning.com/discussion.

Издательство Manning обязуется предоставить своим читателям место встречи, где может состояться содержательный диалог между отдельными читателями и между читателями и автором. Но со стороны авторов отсутствуют какие-либо обязательства уделять форуму какое-то определенное внимание – их присутствие на форуме остается добровольным (и неоплачиваемым). Мы предлагаем задавать авторам стимулирующие вопросы, чтобы их интерес не угасал! Форум и архив с предыдущими обсуждениями остается доступным на сайте издательства, пока книга продолжает издаваться.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Доктор Маурисио Аниче (Maurício Aniche) считает своей целью повышение профессионального уровня инженеров-программистов. Он возглавляет Техническую академию в голландской платежной компании Adyen, которая позволяет предприятиям принимать электронные платежи через мобильные устройства и терминалы.

Маурисио также является доцентом кафедры разработки программного обеспечения в Делфтском техническом университете в Нидерландах, где исследует вопросы повышения продуктивности разработчиков на этапах тестирования и обслуживания программного обеспечения. Его усилия по обучению тестированию принесли ему награду «Учитель информатики 2021 года» и престижную стипендию сообщества преподавателей Делфтского университета, присуждаемую лекторам-новаторам.

Маурисио имеет степень магистра и доктора информатики, полученную в университете Сан-Паулу, Бразилия. Во время учебы в магистратуре он стал соучредителем Alura, одной из самых популярных платформ электронного обучения для инженеров-программистов в Бразилии. Является автором двух популярных среди бразильских разработчиков книг на бразильском португальском языке: «Test-Driven Development in the Real World» и «Object-Oriented Programming and SOLID for Ninjas».

Маурисио твердо верит, что наукоемкость разработки программного обеспечения будет только расти, одной из своих задач он считает ознакомление практикующих специалистов с последними достижениями ученых и стремится дать понять, что ученые понимают реальные проблемы, с которыми программисты сталкиваются в своей повседневной работе.

1 Эффективное и систематическое тестирование программного обеспечения

В этой главе:

- важность эффективного систематического тестирования;
- почему тестирование программного обеспечения затруднено и почему не существует систем без ошибок;
- знакомство с пирамидой тестирования.

В сообществе разработчиков давно утихли споры о важности тестирования программного обеспечения. Каждый разработчик понимает, что сбои в программе могут нанести серьезный ущерб бизнесу, людям или даже обществу в целом. Раньше разработчики несли основную ответственность только за создание программных систем, однако сегодня они также отвечают за качество программных продуктов, которые производят.

Наше сообщество создало множество инструментов мирового класса, помогающих разработчикам в тестировании, включая JUnit, AssertJ, Selenium и jqwik. Мы научились использовать процесс написания тестов, чтобы поразмышлять о том, что должны делать программы, и об организации кода (или организации классов, если вы используете объектно ориентированный язык). Мы также узнали, что написание тестового кода является сложной задачей, и внимание к качеству тестового кода имеет основополагающее значение для поступательного развития набора тестов. Наконец, мы узнали, какие ошибки наиболее распространены и как их искать.

Обычно разработчики очень хорошо владеют инструментами тестирования, но они редко применяют методы систематического

тестирования для поиска и изучения ошибок. Многие практики утверждают, что тесты – это инструмент обратной связи и их следует использовать в основном для того, чтобы помочь в разработке. Это правда (и в этой книге я покажу, как слушать ваш тестовый код), однако тесты также могут помочь в поиске ошибок. В конце концов, в этом и заключается идея тестирования программного обеспечения – поиске ошибок!

Большинству разработчиков не нравится писать тесты. Я слышал много отговорок: написание производственного кода увлекательнее и сложнее, а тестирование требует слишком много времени, нам платят за написание производственного кода и т. д. Разработчики также переоценивают время, которое они тратят на тестирование, как отмечают Беллер (Beller) и его коллеги в результатах опроса, проведенного среди сотен разработчиков в 2019 году. Моя цель в этой книге – убедить вас, что (1) как разработчик вы обязаны обеспечить качество производимого вами кода; (2) тесты – единственный инструмент, который способен помочь справиться с этой ответственностью; и (3) если вы будете неукоснительно следовать методологии, то сможете эффективно и систематически тестировать свой код.

Обратите внимание на слова, которые я использовал: *эффективно* и *систематически*. Вскоре вы поймете, что имелось в виду. Но сначала я хочу убедить вас в необходимости тестирования.

1.1 Разница между разработчиками, тестирующими и не тестирующими свой код

Поздно вечером в пятницу Джон собрался реализовать последнюю функцию очередного этапа разработки. Он работает над гибкой системой управления программным обеспечением, и эта последняя функция должна будет помогать разработчикам в покере планирования.

Покер планирования

Покер планирования (planning poker) – популярный метод гибкой оценки. В покере планирования разработчики оценивают усилия, необходимые для реализации определенной функции. После обсуждения в команде каждый разработчик дает свою оценку: некоторое число от единицы до числа, которое определяет команда. Чем больше число, тем больше предполагается приложить усилий для реализации функции. Например, разработчик, дающий оценку в восемь баллов, предполагает, что для реализации этой функции потребуется в четыре раза больше усилий, чем по мнению разработчика, дающего оценку в два балла.

> Разработчики с наименьшей и с наибольшей оценкой объясняют свои точки зрения другим членам команды. После дальнейшего обсуждения покер планирования повторяется, пока члены команды не договорятся о том, сколько усилий потребуется для реализации этой функции. Дополнительную информацию о технике покера планирования можно найти в книге Маркуса Хаммарберга (Marcus Hammarberg) и Йоахима Сандена (Joakim Sundén) «Kanban in Action» (2014).

Джон собирается реализовать основной метод этой функции. Метод получает список оценок и выдает имена двух разработчиков, которые должны объяснить свою точку зрения. Вот что он планирует сделать.

Метод: `identifyExtremes`

Метод должен получить список разработчиков и их оценки и вернуть имена двух разработчиков с крайними оценками.

На входе: список оценок `Estimate`, каждая из которых содержит имя разработчика и его оценку.

На выходе: список строк, содержащих с именами двух разработчиков, давших самую низкую и самую высокую оценки.

Через несколько минут Джон набросал код, представленный в листинге 1.1.

Листинг 1.1 Первая реализация PlanningPoker

```
public class PlanningPoker {
  public List<String> identifyExtremes(List<Estimate> estimates) {

    Estimate lowestEstimate = null;
    Estimate highestEstimate = null;

    for(Estimate estimate: estimates) {

      if(highestEstimate == null ||
         estimate.getEstimate() > highestEstimate.getEstimate()) {
        highestEstimate = estimate;
      }
      else if(lowestEstimate == null ||
              estimate.getEstimate() < lowestEstimate.getEstimate()) {
        lowestEstimate = estimate;
      }
    }

    return Arrays.asList(
        lowestEstimate.getDeveloper(),
        highestEstimate.getDeveloper()
    );
  }
}
```

Переменные для сохранения самой низкой и самой высокой оценок

Если текущая оценка выше, чем самая высокая оценка, встреченная до сих пор, то заменить предыдущую самую высокую оценку текущей

Если текущая оценка выше, чем самая высокая оценка, встреченная до сих пор, то заменить предыдущую самую высокую оценку текущей

Вернуть имена разработчиков, давших самую низкую и самую высокую оценки

Логика проста: алгоритм перебирает все оценки в списке, отыскивает самую высокую и самую низкую оценки и возвращает имена разработчиков, давших самую высокую и самую низкую оценки. Переменные `lowEstimate` и `highEstimate` инициализируются пустым (`null`) значением, а затем заменяются первой оценкой в цикле `for`.

Обобщение примеров кода

Опытные разработчики могут усомниться в некоторых моих решениях. Возможно, класс `Estimate` не лучший способ представить разработчиков и их оценки. Вероятно, логика поиска наименьшей и наибольшей оценок не самая лучшая. Может быть, операторы `if` могут быть проще. Я согласен. Но моя цель в этой книге не показать приемы объектно ориентированного проектирования или наилучшие практики реализации кода, а познакомить вас с особенностями тестирования написанного кода.

Методы, которые я покажу в этой книге, будут работать независимо от того, как вы реализуете свой код. Поэтому будьте терпимее, когда увидите фрагмент кода, который, по вашему мнению, можно было бы сделать лучше. Попробуйте обобщить мои примеры на свой собственный код. Что касается сложности, я уверен, что вы уже встречали код, подобный представленному в листинге 1.1.

Джон не является поклонником (автоматического) тестирования программного обеспечения. Как и многие другие разработчики, которые не автоматизируют свои тесты, Джон запускает готовое приложение и пробует ввести несколько входных данных. Одно из таких испытаний можно видеть на рис. 1.1. Джон видит, что для входных данных на рисунке (оценки Теда, Барни, Лили и Робина) программа выдает правильный результат.

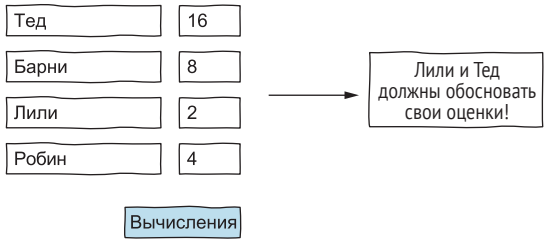


Рис. 1.1 Джон выполняет тестирование вручную перед выпуском приложения

Джон доволен результатами: его реализация заработала с самого начала. Он отправляет свой код в репозиторий, и новая функция автоматически развертывается у клиентов. Джон идет домой, предвкушая отдых в выходные, но не прошло и часа, как служба поддержки начинает получать электронные письма от разгневанных клиентов. Программное обеспечение выдает неверные результаты!

Джон возвращается к работе, просматривает журналы и быстро определяет случай, когда код дает сбой. Сможете ли вы сами найти такие входные данные, которые приводит к сбою в программе? Как показано на рис. 1.2, если оценки разработчиков (случайно) оказались расположены в порядке возрастания, то программа генерирует исключение обращения по пустому указателю.

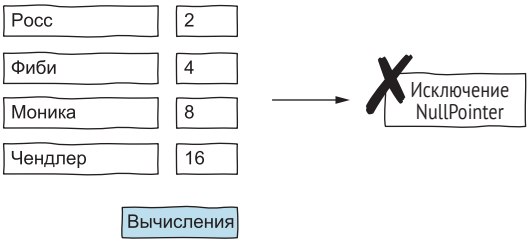


Рис. 1.2 Джон определил, в каком случае его код дает сбой

Джону не потребовалось много времени, чтобы найти ошибку в своем коде: лишний оператор `else` в листинге 1.1. В случае, когда оценки поступают в порядке возрастания, этот невинный оператор `else` препятствует изменению переменной `lowEstimate` с наименьшей оценкой в списке, потому что предыдущий `if` всегда оценивает свое выражение как истинное.

