Билл Лабун

Дружеское знакомство

с тестированием программ

bhv®

A Friendly Introduction to Software Testing

Bill Laboon

Билл Лабун

Дружеское знакомство с тестированием программ

Санкт-Петербург
«БХВ-Петербург»
2022

УДК 004.415.53
ББК 32.972
Л12

Лабун Б.

Л12 Дружеское знакомство с тестированием программ: Пер. с англ. — СПб.: БХВ-Петербург, 2022. — 288 с.: ил.

ISBN 978-5-9775-6807-4

Рассмотрены основные понятия и терминология в сфере тестирования и контроля качества ПО. Приведены рекомендации по составлению правил тестирования и отчетов об обнаруженных дефектах. Описано тестирование производительности, безопасности, комбинаторное тестирование. Подробно рассмотрены классы эквивалентности, граничные случаи, угловые случаи, статическое и динамическое тестирование. Даны сведения о проведении приемочного и исследовательского тестирования, описаны средства автоматизации. Отдельные разделы посвящены юнит-тестированию, разработке через тестирование, попарному и комбинаторному, стохастическому тестированию и тестированию на основе свойств.

Для начинающих тестировщиков ПО

УДК 004.415.53
ББК 32.972

Группа подготовки издания:

Руководитель проекта	*Павел Шалин*
Зав. редакцией	*Людмила Гауль*
Перевод с английского	*Игоря Донченко*
Компьютерная верстка	*Ольги Сергиенко*
Оформление обложки	*Карины Соловьевой*

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-1-523-47737-1 (англ.)
ISBN 978-5-9775-6807-4 (рус.)

Оглавление

Глава 1. Введение 11
1.1. История вопроса 11
1.2. Тестирование и обеспечение качества 11
1.3. Что вы найдете в этой книге 12
1.4. Чего нет в этой книге 13
1.5. Замечание по выбору языка программирования 13

Глава 2. Что такое тестирование программного обеспечения? 14
2.1. Определение тестирования программного обеспечения 14
2.2. Верификация и валидация 15
2.3. Предварительное определение дефекта 16
2.4. Пример тестирования в реальной жизни 18

Глава 3. Зачем тестировать программы? 21
3.1. Тестировать или не тестировать 21
3.2. Ни один из разработчиков не совершенен 22
3.3. Обнаружить дефекты раньше, чем позже 22
3.4. Стабильность 23
3.5. Защита пользователя 23
3.6. Независимый взгляд на всю систему 24
3.7. Обеспечивая качество 24
3.8. Риск 25

Глава 4. Основы тестирования 26
4.1. Классы эквивалентности и поведение 26
4.2. Внутренние и граничные значения 28
4.3. Базовые случаи, граничные случаи, угловые случаи 31
4.4. Успешные и неуспешные случаи 32
4.5. Тестирование черного, белого и серого ящиков 32
4.6. Статическое и динамическое тестирование 34

Глава 5. Требования 36
5.1. Тестируемость 39
5.2. Функциональное против нефункционального 42
5.3. Замечание о наименовании требований 43

Глава 6. Тест-планы 44
6.1. Базовая схема тест-плана 44
6.1.1. Идентификатор 45
6.1.2. Тест-кейс (или краткое изложение) 46
6.1.3. Предусловия 46
6.1.4. Входные значения 48
6.1.5. Шаги выполнения 49
6.1.6. Выходные значения 51
6.1.7. Постусловия 51
6.1.8. Ожидаемое поведение и наблюдаемое поведение 51
6.2. Разработка тест-плана 52
6.3. Тестовые фикстуры 54
6.4. Выполнение тест-плана 55
6.5. Отслеживание тестовых прогонов 57
6.6. Матрицы трассируемости 59

Глава 7. Ломая программу 62
7.1. Ошибки, которые следует искать 62
7.2. Список продолжается и продолжается 73

Глава 8. Прохождение тестового плана 74
8.1. Изучение требований 74
8.2. Разрабатывая тест-план 76
8.3. Заполняя тест-план 77
8.4. Определяя фокус 80
8.5. Тест-кейсы для нефункционального требования 84