Джон заменил `else if` на `if`, как показано в листинге 1.2. Затем запустил программу и опробовал ее с теми же входными данными. Кажется, все работает. Программное обеспечение снова развертывается, и Джон возвращается домой.

Листинг 1.2 Исправленная реализация PlanningPoker

```
if(highestEstimate == null ||
   estimate.getEstimate() > highestEstimate.getEstimate()) {
  highestEstimate = estimate;
}

if(lowestEstimate == null ||
   estimate.getEstimate() < lowestEstimate.getEstimate()) {
  lowestEstimate = estimate;
}
```

Джон исправил ошибку, заменив «else if» на «if»

Вы могли бы подумать: «Эту ошибку так легко заметить! Я бы никогда не совершил такого ляпа!» Возможно, что это действительно так. Но на практике сложно уследить за всем, что может произойти в нашем коде. И это особенно сложно, когда сам код сложный. Ошибки случаются не потому, что мы плохие программисты, а потому что мы часто пишем сложный код (и потому что компьютеры намного точнее нас, людей).

Давайте обобщим случай с Джоном. Джон – очень хороший и опытный разработчик, но, как и всякий человек, может ошибиться. Он вы-

полнил некоторое тестирование вручную, прежде чем выпустить свой код, тестирование вручную занимает слишком много времени, когда требуется исследовать много разных случаев. Кроме того, Джон не придерживался систематического подхода к тестированию – он просто опробовал несколько входных данных, которые пришли ему на ум. Следуя своим придуманным методам, таким как «доверяй своей интуиции», можно позабыть о крайних случаях. Джону очень пошли бы на пользу (1) более систематический подход к определению тестов, чтобы уменьшить вероятность забыть какой-то случай; и (2) автоматизация тестирования, чтобы не приходилось тратить время на выполнение тестов вручную.

Теперь повторим тот же сценарий, но с Элеонорой. Элеонора – тоже очень хороший и опытный разработчик. Она имеет за плечами богатую практику тестирования программного обеспечения и развертывает его только после того, как разработает надежный набор тестов, охватывающий весь написанный ею код.

Предположим, что Элеонора пишет тот же код, что и Джон (листинг 1.1). Она не занимается разработкой через тестирование (TDD), но проводит надлежащее тестирование после написания своего кода.

> **ПРИМЕЧАНИЕ** Методология разработки через тестирование предполагает написание тестов до тестируемой ими реализации. Но неиспользование TDD не является проблемой, как мы обсудим в главе 8.

Элеонора рассуждает о том, что делает метод `identityExtremes`, так же как Джон. Сначала она фокусируется на входных данных метода: списке оценок `Estimate`. Она знает, что метод может получить разные варианты списков: `null` вместо списка, пустой список, список с одним элементом и список с несколькими элементами. Откуда она это знает? Она прочитала эту книгу!

Элеонора размышляет над первыми тремя случаями (`null`, пустой список, единственный элемент) и пытается понять, как этот метод будет сочетаться с остальной частью системы. Текущая реализация в этих случаях сгенерирует ошибку! Итак, она решает, что метод должен отклонять такие входные данные. Она возвращается к коду и добавляет следующие проверки (листинг 1.3).

Листинг 1.3 Дополнительные проверки входных данных

```
public List<String> identifyExtremes(List<Estimate> estimates) {
  if(estimates == null) {
    throw new IllegalArgumentException("estimates cannot be null");
  }

  if(estimates.size() <= 1) {
    throw new IllegalArgumentException("there has to be more than 1
    ➥ estimate in the list");
```

Список оценок должен существовать

Список оценок должен содержать более одного элемента

```
  }

  // здесь следует остальной код...
}
```

Элеонора уверена, что теперь метод корректно обрабатывает недопустимые входные данные (это видно по коду), но она решает написать автоматизированный тест, формализующий тестовый пример. Этот тест также поможет предотвратить регрессии в будущем, если другой разработчик не поймет, зачем добавлены проверки в код, и удалит их. Тест гарантирует, что ошибка тут же будет замечена. В листинге 1.4 показаны три тестовых примера (обратите внимание, что я специально сделал эти тесты очень подробными, чтобы их было легко понять).

Листинг 1.4 Тесты, проверяющие передачу null вместо списка, пустого списка и списка с одним элементом

```
public class PlanningPokerTest {
  @Test
  void rejectNullInput() {

    assertThatThrownBy(
      () -> new PlanningPoker().identifyExtremes(null)
    ).isInstanceOf(IllegalArgumentException.class);
  }

  @Test
  void rejectEmptyList() {

    assertThatThrownBy(() -> {
      List<Estimate> emptyList = Collections.emptyList();
      new PlanningPoker().identifyExtremes(emptyList);
    }).isInstanceOf(IllegalArgumentException.class);
  }

  @Test
  void rejectSingleEstimate() {

    assertThatThrownBy(() -> {
      List<Estimate> list = Arrays.asList(new Estimate("Eleanor", 1));
      new PlanningPoker().identifyExtremes(list);
    }).isInstanceOf(IllegalArgumentException.class);
  }
}
```

Проверяет появление исключения при вызове метода

Проверяет, что возникает исключение IllegalArgumentException

Как и в предыдущем тесте, проверяет появление исключения, если на вход передан пустой список

Проверяет появление исключения при передаче списка с одной оценкой

Три теста имеют одинаковую структуру. Все они вызывают тестируемый метод с недопустимыми входными данными и проверяют, вызывает ли он исключение `IllegalArgumentException`. Это обычное поведение проверок `assert` в Java. Метод `assertThatThrownBy`, предоставляемый библиотекой AssertJ (https://assertj.github.io/doc/), позволяет убедиться, что метод генерирует исключение. Также обратите

внимание на метод `isInstanceOf`, который позволяет нам проверить тип исключения.

Для тех, кто не знаком с лямбда-выражениями в Java, отмечу, что синтаксис `()->` фактически определяет встроенный блок кода. Это особенно очевидно во втором тесте, `rejectEmptyList()`, где блок ограничивается фигурными скобками { и }. Фреймворк тестирования выполнит этот блок кода и, если произойдет исключение, проверит его тип. Если тип исключения совпадает с указанным, то тест будет считаться пройденным. Обратите внимание, что этот тест не будет пройден, если тестируемый метод не сгенерирует исключение; в конце концов, наличие исключения – это ожидаемое поведение в данном случае.

> **ПРИМЕЧАНИЕ** Если вы новичок в автоматизированных тестах, то этот код может вас смутить. Тестирование исключений требует дополнительного кода, а кроме того, это тест «наоборот», который считается пройденным, если генерируется исключение! Но не волнуйтесь – чем больше тестовых методов вы увидите, тем понятнее они будут выглядеть для вас.

Определив тесты для проверки поведения кода при получении недопустимых входных данных, Элеонора может сосредоточиться на тестах, проверяющих правильное поведение программы. В данном случае это означает правильную обработку списков оценок с несколькими элементами. Решить, сколько элементов передать, всегда сложно, но Элеонора видит как минимум два случая: список, содержащий ровно два элемента, и список, содержащий более двух элементов. Почему два? Два – это наименьшее количество элементов в списке, с которым метод должен работать. Между списком с одним элементом (который недопустим) и с двумя (который допустим) проходит граница. Элеонора знает, что «жучки» очень любят границы, поэтому она решает добавить специальный тест, показанный в листинге 1.5.

Он напоминает более традиционный тестовый пример. Здесь определяется входное значение для передачи тестируемому методу (в данном случае список с двумя оценками), затем вызывается тестируемый метод с этим списком и, наконец, проверяется, что метод возвращает два ожидаемых имени.

Листинг 1.5 Тест для проверки списка с двумя элементами

```
@Test
void twoEstimates() {
  List<Estimate> list = Arrays.asList(
    new Estimate("Mauricio", 10),
    new Estimate("Frank", 5)
  );

  List<String> devs = new PlanningPoker()
```

Объявление списка с двумя оценками

Вызов тестируемого метода identityExtremes

```
    .identifyExtremes(list);

  assertThat(devs)
    .containsExactlyInAnyOrder("Mauricio", "Frank");
}

@Test
void manyEstimates() {
  List<Estimate> list = Arrays.asList(
    new Estimate("Mauricio", 10),
    new Estimate("Arie", 5),
    new Estimate("Frank", 7)
  );

  List<String> devs = new PlanningPoker()
    .identifyExtremes(list);

  assertThat(devs)
    .containsExactlyInAnyOrder("Mauricio", "Arie");
}
```

Проверка, что метод правильно возвращает два имени

Объявление еще одного списка с оценками, теперь уже с тремя

Вызов тестируемого метода

Проверка, что возвращаются два ожидаемых имени разработчиков: Mauricio и Arie

Прежде чем продолжить, я хочу подчеркнуть, что у Элеоноры теперь есть пять тестов, которые проходятся успешно, но ошибка `else if` все еще не выявлена. Элеонора даже не подозревает о ней (точнее, она ее пока не нашла). Однако она знает, что всякий раз, когда на вход передаются списки, порядок элементов может влиять на алгоритм. Поэтому она решает написать тест, передающий оценки в случайном порядке. Для этого теста Элеонора использует не тестирование на основе примеров (когда выбирается один конкретный пример входных данных из множества возможных), а тестирование на основе свойств, как показано в листинге 1.6.

Листинг 1.6 Тестирование обработки нескольких оценок на основе свойств

```
@Property
void inAnyOrder(@ForAll("estimates") List<Estimate> estimates) {

  estimates.add(new Estimate("MrLowEstimate", 1));
  estimates.add(new Estimate("MsHighEstimate", 100));

  Collections.shuffle(estimates);

  List<String> dev = new PlanningPoker().identifyExtremes(estimates);

  assertThat(dev)
      .containsExactlyInAnyOrder("MrLowEstimate", "MsHighEstimate");
```

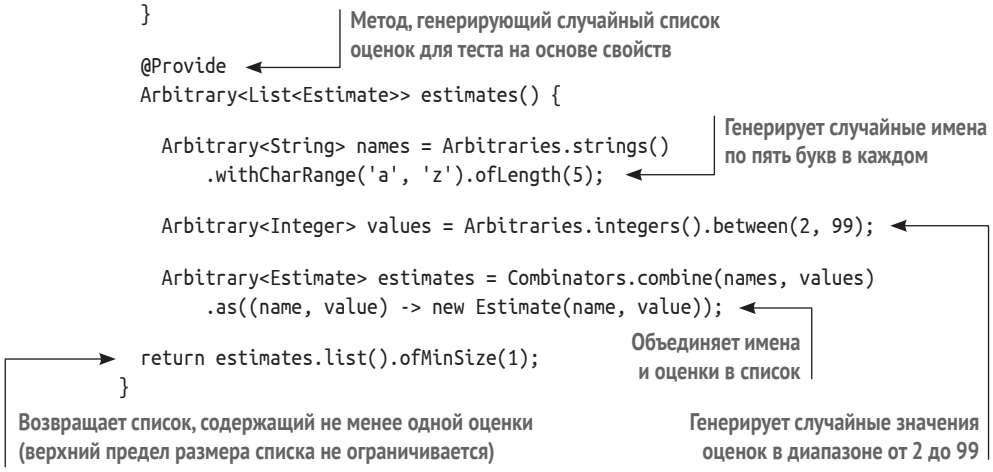
Объявление метода тестом на основе свойств вместо традиционного теста JUnit

Список, генерируемый фреймворком, будет содержать случайно сгенерированные оценки. Этот список создается методом с именем estimates (определяется ниже)

Добавляет в сгенерированный список элементы, которые гарантированно имеют самую низкую и самую высокую оценки

Перемешивает список в случайном порядке

Проверяет, что, независимо от порядка оценок в списке, всегда будут возвращаться имена MrLowEstimate и MsHighEstimate



Цель тестирования на основе свойств состоит в том, чтобы подтвердить конкретное свойство. Более подробно этот вид тестирования мы обсудим в главе 5, а пока лишь кратко опишу наш тест. Метод `estimates()` возвращает случайные оценки `Estimate`. Согласно реализации метода оценка получает случайное имя разработчика (для простоты длиной в пять символов) и случайную оценку в диапазоне от 2 до 99. Полученный список оценок возвращается тестовому методу. Сгенерированные списки содержат не менее одного элемента. Затем тестовый метод добавляет еще две оценки: самую низкую и самую высокую. Поскольку в сгенерированном списке присутствуют оценки не меньше 2 и не больше 99, метод добавляет еще две оценки, которые гарантированно являются самой низкой и самой высокой, используя значения 1 и 100 соответственно. Далее список перемешивается, поэтому порядок следования оценок не имеет значения. Наконец, выполняется проверка, что независимо от содержимого списка всегда возвращаются имена `MrLowEstimate` и `MsHighEstimate`.

Фреймворк тестирования на основе свойств запускает один и тот же тест 100 раз, каждый раз с различной комбинацией оценок. Если тест потерпит неудачу хотя бы в одном случае, фреймворк останавливает тест и сообщает входные данные, которые привели к сбою. В этой книге мы будем использовать библиотеку jqwik (https://jqwik.net), но вы легко найдете фреймворк тестирования на основе свойств для своего языка.

К удивлению Элеоноры, когда она запустила этот тест, он потерпел неудачу! Изучив образец входных данных, предоставленный тестом, она нашла ошибку в операторе `else if` и заменила его простым `if`. После этого тест начал выполняться успешно.

Элеонора решает удалить тест `manyEstimates`, так как те же проверки выполняют новый тест на основе свойств. Удалять ли повторяющиеся тесты – дело вкуса; вы можете возразить, что простой тест на основе примеров проще понять, чем тест на основе свойств. И наличие простых тестов, которые быстро объясняют поведение тестиру-

емого кода, всегда полезно, даже если при этом в ходе тестирования некоторые проверки будут дублироваться.

Затем Элеонора вспоминает, что повторяющиеся элементы в списках тоже могут вызывать нарушения в работе кода. В данном случае такими повторяющимися элементами будут имена разработчиков, давших одинаковые оценки. Она не учла этот случай в своей реализации. Размышляя о том, как это повлияет на метод, она консультируется с владельцем продукта, который в свою очередь решает, что программа должна вернуть имя того разработчика из числа давших одинаковую оценку, который стоит первым в списке.

Элеонора замечает, что программа именно так и действует. Тем не менее она решает формализовать это поведение в виде теста, показанного в листинге 1.7. Тест прост: он создает список, в котором два разработчика имеют одинаковую самую низкую оценку, а два других разработчика – одинаковую самую высокую оценку. Затем тест вызывает тестируемый метод и проверяет, что он возвращает имена разработчиков, которые стоят в списке раньше.

Листинг 1.7 Тест, проверяющий получение имен разработчиков, которые стоят в списке раньше

```
@Test
void developersWithSameEstimates() {
  List<Estimate> list = Arrays.asList(
    new Estimate("Mauricio", 10),
    new Estimate("Arie", 5),
    new Estimate("Andy", 10),
    new Estimate("Frank", 7),
    new Estimate("Annibale", 5)
  );
  List<String> devs = new PlanningPoker().identifyExtremes(list);

  assertThat(devs)
      .containsExactlyInAnyOrder("Mauricio", "Arie");
}
```

Объявление списка оценок с повторяющимися значениями

Проверка, что всякий раз, когда есть имеются повторяющиеся оценки, метод возвращает имя разработчика, которое стоит в списке раньше

Затем Элеонора задумалась над тем, что случится, если в списке будут только разработчики с одинаковыми оценками. Это еще один крайний случай, который обнаруживается при систематическом подходе к анализу входных данных, представленных в виде списка. Списки с нулевым числом элементов, с одним элементом, с множеством элементов, с разными значениями и с одинаковыми значениями – все это типичные примеры, которые нужно проверять всякий раз, когда списки используются в качестве входных данных.

Она снова поговорила с владельцем продукта. Тот удивился, что не предвидел этот крайний случай, и попросил, чтобы в этом случае код возвращал пустой список. Элеонора изменила реализацию, добавив оператор `if` в конец метода, как показано в листинге 1.8.

Листинг 1.8 Возврат пустого списка, если все оценки во входном списке одинаковые

```
public List<String> identifyExtremes(List<Estimate> estimates) {

  if(estimates == null) {
    throw new IllegalArgumentException("Estimates
    ➥ cannot be null");
  }

  if(estimates.size() <= 1) {
    throw new IllegalArgumentException("There has to be
    ➥ more than 1 estimate in the list");
  }

  Estimate lowestEstimate = null;
  Estimate highestEstimate = null;

  for(Estimate estimate: estimates) {
    if(highestEstimate == null ||
       estimate.getEstimate() > highestEstimate.getEstimate()) {
      highestEstimate = estimate;
    }

    if(lowestEstimate == null ||
       estimate.getEstimate() < lowestEstimate.getEstimate()) {
      lowestEstimate = estimate;
    }
  }

  if(lowestEstimate.equals(highestEstimate))
    return Collections.emptyList();

  return Arrays.asList(
      lowestEstimate.getDeveloper(),
      highestEstimate.getDeveloper()
  );
}
```

Если наибольшая и наименьшая оценки совпадают, то это означает, что все разработчики дали одинаковые оценки, и поэтому нужно вернуть пустой список

Для проверки этой обновленной реализации Элеонора написала новый тест (листинг 1.9).

Листинг 1.9 Проверка получения пустого списка, когда все оценки совпадают

```
@Test
void allDevelopersWithTheSameEstimate() {
  List<Estimate> list = Arrays.asList(
      new Estimate("Mauricio", 10),
      new Estimate("Arie", 10),
      new Estimate("Andy", 10),
      new Estimate("Frank", 10),
      new Estimate("Annibale", 10)
  );
```

Объявление списка, в котором все оценки одинаковые

```
  List<String> devs = new PlanningPoker().identifyExtremes(list);

  assertThat(devs).isEmpty();  <-- Проверка, что возвращаемый список пуст
}
```

Теперь Элеонора удовлетворена своим набором тестов. В качестве следующего шага она решает сосредоточиться на самом коде. Возможно, есть что-то еще, что не проверяется тестами. Чтобы помочь себе в этом, она запускает инструмент оценки охвата кода тестами, имеющийся в ее IDE (рис. 1.3).

Как показала проверка, все ветви в коде охвачены тестами. Элеонора знает, что инструменты не идеальны, поэтому вручную проверила код на наличие других случаев. Она не смогла ничего найти и сделала вывод, что код достаточно протестирован. Она отправляет код в репозиторий и уезжает домой на выходные. Код немедленно поступает к клиентам. Утром в понедельник Элеонора была рада увидеть, что за выходные не было сообщено ни об одном сбое.

1.2 Эффективное тестирование программного обеспечения для разработчиков

Я надеюсь, что разница между двумя разработчиками, описанными в предыдущем разделе, достаточно ясна. Элеонора использовала автоматизированные тесты, систематически и эффективно разрабатывала сценарии тестирования. Она разбила требования на небольшие части и использовала их для создания тестов, применив технику, называемую *тестированием предметной области*. Закончив со спецификацией, она сосредоточилась на коде и с помощью *структурного тестирования* (позволяющего оценить охват кода) убедилась в достаточности текущих тестов. Для некоторых проверок Элеонора написала *тесты на основе примеров* (т. е. выбрала для тестирования одну точку данных). В одном конкретном случае она использовала *тестирование на основе свойств*, которое помогло ей проверить возможные ошибки в коде. Наконец, она часто размышляла о *контрактах*, а также о *пред-* и *постусловиях* разрабатываемого ею метода (правда, она реализовала набор проверок действительности значений, а не предусловия как таковые; мы обсудим различия между контрактами и проверкой действительности в главе 4).

Это был пример того, что я называю *эффективным и систематическим тестированием программного обеспечения*. В оставшейся части этой главы я объясню, как разработчики программного обеспечения могут проводить эффективное тестирование одновременно с разработкой. Прежде чем мы углубимся в конкретные методы, я опишу эффективное тестирование в рамках процессов разработки и как методы тестирования дополняют друг друга. Я расскажу о различных типах тестов и о том, на каких из них следует сосредоточить основное

внимание. Наконец, я покажу, почему тестирование программного обеспечения так сложно.