Глава 9. Дефекты 85
9.1. Что такое дефект? 85
9.2. Жизненный цикл дефекта 86
9.3. Стандартизованный шаблон дефекта 89
9.3.1. Краткое описание 89
9.3.2. Описание 90
9.3.3. Шаги воспроизведения 90
9.3.4. Ожидаемое поведение 91
9.3.5. Наблюдаемое поведение 92
9.3.6. Влияние 92
9.3.7. Серьезность 93
9.3.8. Решение 94
9.3.9. Заметки 95
9.4. Исключения в шаблоне 95
9.5. Примеры дефектов 96

Глава 10. Дымовое и приемочное тестирование 99
10.1. Дымовое тестирование 99
10.2. Приемочное тестирование 101

Глава 11. Исследовательское тестирование 104
11.1. Преимущества и недостатки исследовательского тестирования 104
11.2. Руководство по исследовательскому тестированию 106

Глава 12. Ручное тестирование против автоматизированного тестирования 109
12.1. Преимущества и недостатки ручного тестирования 109
12.1.1. Преимущества ручного тестирования 109
12.1.2. Недостатки ручного тестирования 111
12.2. Преимущества и недостатки автоматизированного тестирования 112
12.2.1. Преимущества автоматизированного тестирования 112
12.2.2. Недостатки автоматизированного тестирования 114
12.3. Реальный мир 115

Глава 13. Введение в юнит-тестирование 117
13.1. Юнит-тестирование: сама идея 117
13.2. Пример на естественном языке 120
13.3. Превратим наш пример в юнит-тест 121
13.3.1. Предусловия 122
13.3.2. Шаги выполнения 123
13.3.3. Утверждения 123
13.3.4. Обеспечение проверки тестами того, что вы ожидаете 124
13.4. Проблемы с юнит-тестированием 126
13.5. Создание тест-раннера 128

Глава 14. Продвинутое юнит-тестирование 131
14.1. Тестовые двойники 131
14.2. Заглушки 135
14.3. Моки и верификация 136
14.4. Фейки 139
14.5. *setup()* и *tearDown()* 140
14.6. Тестирование системного вывода 142
14.7. Тестирование *private*-методов 143
14.8. Структура юнит-теста 145
14.8.1. Основной план 145
14.8.2. Что тестировать? 145
14.8.3. Утверждайте меньше, называйте прямо 146
14.8.4. Юнит-тесты должны быть независимыми 146
14.8.5. Старайтесь сделать тесты лучше каждый раз, когда вы их касаетесь 149
14.9. Покрытие кода 149

Глава 15. Разработка через тестирование 153
15.1. Что такое разработка через тестирование? 153
15.2. Цикл "красный — зеленый — рефакторинг" 156
15.3. Принципы разработки через тестирование 158
15.4. Пример: создание программы FizzBuzz с использованием разработки через тестирование 159
15.5. Преимущества TDD 163
15.6. Недостатки TDD 165

Глава 16. Написание тестируемого кода 167
16.1. Что мы понимаем под тестируемым кодом? 167
16.2. Основные стратегии тестируемого кода 168

16.3. Предусмотрите сценарный интерфейс ... 171
16.4. Написание тестов заранее ... 172
16.5. Пусть ваш код будет DRY ... 172
16.6. Внедрение зависимости ... 174
16.7. Недружественные к тестированию функции и конструкции ... 175
16.8. Работа с чужим унаследованным кодом ... 176
16.9. Заключительные мысли о написании тестируемого кода ... 179

Глава 17. Попарное и комбинаторное тестирование ... 180
17.1. Перестановки и комбинации ... 182
17.2. Попарное тестирование ... 184
17.3. *n*-сторонние взаимодействия ... 188
17.4. Работа с большими наборами переменных ... 189

Глава 18. Стохастическое тестирование и тестирование на основе свойств ... 191
18.1. Бесконечные обезьяны и бесконечные пишущие машинки ... 192
18.2. Тестирование на основе свойств ... 192
18.2.1. Взбираясь по лестнице абстракции ... 193
18.3. Умные, тупые, злые и хаотические обезьяны ... 194
18.4. Мутационное тестирование ... 197

Глава 19. Тестирование производительности ... 202
19.1. Категории показателей производительности ... 203
19.2. Тестирование производительности: пределы и цели ... 204
19.3. Ключевые показатели производительности ... 205
19.4. Тестирование показателей, ориентированных на сервис: время отклика ... 206
19.4.1. Что такое время? ... 207
19.4.2. Какие события следует измерять? ... 209
19.5. Тестирование показателей, ориентированных на сервис: доступность ... 210
19.6. Тестирование показателей, ориентированных на эффективность: пропускная способность ... 215
19.7. Тестирование показателей, ориентированных на эффективность: утилизация ... 217
19.8. Общие советы и рекомендации для нагрузочного тестирования ... 218