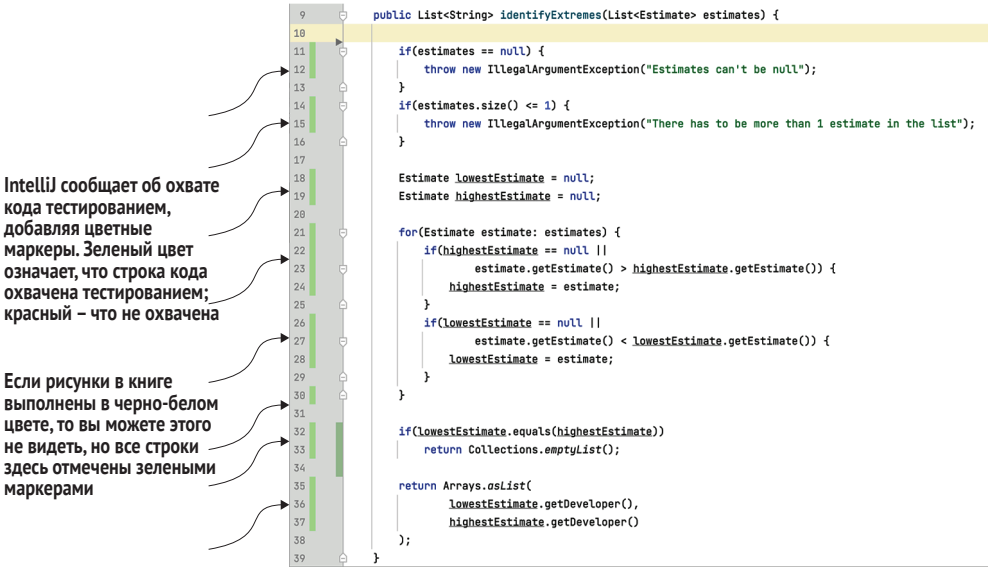


Рис. 1.3 Результат анализа охвата кода, произведенного в моей IDE IntelliJ. Как видите, все ветви охвачены тестированием

1.2.1 Эффективное тестирование в процессе разработки

В этой книге я предлагаю простой алгоритм для разработчиков, применяющих методику эффективного и систематического тестирования. Сначала мы реализуем функцию с использованием тестов, облегчающих и направляющих разработку. По окончании реализации мы погружаемся в эффективное и систематическое тестирование, чтобы убедиться, что оно работает так, как ожидалось (т. е. мы тестируем, стараясь найти ошибки). На рис. 1.4 более подробно показан рабочий процесс разработки, давайте пройдемся по нему.

1. Разработка новых особенностей часто начинается с того, что разработчик получает некоторое *требование*. Требования часто описываются на естественном языке и могут оформляться в определенном виде, например в форме сценариев использования на унифицированном языке моделирования (Unified Modeling Language, UML) или функциональных требований. После анализа требований разработчик приступает к написанию кода.
2. Чтобы сделать процесс разработки более управляемым, разработчик выполняет короткие итерации *разработки через тестирование* (TDD). Эти итерации позволяют разработчику иметь быструю обратную связь о работоспособности только что на-

писанного кода. Они также помогают ему более уверенно выполнять реорганизацию (рефакторинг) кода в процессе расширения реализации новой особенности.

3 Требования часто бывают объемными и сложными и редко реализуются одним классом или методом. Разработчик создает несколько модулей (классов и методов) с разными *контрактами*, которые взаимодействуют между собой и вместе реализуют требуемый функционал. Написание классов таким способом, чтобы их было легко тестировать, – сложная задача, и разработчик должен проектировать их с учетом *возможности тестирования*.

4 Как только разработчик удовлетворен созданными им модулями и считает, что требование к функционалу выполнено, он переходит к тестированию. Первый шаг – проверка каждого нового модуля. На этом этапе широко используются *предметное*, *граничное* и *структурное тестирование*.

5 Некоторые части системы могут потребовать создания *более крупных тестов* (интеграционных или системных). Для разработки таких тестов разработчик использует те же три метода – предметное, граничное и структурное тестирование, – но для более крупных частей программной системы.

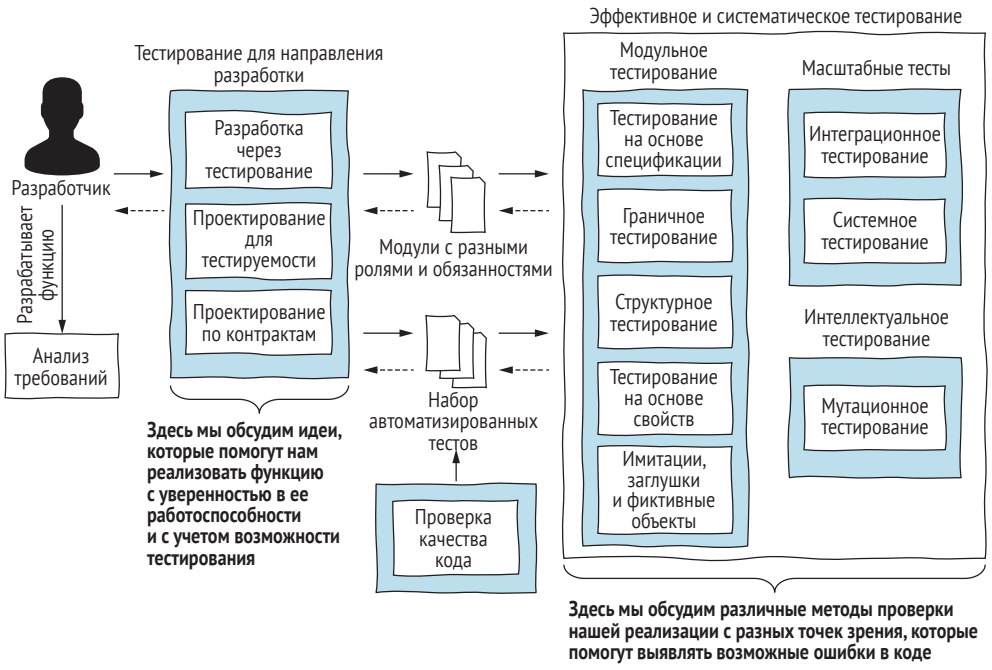


Рис. 1.4 Рабочий процесс разработчика, применяющего эффективное и систематическое тестирование. Стрелки указывают на итеративный характер процесса; разработчики могут переключаться между различными приемами по мере накопления знаний о разрабатываемой и тестируемой программе

6 По окончании проектирования тестов с использованием различных методов разработчик применяет автоматизированные *интеллектуальные инструменты тестирования* для поиска тестов, которые люди не умеют обнаруживать. В число наиболее популярных методов входят: *генерирование тестовых случаев*, *мутационное тестирование* и *статический анализ*. Из них в этой книге мы рассмотрим только *мутационное тестирование*.

7 Наконец, когда после тщательного тестирования разработчик почувствует достаточную уверенность, он выпускает новую реализованную особенность.

1.2.2 Эффективное тестирование как итеративный процесс

Предыдущая последовательность действий может выглядеть как описание последовательного/водопадного процесса, однако в действительности она является итеративной. В процессе тестирования класса разработчик может вдруг заметить, что решение, принятое им несколько часов назад, не было идеальным. Поэтому он возвращается назад и переделывает код. Выполняя циклы TDD, разработчик может понять, что требования изложены недостаточно четко. В этом случае он возвращается к анализу требований, чтобы лучше понять ожидания заказчика. Довольно часто при тестировании обнаруживаются ошибки. В таких случаях разработчик вновь возвращается к коду, исправляет его и продолжает тестирование. Или он может реализовать только половину функции и посчитает более продуктивным тщательно протестировать прямо сейчас и только потом продолжать реализацию.

Рабочий процесс разработки, который я предлагаю в этой книге, предназначен не для того, чтобы сдерживать вас. Вы можете свободно переключаться между приемами или менять порядок их применения. Вы должны найти тот стиль, который лучше всего подходит вам и делает вас наиболее продуктивным.

1.2.3 Сосредоточение внимания на разработке, а затем на тестировании

Я считаю, что на разработке и тестировании нужно сосредотачиваться отдельно. Когда я пишу код, то не хочу отвлекаться на непонятные крайние случаи. Думая о чем-то одном, я периодически делаю заметки, чтобы не забыть протестировать тот или иной случай позже. Однако я предпочитаю сосредотачивать всю свою энергию на бизнес-правилах, которые реализую, и стремлюсь упростить поддержку кода будущими разработчиками.

Закончив реализацию решения, я сосредотачиваюсь на его тестировании. Сначала использую различные приемы, как если бы работал над систематическим контрольным списком. Как было показано

в примере с Элеонорой, ей не нужно было много думать о том, что делать, когда метод получил список с исходными данными: она действовала так, будто у нее была конкретная инструкция, в которой говорилось: «Значение `null`, пустой список, один элемент, много элементов». Только после этого я использую свой творческий потенциал и знание предметной области для выполнения других дел, которые считаю важными.

1.2.4 Миф о «правильности по замыслу»

Теперь, когда вы получили более полное представление о том, что я имею в виду под эффективным и систематическим тестированием программного обеспечения, позвольте мне развенчать миф. Среди разработчиков программного обеспечения бытует мнение, что если код разрабатывается с использованием простых приемов, то в нем не будет ошибок, как будто секрет кода без ошибок в его простоте.

Эмпирические исследования в области разработки программного обеспечения неоднократно показывали, что простой код действительно менее подвержен дефектам, чем сложный (см., например, статью Шатнави (Shatnawi) и Ли (Li), написанную ими в 2006 году). Однако простота не является достаточным условием. Наивно полагать, что тестирование можно полностью заменить простотой кода. То же верно и для «правильности по замыслу» (correctness by design): если код имеет хорошо продуманную организацию, то это не означает, что в нем не может быть ошибок.

1.2.5 Стоимость тестирования

Можно подумать, что тщательное тестирование может обходиться слишком дорого. На рис. 1.4 показано множество методов, которые разработчикам приходится применять, следуя предложенному мной алгоритму. Это правда: правильное тестирование программного обеспечения увеличивает объем выполняемой работы. Но это не напрасный труд, и вот почему:

- стоимость ошибок, возникающих в ходе промышленной эксплуатации, часто превышает стоимость их предотвращения (как показали Бем (Boehm) и Папаччо (Papaccio), 1988). Представьте популярный интернет-магазин и вообразите, в какую сумму обойдется магазину отключение платежного приложения на 30 мин из-за ошибки, которую легко можно было бы предотвратить с помощью тестирования;
- команды, допускающие много ошибок, как правило, тратят массу времени на цикл, когда разработчики допускают ошибки, клиенты (или специальный отдел проверки качества) находят их, разработчики исправляют, клиенты находят другие ошибки и т. д.;
- главное – практика. Привыкнув к разработке тестов, разработчики будут делать это намного быстрее.

1.2.6 Что подразумевается под словами «эффективный» и «систематический»

Описывая, как, по моему мнению, разработчик должен подходить к тестированию, я использовал два слова: *эффективно* и *систематически*. Под эффективностью я подразумеваю сосредоточенность на разработке правильных тестов. Тестирование программного обеспечения основано на компромиссах. Тестировщики хотят максимизировать количество обнаруживаемых ошибок и минимизировать усилия, необходимые для их поиска. Как этого добиться? Только зная, что нужно тестировать.

Все приемы, которые я представляю в этой книге, имеют четкое начало (что тестировать) и четкий конец (когда остановиться). Конечно, я не гарантирую, что системы будут свободны от ошибок, если вы будете следовать этим приемам. Как сообщество, мы до сих пор не умеем создавать безошибочные системы. Но я могу с уверенностью сказать, что количество ошибок будет снижено до, надеюсь, приемлемого уровня.

Под *систематичностью* я подразумеваю создание единого набора тестов для каждого данного фрагмента кода. Тестирование часто происходит однократно. Разработчики продумывают тесты для проверки ситуаций, которые приходят им на ум. Часто можно видеть, как два разработчика создают разные наборы тестов для одной и той же программы. Мы должны стремиться систематизировать наши процессы, чтобы уменьшить зависимость от разработчика, выполняющего работу.

Я понимаю и согласен с тем, что разработка программного обеспечения – это творческий процесс, который не может выполняться роботами. Я считаю, что в создании программного обеспечения всегда будут участвовать люди; но почему бы не позволить разработчикам сосредоточиться на творческой стороне? Многое в тестировании программного обеспечения можно систематизировать, и именно об этом рассказывается в книге.

1.2.7 Роль автоматизации тестирования

Автоматизация является ключом к эффективному тестированию. Каждый тест, разрабатываемый нами, автоматизируется с помощью фреймворка тестирования, такого как JUnit. Между проектированием теста и его выполнением есть четкая грань. Как только тест написан, фреймворк запускает его и показывает отчет о его выполнении. Собственно, это все, что делают такие фреймворки. Их роль очень важна, но реальная задача в тестировании программного обеспечения заключается не в написании кода для JUnit, а в разработке действенных тестов, способных выявлять ошибки. Разработка тестов – это творческая работа, и именно этому в первую очередь посвящена книга.

ПРИМЕЧАНИЕ Если вы не знакомы с JUnit, то не считайте это проблемой, потому что примеры в книге легко читаются. Но, как я постоянно буду упоминать на протяжении всей книги, чем ближе вы знакомы с инфраструктурой тестирования, тем лучше.

В главах, где обсуждаются методы тестирования, мы будем сначала разрабатывать тесты и только потом автоматизировать их с помощью кода JUnit. В реальной жизни вы можете совмещать оба вида деятельности; но здесь я решил разделить их, чтобы вы могли видеть разницу. Также в книге мало говорится о конкретных инструментах. JUnit и другие фреймворки тестирования – мощные инструменты, и я рекомендую прочитать руководства и книги, посвященные им.

1.3 Принципы тестирования программного обеспечения (или Почему тестирование такое сложное)

С упрощенной точки зрения на тестирование программного обеспечения, если мы хотим, чтобы наши системы были хорошо протестированы, нужно продолжать добавлять тесты, пока их не наберется достаточное количество. Хотел бы я, чтобы это было так просто. Гарантировать отсутствие ошибок в программах практически невозможно, и разработчики должны понимать, почему это так.

В этом разделе я расскажу о некоторых принципах, усложняющих нашу жизнь как тестировщиков программного обеспечения, и о том, что можно предпринять, чтобы смягчить ситуацию. Эти принципы во многом были заимствованы из книги «Foundations of Software Testing ISQTB Certification» Блэка (Black), Винендаала (Veenendaal) и Грэма (Graham) (2012).

1.3.1 Всеобъемлющее тестирование невозможно

У нас нет ресурсов для реализации всеобъемлющего тестирования наших программ. Тестирование всех возможных ситуаций в программной системе может оказаться невозможным, даже если бы у нас были неограниченные ресурсы. Представьте программную систему с «всего» 300 различными флагами или конфигурационными параметрами (например, операционную систему Linux). Каждый флаг может иметь значение true или false (логическое значение) и может устанавливаться независимо от других флагов. Программная система ведет себя по-разному с каждой комбинацией флагов. Наличие двух возможных значений для каждого из 300 флагов дает 2^{300} комбинаций, которые необходимо протестировать. Для сравнения: количество атомов во Вселенной оценивается в 10^{80}. Иначе говоря, эта программная систе-

ма имеет больше возможных комбинаций для проверки, чем атомов во Вселенной.

Зная, что протестировать все невозможно, мы должны выбрать (или расставить приоритеты), что тестировать. Вот почему я подчеркиваю необходимость *эффективных тестов*. В этой книге обсуждаются приемы, которые помогут вам определить наиболее подходящие тестовые случаи.

1.3.2 Своевременное прекращение тестирования

Выбор тестов для разработки – трудная задача. Если создать слишком мало тестов, какие-то ошибки могут остаться незамеченными и программная система будет проявлять поведение, весьма далекое от предполагаемого. С другой стороны, неконтролируемое создание большого числа тестов может привести к появлению неэффективных тестов (и затратам времени и денег). Как я уже говорил, нашей целью должно быть максимальное количество найденных ошибок при минимальных затратах ресурсов на их поиск. Поэтому я расскажу о различных критериях адекватности, которые помогут вам решить, когда прекратить тестирование.

1.3.3 Изменчивость важна (парадокс пестицидов)

В тестировании программного обеспечения нет универсального рецепта. Другими словами, не существует единой методики тестирования, которую можно применить для поиска любых возможных ошибок. Разные методы тестирования помогают выявлять разные ошибки. Если использовать только один метод, то вы найдете лишь все ошибки, которые способен обнаружить этот метод, и не более того.

В качестве более конкретного примера представьте команду, использующую исключительно методы модульного тестирования. Команда может найти все ошибки, которые способен обнаружить метод модульного тестирования, но может пропустить ошибки, возникающие только на уровне интеграции.

Этот эффект известен как *парадокс пестицидов* (pesticide paradox): каждый метод, используемый для предотвращения или обнаружения «жучков», может не обнаруживать других «жучков» (не оказывать влияния), против которых он неэффективен. Тестировщики должны использовать разные стратегии тестирования, чтобы свести к минимуму количество ошибок, оставшихся в программном обеспечении. Изучая различные стратегии тестирования, представленные в этой книге, имейте в виду, что их объединение почти наверняка будет мудрым решением.

1.3.4 В одних частях ошибок больше, чем в других

Как я уже говорил выше, поскольку всеобъемлющее тестирование невозможно, тестировщики должны расставлять приоритеты, выбирая

тесты для реализации. Определяя приоритеты, обращайте внимание на возможную неравномерность распределения ошибок. Опытным путем наше сообщество заметило, что в любой системе одни компоненты содержат больше ошибок, чем другие. Например, модуль `Payment` может потребовать более тщательного тестирования, чем модуль `Marketing`.

Возьмем реальный пример, описанный Шретером (Schröter) и его коллегами (2006), которые изучали ошибки в проектах Eclipse. Они заметили, что 71 % файлов, импортировавших пакеты компилятора, пришлось исправлять позже. Другими словами, эти файлы были более подвержены дефектам, чем другие файлы в системе. Как разработчику программного обеспечения, вам придется наблюдать за своей программной системой и изучать ее поведение. Дополнительные данные, помимо исходного кода, могут помочь вам расставить приоритеты в тестировании.

1.3.5 Никакой объем тестирования не будет идеальным или достаточным

Как говорил Дейкстра (Dijkstra), «тестирование программ может использоваться для демонстрации наличия ошибок, но оно никогда не покажет их отсутствие». Другими словами, мы можем найти больше ошибок, увеличив количество тестов, но никакие наборы тестов, даже самые большие, не смогут гарантировать, полное отсутствие ошибок в программной системе. Они только гарантируют, что в ситуациях, которые мы проверяем, система ведет себя должным образом.

Это важный принцип, который нужно осознать и запомнить, так как он поможет вам формулировать ваши ожидания (и ожидания ваших клиентов). Ошибки по-прежнему будут возникать, но (надеюсь) деньги, которые вы платите за тестирование и предотвращение ошибок, окупятся за счет того, что незамеченными останутся только менее серьезные ошибки. Вы должны понять и принять утверждение: «Нельзя проверить все».

> **ПРИМЕЧАНИЕ** Хотя мониторинг не является основной темой этой книги, я рекомендую не забывать о системах мониторинга. Ошибки случаются, и вы должны быть уверены, что обнаружите их в промышленном окружении максимально быстро. Вот почему такие инструменты, как стек ELK (Elasticsearch, Logstash и Kibana; www.elastic.co), приобрели столь большую популярность. Этот подход иногда называют *тестированием в промышленном окружении* (Wilsenach, 2017).

1.3.6 Контекст имеет решающее значение

Контекст играет важную роль в разработке тестов. Например, разработка тестов для мобильного приложения радикально отличается от

разработки тестов для веб-приложения или программного обеспечения, используемого в ракете. Другими словами, тестирование зависит от контекста.

Большая часть этой книги мало зависит от контекста. Методы, которые я обсуждаю (предметное и структурное тестирование, а также тестирование на основе свойств и др.), могут применяться в программных системах любого типа. И все же, если вы работаете над мобильным приложением, я рекомендую прочитать книгу, посвященную теме тестирования мобильных приложений, когда вы закончите читать эту. В главе 9 я дам несколько подсказок, связанных с контекстом, где мы обсудим масштабные тесты.

1.3.7 Верификация не валидация

Наконец, обратите внимание, что программная система, работающая безупречно, но бесполезная для пользователей, не является хорошей программной системой. Как сказал мне рецензент этой книги, «охват кода легко измерить; охват требований – это другой вопрос». Тестировщики программного обеспечения сталкиваются с этой *проблемой отсутствия ошибок*, когда сосредотачиваются исключительно на верификации (проверке правильности), а не на валидации (проверке эффективности).

Как гласит популярная поговорка, которая может помочь вам запомнить разницу, «верификация помогает убедиться в правильной работе системы, а валидация – в ее эффективности». Эта книга в основном посвящена приемам верификации (проверки правильной работы). Другими словами, я не буду обсуждать методы взаимодействий с клиентами, чтобы понять их реальные потребности, а покажу лишь приемы, помогающие гарантировать, что при заданном конкретном требовании программная система правильно его реализует.

Верификация и валидация (проверки правильности и эффективности) могут идти рука об руку. В примере алгоритма покера планирования, представленном выше в этой главе, именно это и произошло, когда Элеонора подумала о том, что произойдет, если все разработчики дадут одинаковые оценки. Владелец продукта не подумал об этом случае. Систематический подход к тестированию может помочь вам выявить крайние случаи, которые даже эксперты по продукту не предусмотрели.