Глава 20. Тестирование безопасности ... 220
20.1. Вызовы в тестировании безопасности ... 221
20.2. Основные концепции компьютерной безопасности ... 223
20.3. Распространенные атаки и как использовать тестирование против них ... 226
20.3.1. Инъекция ... 226
20.3.2. Переполнение буфера ... 228
20.3.3. Неправильная настройка безопасности ... 229
20.3.4. Небезопасное хранение ... 229
20.3.5. Социальная инженерия ... 230
20.4. Тестирование на проникновение ... 232
20.5. Общие рекомендации ... 233

Глава 21. Взаимодействие с заинтересованными лицами ... 234
21.1. Кто такие заинтересованные лица? ... 234
21.2. Отчеты и общение ... 236

21.3. Красно-желто-зеленый шаблон 239
21.4. Отчеты о состоянии 242
21.4.1. Пример: ежедневный отчет о состоянии дел для тестировщика программного обеспечения 242
21.5. Замечание об управлении ожиданиями 243
21.6. Замечание о встречах 244
21.7. Разъяснение требований 244
21.8. Этические обязательства 245
21.9. Уважение 246

Глава 22. Заключение 248

Глава 23. Шаблоны тестирования 249
23.1. Шаблон тест-кейса 249
23.2. Шаблон отчета о дефектах 249
23.3. Красно-желто-зеленый шаблон 250
23.4. Ежедневный отчет о состоянии 250

Глава 24. Использование рефлексии для тестирования *private*-методов в Java 251

Глава 25. Что еще почитать 256

Глава 26. Словарь терминов 259

Глава 27. Благодарности 281

Предметный указатель 283

Глава 1

Введение

Признаюсь, я редко читаю вступления к техническим книгам, и поэтому не обижусь, если вы не станете читать это.

1.1. История вопроса

Давным-давно в одной далекой компании я работал ведущим тестировщиком (тест-лидом). Одной из моих обязанностей было проведение собеседований с кандидатами на должность инженера по тестированию в нашу команду, и я понял, что у многих из них нет необходимого понимания тестирования. Те, кто добился какого-то карьерного успеха, зачастую набирались опыта сами по себе по ходу работы. Даже те, у кого было образование в области информационных (или сопутствующих) технологий, зачастую не знали о тестировании программного обеспечения (ПО). Разработчики учились тестированию своего кода, используя подход "новичка" и спрашивая старших разработчиков, что именно необходимо протестировать.

Я понял, что могу либо жаловаться на это, либо попытаться что-то сделать. Мне удалось убедить руководство кафедры вычислительной техники (Computer Science department) Университета Питтсбурга позволить мне разработать и преподавать курс по тестированию программного обеспечения. Этот маленький курс вырос, стал раскрывать многие аспекты качества ПО и в итоге оказался очень популярным, заняв свое место среди дисциплин университета (CS 1632: Software Quality Assurance, если вы захотите принять участие!). Мне пришлось разработать свой учебный план, т. к. мне не удалось найти хорошую книгу или конспект, в которых был бы выдержан баланс между теорией и практикой. Отыскать хорошую книгу оказалось даже более сложной задачей, чем выбрать при собеседовании хорошего инженера по обеспечению качества (Quality Assurance, QA)! И снова я понял, что могу либо жаловаться на это, либо попытаться что-то сделать, и я выбрал последнее.

1.2. Тестирование и обеспечение качества

Тестирование является важной частью процесса разработки ПО и полезно не только для тех, кто собирается сделать карьеру в области QA. Разработчик, не заботя-

щийся о качестве ПО, не хороший разработчик. Эта книга предназначена для тех, кто интересуется тестированием ПО или написанием тестов как разработчик.

1.3. Что вы найдете в этой книге

Целью этой книги является сравнительно достаточный обзор тестирования ПО. Я надеюсь, что после ее прочтения у читателя будут все знания и навыки, необходимые для включения в процесс обеспечения качества, и я также рассчитываю на то, что менеджеры, программисты и все, кто связан с разработкой ПО, могут найти ее интересной.