1.4 Пирамида тестирования и на чем следует сосредоточиться

Всякий раз, когда мы говорим о прагматическом тестировании, одно из первых решений, которое нужно принять, – это уровень тестирования кода. Под *уровнем тестирования* я подразумеваю *модульный*,

интеграционный или *системный* уровень. Давайте кратко рассмотрим каждый из них.

1.4.1 Модульное тестирование

В некоторых ситуациях цель тестировщика – протестировать одну функцию программного обеспечения, намеренно игнорируя все остальное. Такой подход мы видели в примере с покером планирования. Цель состояла в том, чтобы протестировать метод `identifyExtremes()`, и ничего больше. Конечно, нас заботило, как этот метод будет взаимодействовать с остальной системой, и именно поэтому мы протестировали его контракты. Однако мы не тестировали его в комплексе с другими частями системы.

Изолированное тестирование модулей по отдельности называется *модульным тестированием*. Этот уровень тестирования предлагает следующие преимущества:

- *модульные тесты выполняются быстро*. На выполнение модульного теста обычно уходит всего пара миллисекунд. Быстрые тесты позволяют тестировать огромные части системы за небольшой промежуток времени. Быстрые автоматизированные наборы тестов дают нам постоянную обратную связь. Эта быстрая система безопасности позволяет нам чувствовать себя более уверенно при внесении эволюционных изменений в программную систему;
- *модульные тесты легко контролируются*. Модульный тест проверяет программное обеспечение, передавая определенные параметры методу, а затем сравнивая возвращаемое значение с ожидаемым результатом. Входные аргументы и ожидаемый результат в тесте легко адаптировать или изменить. Опять же, взгляните на пример с `identifyExtremes()` и посмотрите, как легко было организовать передачу разных входных данных и проверить результат;
- *модульные тесты легко писать*. Они не требуют сложной настройки или дополнительной работы. Кроме того, один модуль часто является небольшим, что облегчает работу тестировщика. Намного сложнее тестировать компоненты, обращающиеся к базам данных, внешним интерфейсам и веб-сервисам.

Но модульные тесты имеют свои недостатки:

- *они далеки от реальности*. Программная система редко состоит из одного класса. Большое количество классов в системе и взаимодействия между ними могут привести к тому, что в реальном применении система будет вести себя иначе, чем в модульных тестах. Следовательно, модульные тесты не полностью отражают реальное выполнение программной системы;
- *некоторые типы ошибок не обнаруживаются*. На уровне модульного тестирование некоторые типы ошибок не обнаруживаются; они возникают только при интеграции различных компонентов (кото-

рые не участвуют в модульном тестировании). Подумайте о веб-приложении со сложным пользовательским интерфейсом: вы можете тщательно протестировать серверную часть и пользовательский интерфейс, но ошибка может проявиться только при объединении серверной части с интерфейсом. Или представьте себе многопоточный код: на уровне модулей все может работать прекрасно, но при совместной работе потоков могут возникать ошибки.

Интересно отметить, что одной из самых сложных задач модульного тестирования является определение модуля. Модулем может быть один метод или несколько классов. Вот понравившееся мне определение модульного тестирования, которое дал Рой Ошеров (Roy Osherove) (2009): «Модульный тест – это автоматизированный фрагмент кода, который выполняет единицу работы в системе. Причем единица работы может охватывать один метод, целый класс или несколько классов, работающих вместе для достижения одной единственной логической цели, которую можно проверить».

Для меня модульное тестирование означает тестирование (небольшого) набора классов, не зависящих от внешних систем (таких как базы данных или веб-сервисы) или чего-то еще, чем я не управляю. Когда я выполняю модульное тестирование целого набора классов, то количество классов, как правило, невелико в первую очередь потому, что одновременное тестирование большого количества классов может оказаться слишком сложным, а не потому, что такое тестирование нельзя отнести к разряду модульного.

Но что, если класс, который я хочу протестировать, зависит от другого класса, который, например, в свою очередь взаимодействует с базой данных (рис. 1.5)? В таких случаях модульное тестирование становится сложнее. Вот краткий ответ: если мне нужно протестировать класс, зависящий от другого класса, который в свою очередь зависит от базы данных, я создам имитацию класса, взаимодействующего с базой данных. Другими словами, я создам заглушку, действующую как оригинальный класс, но намного проще и удобнее для тестирования. Мы углубимся в эту конкретную проблему в главе 6, где будем обсуждать фиктивные объекты.

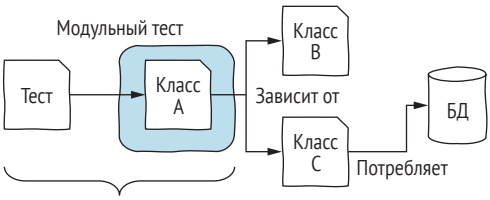


Рис. 1.5 Модульное тестирование. Наша цель – протестировать один модуль системы, максимально изолированный от остальной системы

1.4.2 Интеграционное тестирование

Модульные тесты фокусируются на самых маленьких частях системы. Однако тестирования компонентов в изоляции друг от друга иногда бывает недостаточно. Это особенно верно, когда тестируемый выполняет вызовы за пределы системы и использует другие (часто внешние) компоненты. Интеграционное тестирование – это уровень тестирования, который используется для проверки интеграции (связей) между нашим кодом и внешними компонентами и системами.

Рассмотрим реальный пример. Многие программные системы используют внешние базы данных. Для связи с базой данных обычно создается класс, единственной обязанностью которого является взаимодействие с этим внешним компонентом (это так называемые объекты доступа к данным Data Access Object [DAO]). Эти объекты DAO могут содержать сложный SQL-код. Соответственно, возникает необходимость протестировать SQL-запросы. Тестировщик не хочет тестировать всю систему, а только интеграцию между классом DAO и базой данных. Он также не хочет тестировать класс DAO в полной изоляции. В конце концов, лучший способ узнать, работает ли SQL-запрос, – передать его базе данных и посмотреть, что она вернет.

Это пример интеграционного теста. Интеграционное тестирование направлено на тестирование нескольких компонентов системы в совокупности, фокусируясь на взаимодействии между ними, а не на тестировании системы в целом (рис. 1.6). Правильно ли они взаимодействуют? Что произойдет, если компонент A отправит сообщение X компоненту B? Они по-прежнему будут демонстрировать правильное поведение?

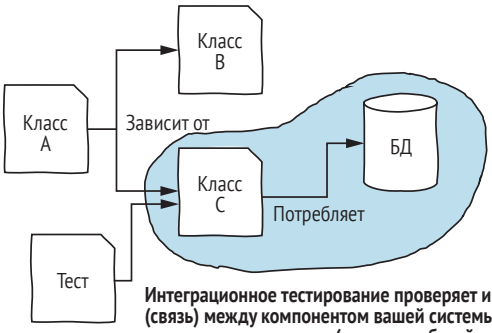


Рис. 1.6 Интеграционное тестирование. Наша цель – проверить, насколько правильно наш компонент интегрируется (взаимодействует) с внешним компонентом

В интеграционном тестировании участвуют две стороны: наш и внешний компоненты. Написание такого теста менее сложно, чем того, который охватывает всю систему и включает компоненты, нас не интересующие.

Интеграционные тесты писать сложнее, чем модульные. Например, настройка базы данных для теста требует определенных усилий. Тесты, включающие базы данных, обычно должны использовать изолированный экземпляр базы данных, обновлять схему базы данных, переводить базу данных в состояние, ожидаемое тестом, путем добавления или удаления записей, а затем очищать ее. Аналогичные усилия приходится прикладывать и на интеграционное тестирование с участием других внешних компонентов: веб-сервисов, файловой системой и т. д. Мы обсудим эффективное написание интеграционных тестов в главе 9.

1.4.3 Системное тестирование

Чтобы получить более реалистичное представление о программном обеспечении и выполнить более реалистичное тестирование, необходимо запустить в работу всю программную систему со всеми ее базами данных, внешними приложениями и другими компонентами. Тестируя систему целиком вместо отдельных ее частей, мы проводим системное тестирование (рис. 1.7). Нам все равно, какие действия система выполняет внутри; нам неважно, была ли она разработана на Java или Ruby, и не имеет значения, использует ли она реляционную базу данных. Нас интересует только результат, когда для конкретных входных данных X система выдает результат Y.

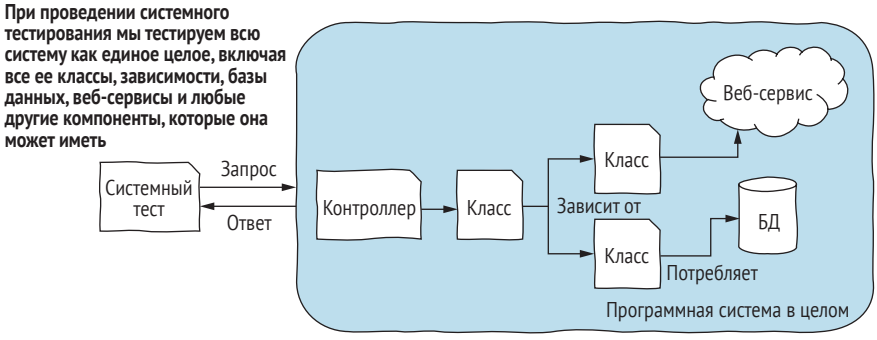


Рис. 1.7 Системное тестирование. Наша цель – протестировать работу системы в целом, всех ее компонентов

Очевидным преимуществом системного тестирования является его *реалистичность*. Наши конечные клиенты не будут вызывать метод `identifyExtremes()` изолированно, а откроют веб-страницу, отправят форму и увидят результаты. Системные тесты проверяют систему именно таким образом. Чем более реалистичны тесты (т. е. чем ближе их действия к выполняемым конечными пользователями), тем больше наша уверенность во всей системе.

Однако системное тестирование имеет свои недостатки:

- системные тесты часто выполняются *медленнее*, чем модульные. Представьте, что должен сделать системный тест, чтобы запус-

тить всю систему со всеми ее компонентами. Тест также должен взаимодействовать с реальным приложением, и на эти взаимодействия может потребоваться несколько секунд. Представьте, что тест запускает контейнеры с веб-приложением и базой данных. Затем он отправляет HTTP-запрос веб-сервису, предоставляемому этим веб-приложением. Веб-сервис извлекает данные из базы данных и возвращает ответ в формате JSON. Очевидно, что для этого требуется больше времени, чем на запуск простого модульного теста, который практически не имеет зависимостей;

- системные тесты *сложнее писать*. Некоторые компоненты (например, базы данных) могут потребовать сложной подготовки, прежде чем их можно будет использовать в сценарии тестирования. Подумайте о подключении, аутентификации и обеспечении наличия в базе данных всей информации, необходимой для теста. Дополнительный код требуется только для автоматизации тестов;
- системные тесты более *склонны к нестабильности*. *Ненадежный* тест демонстрирует ошибочное поведение: если запустить его, он может выполниться успешно или потерпеть неудачу с одной и той же конфигурацией. Нестабильность тестов – серьезная проблема для разработчиков программного обеспечения, и мы обсудим ее в главе 10. Представьте системный тест, проверяющий веб-приложение. После того как тестировщик щелкнет по кнопке, на обработку HTTP-запроса веб-приложением потребуется на полсекунды больше времени, чем обычно (из-за небольших изменений, которые мы часто не контролируем в реальных сценариях). Тест не ожидает этого и, следовательно, терпит неудачу. Затем выполняется снова, веб-приложение укладывается в отведенное время, и тест проходится успешно. Многие неопределенности в системном тесте могут привести к неожиданному поведению.

1.4.4 Когда использовать каждый уровень тестирования

Имея четкое представление о различных уровнях тестирования и их преимуществах, мы должны решить, уделять ли больше внимания модульному или системному тестированию, и определить, какие компоненты тестировать с помощью модульного тестирования, а какие – с помощью системного. Неверное решение способно существенно повлиять на качество системы: неправильный выбор уровня может потребовать слишком много ресурсов и не позволит найти достаточное количество ошибок. Как вы, наверное, догадались, лучший ответ – «все зависит от обстоятельств».

Некоторые разработчики, включая меня, отдают предпочтение модульному тестированию. Это не означает, что такие разработчики не занимаются интеграционным или системным тестированием; но, когда это возможно, они предпочитают модульное тестирование. Для иллюстрации этой идеи часто используется пирамида, изображенная

на рис. 1.8. Площадь сегмента в пирамиде пропорциональна относительному количеству тестов на каждом уровне.

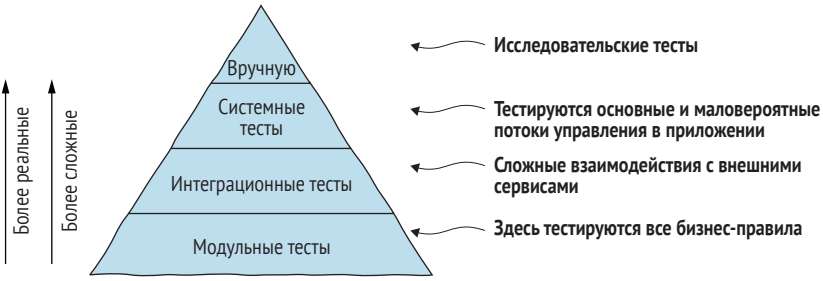


Рис. 1.8 Моя версия пирамиды тестирования. Чем ближе тест к вершине, тем более реальным и сложным он становится. Справа кратко описывается, что тестируется на каждом уровне

Модульное тестирование находится в нижней части пирамиды, и соответствующий ему сегмент имеет наибольшую площадь. Это означает, что разработчики, следующие этой схеме, должны отдавать предпочтение модульному тестированию (т. е. должны писать больше модульных тестов). Следующий уровень – интеграционное тестирование. Площадь этого сегмента меньше, а это означает, что на практике интеграционных тестов пишется меньше, чем модульных. Учитывая необходимость приложения дополнительных усилий для разработки интеграционных тестов, разработчики пишут тесты только для проверки тех связей, которые им нужны. Далее на диаграмме видно, что системным тестам уделяется еще меньше внимания, чем интеграционным, и еще меньше – тестам, выполняемым вручную.

1.4.5 Почему я предпочитаю модульные тесты?

Как я уже сказал, я отдаю предпочтение модульному тестированию. Я ценю преимущества, которые дают модульные тесты. Их легко писать, они быстрые, их можно писать вперемежку с прикладным кодом и т. д. Я также считаю, что модульное тестирование очень хорошо согласуется с практикой разработки программного обеспечения. Когда разработчик реализует новую функцию, он пишет отдельные модули, которые в конечном итоге будут работать вместе для обеспечения более широкой функциональности. При разработке каждого модуля легко проверить, что он работает должным образом. Тщательно и эффективно тестировать небольшие модули гораздо проще, чем большой функциональный блок.

Кроме того, зная о недостатках модульного тестирования, я тщательно обдумываю, как разрабатываемый модуль будет использоваться другими модулями системы. Соблюдение четких контрактов и их систематическое тестирование дает мне больше уверенности в том, что в комплексе они будут работать безошибочно.

Наконец, учитывая интенсивность тестирования кода с помощью (простых и дешевых) модульных тестов, я могу использовать интеграционные и системные тесты для тех частей, которые действительно важны. Мне не нужно снова тестировать все функции на этих уровнях. Я использую интеграционное или системное тестирование для проверки определенных частей кода, которые, по моему мнению, могут вызвать проблемы интеграции.

1.4.6 Что я тестирую на разных уровнях?

Модульные тесты я использую для проверки модулей, реализующие отдельные алгоритмы или части бизнес-логики программной системы. Большинство корпоративных бизнес-систем реализует преобразование данных. Такая бизнес-логика часто выражается с помощью классов сущностей (например, класса `Invoice` и класса `Order`), обменивающихся сообщениями. Бизнес-логика часто не зависит от внешних сервисов, поэтому ее легко протестировать и проверить с помощью модульных тестов. Модульные тесты дают полный контроль над входными данными, а также полную наблюдаемость с точки зрения подтверждения соответствия поведения ожиданиям.

> **ПРИМЕЧАНИЕ** Если фрагмент кода реализует конкретную бизнес-логику, но не может быть протестирован с помощью модульных тестов (например, бизнес-логику можно протестировать, только когда работает вся система), то, возможно, в свое время были приняты проектные или архитектурные решения, препятствующие разработке модульных тестов. Подходы к проектированию классов оказывают значительное влияние на простоту разработки модульных тестов. Мы обсудим приемы проектирования для поддержки простоты тестирования в главе 7.

Интеграционные тесты я использую всякий раз, когда тестируемый компонент взаимодействует с внешним компонентом (например, с базой данных или веб-службой). Объект доступа к данным (DAO), единственной обязанностью которого является связь с базой данных, лучше тестировать на интеграционном уровне, потому что требуется убедиться, что связь с базой данных работает, SQL-запрос возвращает ожидаемый результат и транзакции фиксируются в базе данных. И снова обратите внимание, что интеграционные тесты обходятся дороже и сложнее в настройке, чем модульные тесты, и я использую их только потому, что это единственный способ протестировать определенную часть системы. В главе 7 я расскажу, как четкое разделение между бизнес-правилами и кодом инфраструктуры помогает тестировать бизнес-правила с помощью модульных тестов, а интеграционный код – с помощью интеграционных.

Как мы уже знаем, системные тесты, находящиеся на вершине пирамиды, очень затратны (их сложно писать, и они медленно выпол-

няются). На системном уровне невозможно повторно протестировать всю систему. Поэтому я должен правильно выбрать, что тестировать на этом уровне, и выполняю простой анализ рисков перед принятием решения. Какие части тестируемой системы являются критическими? Другими словами, в каких частях системы ошибки окажут наиболее существенное влияние? Для проверки этих частей я провожу системное тестирование.

Помните *парадокс пестицидов*: одного метода обычно недостаточно для выявления всех «жучков». Приведу реальный пример из одного из моих проектов. При разработке платформы электронного обучения одной из самых важных функций была оплата. Самыми худшими ошибками являются те, что могут помешать пользователям купить наш продукт. Поэтому мы тщательно протестировали весь код, связанный с оплатой. Мы использовали модульные тесты для проверки бизнес-правил, определяющих, что купил пользователь, разрешения на доступ и т. д. Интеграция с двумя поддерживаемыми нами платежными шлюзами была протестирована с помощью интеграционного тестирования: интеграционные тесты выполняли реальные HTTP-запросы к пробной веб-службе, предоставляемой платежными шлюзами, и мы протестировали покупки различных продуктов различными пользователями с использованием различных кредитных карт. Наконец, наши системные тесты проверили весь путь пользователя к покупке нашего продукта. Эти тесты запускали браузер Firefox, имитировали щелчки на элементах управления HTML, отправляли формы и проверяли доступность выбранного продукта после подтверждения оплаты.

Пирамида на рис. 1.8 включает также ручное тестирование. Выше я говорил, что каждый тест должен быть автоматизирован, но я вижу определенную ценность в ручном тестировании, когда эти тесты сосредоточены на исследовании и проверке. Любому разработчику приятно время от времени использовать и исследовать создаваемую им программную систему как в реальной жизни, так и с помощью тестового сценария. Откройте браузер или приложение и поэкспериментируйте с ним – возможно, вы лучше поймете, что еще нужно протестировать.

1.4.7 Что делать, если вы не согласны с пирамидой тестирования

Многие люди не согласны с идеей пирамиды тестирования и с предпочтительностью модульного тестирования. Эти разработчики выступают за *призовое тестирование* (testing trophy): более тонкий нижний уровень с модульными тестами, более широкий сегмент с интеграционными тестами и более тонкий уровень с системными тестами. Очевидно, что эти разработчики видят наибольшую ценность в написании интеграционных тестов.

Пусть я и не согласен с этим мнением, но понимаю его. Во многих программных системах основная сложность связана с интеграцией компонентов. Представьте распределенную архитектуру микросервисов: в таком сценарии разработчик чувствует себя более комфортно, если автоматизированные тесты выполняют реальные вызовы к другим микросервисам, не полагаясь на заглушки или имитации. Зачем писать модульные тесты для чего-то, что все равно нужно протестировать с помощью интеграционных тестов?

В этом конкретном случае, как сторонник модульного тестирования, я бы предпочел решить проблему тестирования микросервисов так: сначала написал много-много модульных тестов для каждого микросервиса, чтобы убедиться, что все они действуют правильно, и приложил бы все силы для разработки контрактов, чтобы гарантировать, что микросервисы имеют четкие пред- и постусловия. Затем я бы использовал множество интеграционных тестов, чтобы удостовериться, что обмен данными происходит должным образом и что обычные флуктуации в распределенной системе не нарушают работу системы, – да, множество, потому что в этой ситуации их преимущества перевешивают затраты на их разработку. Я мог бы даже потратить время и силы на разработку некоторых интеллектуальных тестов (возможно, управляемых ИИ), чтобы исследовать крайние случаи, незаметные для меня.