Именно поэтому книга начинается с общего описания предметной области — что такое тестирование программного обеспечения, в конце концов?! Довольно сложно изучать предмет без понимания, что это! Затем мы перейдем к теории и терминологии, используемым теми, кто работает в индустрии тестирования ПО. Открою вам маленький секрет — пожалуй, это наименее интересная часть книги. Тем не менее мы должны говорить на одном языке при обсуждении основ. Наверное, было бы трудно объяснить принцип работы водопровода, если бы нам пришлось отказаться от использования слова "труба" из опасения, что кто-то не понимает его.

Далее мы перейдем к основам разработки ручных тест-планов и к обработке дефектов, найденных в процессе их выполнения. Ручные тесты сейчас используются гораздо меньше, чем раньше; автоматизированные тесты освободили нас от скуки, связанной с их выполнением. Тем не менее существуют определенные преимущества разработки тестов без необходимости беспокоиться о синтаксисе какого-либо языка программирования или наборе инструментов. Мы также посмотрим, как правильно описывать наше тестирование и сообщать о дефектах, найденных во время его проведения.

Так как ручное тестирование несколько устарело, мы перейдем к автоматизированным тестам — тестам системного уровня и юнит-тестам. Автоматизация позволяет вам выполнять тесты очень быстро как на низком уровне (например, позволяя убедиться, что алгоритм сортировки работает правильно), так и на высоком (путем добавления чего-либо в корзину вашего интернет-магазина). Если вы когда-либо занимались выполнением ручных тест-планов, вы поймете, что передача компьютеру выполнения всех ваших тестов освободит для вас немало времени. Но, пожалуй, гораздо более важно то, что это также освободит вас и от гнева, т. к. постоянное выполнение ручных тестов является самым быстрым способом вывести тестировщика из себя.

И в итоге мы придем к по-настоящему интересным вещам! Это та часть книги, в которой вы прочитаете о таких специализированных видах тестирования, как комбинаторное тестирование, тестирование производительности и тесты безопасности. Мир тестирования ПО довольно большой, и тестирование встроенного ПО сильно отличается от тестирования веб-приложений, тестирование производительности отличается от функционального тестирования, а тестирование первого про-

дукта-прототипа, выпущенного стартапом, не похоже на тестирование медицинского устройства, выпускаемого большой компанией.

1.4. Чего нет в этой книге

Эта книга — ознакомительный текст о мире тестирования; и вам откроются двери в огромное пространство для исследований, если вы решите углубиться в рассматриваемые здесь вопросы. Однако нашей целью является не соревнование с "Войной и миром" в объеме, а общее введение в тематику. Потратить много времени на сложные разделы каждого из вопросов может быть интересно мне и еще ограниченному количеству читателей книги, но моими целями являются создание хорошего основания для знаний по практическому тестированию ПО и знакомство с некоторыми особенностями и аспектами этой индустрии. Рассматривайте книгу как обзорную экскурсию по Европе, а не год жизни в Париже.

1.5. Замечание по выбору языка программирования

Примеры, которые я привожу, написаны на языке программирования Java и используют связанные с ним инструменты (например, JUnit). Это не связано с какой-то особенной любовью к Java. Существуют другие языки, о которых говорят, что они легче в использовании или более удобны для минимизации количества дефектов. Но Java сегодня является *lingua franca*[1] для разработчиков. Это стандартный, популярный и много почерпнувший из ALGOL язык программирования, и даже если вы не очень знакомы с ним, вы, возможно, сможете разобраться в том, что мы будем делать.

[1] Универсальным языком. — *Прим. ред.*

Глава 2

Что такое тестирование программного обеспечения?

Начнем с того, чем оно не является.

1. Это не поиск всех без исключения дефектов.
2. Это не случайные нажатия по клавиатуре в надежде, что что-то сломается.
3. Даже не надейтесь, что что-то сломается. Точка.
4. Это не то, чем начинают заниматься после завершения программирования.
5. Это совсем, СОВСЕМ не то, что можно отложить до того момента, как пользователи начнут жаловаться.