Еще один распространенный довод, приводимый в пользу предпочтительности интеграционного тестирования перед модульным, касается информационных систем, ориентированных на базы данных, т. е. систем, основная задача которых заключается в хранении, извлечении и отображении информации. В таких системах сложность зависит от обеспечения успешного прохождения потока информации через пользовательский интерфейс в базу данных и обратно. Подобные приложения часто не реализуют сложных алгоритмов или бизнес-правил. В этом случае действительно предпочтительнее больше внимания уделять интеграционным тестам, чтобы убедиться, что SQL-запросы (которые часто бывают сложными) работают должным образом, и системным тестам, чтобы проверить, что приложение в целом ведет себя так, как ожидалось. Как я уже говорил и буду повторять много раз в этой книге, многое зависит от конкретных обстоятельств.

Я написал большую часть этого раздела от первого лица, потому что он отражает мою точку зрения и основан на моем опыте разработчика. Предпочтение одного подхода перед другим во многом зависит от личного вкуса, опыта и обстоятельств. Вы должны проводить тот тип тестирования, который, по вашему мнению, принесет пользу вашему программному обеспечению. Мне неизвестны какие-либо научные доказательства, говорящие в пользу или против пирамиды тестирования. В 2020 году Трауч (Trautsch) и его коллеги проанализировали возможности обнаружения ошибок на выборке с 30 000 тестами (модульными и интеграционными) и не смогли найти никаких доказательств, что определенные типы дефектов эффективнее обна-

руживаются на том или ином уровне тестирования. У каждого подхода есть свои плюсы и минусы, и вам придется выбирать тот, что лучше подходит для вас и вашей команды.

Я предлагаю познакомиться с мнениями других, выступающих как за предпочтительность модульного тестирования, так и интеграционного:

- в книге «Software Engineering at Google» (Winters, Manshreck и Wright, 2020)[1] авторы отмечают, что в Google часто отдают предпочтение модульным тестам, потому что они, как правило, дешевле и выполняются быстрее. Интеграционные и системные тесты они тоже используют, но в меньшей степени. По словам авторов, около 80 % – это модульные тесты;
- Хэм Воке (Ham Vocke) (2018) выступает в защиту пирамиды тестирования, опубликованной в вики Мартина Фаулера (Martin Fowler);
- сам Фаулер (2021) обсуждает различные формы тестирования (пирамида тестирования и призовое тестирование);
- Андре Шаффер (André Schaffer) (2018) объясняет, почему в Spotify предпочитают интеграционное тестирование;
- Юлия Заречнева (Julia Zarechneva) и голландская компания Picnic (2021) размышляют о пирамиде тестирования.

Размеры тестов, а не их охват

В Google также дают интересное определение *размеров* тестов, которые инженеры учитывают при разработке тестовых случаев. *Небольшой тест* – это тест, который можно выполнить в одном процессе. На такие тесты обычно не влияют основные источники задержек или неопределенности. Другими словами, они выполняются быстро и надежно. *Средний тест* может охватывать несколько процессов, использовать потоки и выполнять внешние вызовы (например, сетевые) к локальному хосту. Такие тесты, как правило, медленнее и менее надежные, чем небольшие. Наконец, *большие тесты* снимают ограничение на отправку вызовов только локальному хосту и по этой причине могут выполнять вызовы к нескольким машинам. Большие тесты в Google используются для комплексного сквозного тестирования.

Также среди многих разработчиков популярна идея классификации тестов не по их границам (модульные, интеграционные, системные), а по скорости выполнения. Однако и в этом случае важна главная цель тестирования – максимизировать его эффективность. Тесты должны быть максимально простыми в разработке и предельно быстрыми в выполнении, а также должны давать как можно больше информации о качестве системы.

---

[1] *Винтерс Титус, Райт Хайрам, Маншрек Том.* Делай как в Google. Разработка программного обеспечения. СПб.: Питер, 2021. ISBN: 978-5-4461-1774-1. – *Прим. перев.*

Большинство примеров кода в оставшейся части этой книги посвящены методам, классам и модульному тестированию, но описанные приемы легко обобщить и на компоненты общего назначения. Например, всякий раз, когда я показываю метод, вы можете думать о нем как о веб-сервисе. Суть от этого не изменится, но вам, вероятно, придется предусмотреть больше тестов, так как ваш компонент будет реализовать более обширную логику.

1.4.8 Поможет ли эта книга найти все ошибки?

Надеюсь, ответ на этот вопрос ясен из предыдущего обсуждения: нет, не поможет! И все же методы, обсуждаемые в этой книге, помогут вам обнаружить множество ошибок – надеюсь, все наиболее важные.

Многие ошибки имеют сложную природу. И для поиска некоторых из них у нас даже нет нужных инструментов. Но мы много знаем о тестировании и как находить различные классы ошибок, и именно на них мы сосредоточимся в этой книге.

Упражнения

1.1 Объясните своими словами, что такое систематическое тестирование и чем оно отличается от несистематического тестирования.

1.2 Келли, очень опытный тестировщик программного обеспечения, был приглашен для тестирования Books! – социальной сети, предназначенной для сопоставления людей по книгам, которые они читают. Пользователи не часто сообщают об ошибках, так как разработчики Books! имеют богатый опыт тестирования. Однако пользователи сообщают, что программное обеспечение не дает обещанного. Какой принцип тестирования применим в этом случае?

1.3 Сюзанна, младший тестировщик программного обеспечения, только что поступила на работу в очень крупную компанию онлайн-платежей в Нидерландах. Ей было поручено проанализировать отчеты об ошибках за последние два года. Занимаясь этим, она подметила, что более 50 % ошибок происходят в модуле международных платежей. Сюзанна пообещала своему руководителю, что разработает тестовые примеры, полностью охватывающие модуль международных платежей, и таким образом найдет все ошибки.

Какой из следующих принципов тестирования может объяснить, почему это невозможно?

A Парадокс пестицидов.
B Исчерпывающее тестирование.
C Раннее тестирование.
D Объединение дефектов.

1.4 Джон свято верит в модульное тестирование. Фактически это единственный тип тестирования, который он проводил во всех проектах, где ему доводилось работать. Какой из следующих принципов тестирования *не* поможет убедить Джона отказаться от своего подхода «только модульное тестирование»?

- **A** Парадокс пестицидов.
- **B** Все зависит от обстоятельств.
- **C** Проблема отсутствия ошибок.
- **D** Раннее тестирование.

1.5 Салли только что начала консультировать компанию, разрабатывающую мобильное приложение, которое помогает людям не отставать от своих ежедневных упражнений. Члены команды разработчиков являются поклонниками автоматизированного тестирования и, в частности, модульных тестов. Их модульные тесты охватывают большую часть кода (> 95 % ветвей), но пользователи по-прежнему сообщают о значительном количестве ошибок.

Салли, хорошо разбирающаяся в тестировании программного обеспечения, объяснила команде принцип тестирования. О каком из следующих принципов она говорила?

- **A** Парадокс пестицидов.
- **B** Всеобъемлющее тестирование.
- **C** Раннее тестирование.
- **D** Объединение дефектов.

1.6 Рассмотрим следующее требование: «Интернет-магазин запускает пакетное задание один раз в день для доставки всех оплаченных заказов. Он также корректирует дату доставки заказов для международных клиентов. Информация о заказах извлекается из внешней базы данных. Оплаченные заказы затем пересылаются во внешний веб-сервис».

Как тестировщик, вы должны решить, какой уровень тестирования (модульное, интеграционное или системное) применить. Какие из следующих утверждений являются верными?

- **A** Интеграционные тесты, хотя и более сложные (с точки зрения автоматизации), окажут больше помощи в поиске ошибок во взаимодействиях с веб-службой и/или с базой данных.
- **B** Учитывая, что модульные тесты проще в реализации (с использованием фиктивных объектов) и охватывают такой же объем кода, что и интеграционные тесты, модульные тесты – лучший вариант для любой ситуации.
- **C** Самый эффективный способ поиска ошибок в этом коде – системные тесты. В этом случае тестировщик должен запустить всю систему и выполнить пакетный процесс. Поскольку этот код легко сымитировать, системные тесты также будут дешевыми.
- **D** Для этой задачи можно использовать все уровни тестирования, однако тестировщики с большей вероятностью найдут больше

ошибок, если выберут один уровень и изучат все возможности и крайние случаи с его использованием.

1.7 Делфтский технический университет (TU Delft) разработал собственное программное обеспечение для расчета заработной платы сотрудников. Приложение использует веб-технологии Java и хранит данные в базе данных Postgres. Приложение часто извлекает, изменяет и вставляет большие объемы данных. Все это осуществляется с помощью классов Java, отправляющих (сложные) SQL-запросы в базу данных.

Как тестировщики, мы знаем, что ошибка может быть где угодно, в том числе и в SQL-запросах. Мы также знаем, что есть много способов исследовать нашу систему. Что из перечисленного ниже не подходит для обнаружения ошибок в SQL-запросах?

A Модульное тестирование.
B Интеграционное тестирование.
C Системное тестирование.
D Стресс-тестирование.

1.8 Выбор уровня тестирования предполагает компромисс, поскольку каждый уровень имеет свои преимущества и недостатки. Что из следующего является основным преимуществом системных тестов?

A Взаимодействия с системой ближе к реальности.
B В среде непрерывной интеграции системные тесты предоставляют разработчикам реальную обратную связь.
C Поскольку системные тесты никогда не бывают нестабильными, они обеспечивают разработчикам более стабильную обратную связь.
D Системный тест пишется владельцами продукта, что делает его ближе к реальности.

1.9 В чем основная причина того, что количество рекомендуемых системных тестов в пирамиде тестирования меньше количества модульных тестов?

A Модульные тесты так же хороши, как системные.
B Системные тесты, как правило, медленные, и их трудно сделать детерминированными.
C Отсутствие хороших инструментов для системных тестов.
D Системные тесты не дают разработчикам достаточно качественной обратной связи.

Итоги

- Тестирование и тестовый код помогают направлять разработку программного обеспечения. Но главная цель тестирования – поиск ошибок, и именно этому в первую очередь посвящена данная книга.

- Систематическое и эффективное тестирование программного обеспечения помогает разрабатывать тесты, проверяющие все закоулки вашего кода и (надеюсь) не оставляющие места для неожиданного поведения.
- Несмотря на систематическое применение тестирования, никогда нельзя быть уверенным, что в программе нет ошибок.
- Всеобъемлющее тестирование невозможно. Работа тестировщика предполагает поиск компромиссов в отношении объема тестирования.
- Программы можно тестировать на разных уровнях, от небольших методов до целых систем с базами данных и веб-сервисами. Каждый уровень имеет свои преимущества и недостатки.