2.1. Определение тестирования программного обеспечения

На высоком уровне тестирование ПО является способом предоставления оценки качества программного обеспечения заинтересованным лицам (т. е. людям, напрямую заинтересованным в работе системы, — потребителям, пользователям и менеджерам). Хотя эти лица могут напрямую не беспокоиться о качестве ПО, они заинтересованы в управлении риском. Данный риск может принимать разнообразные формы. Для покупателей использование ПО несет риски потери данных, риски прекращения деятельность, риски того, что использование ПО принесет больше убытков, чем прибыли. Для внутренних заинтересованных лиц риски могут включать задержки с релизом ПО (или его невыходом вообще), потерей клиентов из-за выпуска некачественного ПО или даже судебные иски. Благодаря тестированию ПО вы сможете лучше оценить, какие риски несут заинтересованные лица.

Тестирование ПО позволяет непредвзято взглянуть на программный продукт. Разработчики известны тем, что — осознанно или неосознанно — относятся довольно просто к коду, который они пишут. Хотел бы я сказать, что, когда сам пишу код, избегаю этого и фокусируюсь на качестве ПО. Тем не менее зачастую мое общение с тем, кто тестирует мою программу, проходит примерно так:

> *Я*: "Моя процедура вычисления квадратного корня работает! И работает в два раза быстрее, чем раньше!"

Тестировщик: "Хм... Когда я ввожу букву вместо числа, программа перестает работать".

Я: "О, я уверен, что никто не будет вводить буквы".

Тестировщик: "Когда я ввожу отрицательное число, программа перестает работать".

Я: "Да, мы не поддерживаем работу с комплексными числами. Поэтому я не делаю такую проверку".

Тестировщик: "Я ввожу 2.0 и получаю сообщение об ошибке, что программа не может работать с дробными числами".

Я: "Да, по идее, программа должна работать, но в данный момент она принимает в качестве данных только целые числа. Но пользователь должен знать об этом".

Тестировщик: "Хорошо, когда я ввожу 2, экран заполняется цифрами, идущими после запятой..."

Я: "Да, конечно! Квадратный корень двойки является иррациональным числом, поэтому расчет будет идти до тех пор, пока существует Вселенная! Просто введи любое положительное число, возведенное в квадрат".

Тестировщик: "Когда я печатаю 25, то программа выдает мне 3".

Я: "Да, пожалуй, здесь ошибка. Я тестировал программу только с 9, и она прошла все мои тесты!"

(...И так далее.)

Помните, что я тот, кто читает лекции по тестированию ПО. И даже я не могу не любить эти маленькие бедные функции, которые пишу. И работа тестировщика заключается в том, чтобы выбить из меня эту родительскую любовь. Я могу вырастить свою маленькую бедную функцию, но не могу предсказать, как она поведет себя, оказавшись в сложной ситуации.

2.2. Верификация и валидация

Тестирование также должно гарантировать, что создано программное обеспечение требуемого качества. Представьте себе такой разговор между менеджером проекта и пользователем.

Менеджер проекта: "Я прошелся по всему проекту. Криптографический движок просто пуленепробиваемый, рекордно быстрый и использует 8192-битное кодирование — ваши секреты будут в безопасности триллион лет".

Пользователь: "На самом деле, я всего лишь хотел поиграть в пасьянс..."

Можно ли сказать, что программа удовлетворяет требованиям пользователя? Конечно, нет. Даже если программа удовлетворяет всем предъявляемым к программам требованиям, не падает, дает правильные ответы и т. д., но при этом не отвечает ожиданиям пользователя, она не будет успешной.

Это иллюстрирует разницу между **верификацией** и **валидацией**. Верификация гарантирует, что вы создаете программу правильно; валидация гарантирует, что вы создаете правильную программу. Другими словами, верификация гарантирует, что система не ломается, она отвечает необходимым требованиям, обрабатывает ошибки корректно и т. д. Валидация обеспечивает соответствие требований реальным ожиданиям потребителя: выполняет ли программа то, что нужно пользователю? Существуют ли какие-то недочеты в требованиях, ведущие к тому, что даже если программа удовлетворяет им всем, пользователь все рано останется недовольным?

И верификация, и валидация являются частью процесса тестирования ПО. Хотя большинство тестировщиков тратят бóльшую часть своего времени на верификацию, тестировщик не относится формально к тому, что программа соответствует требованиям. Тестировщиков можно представить как защитников интересов пользователя, даже если это противоречит интересам других заинтересованных лиц — необходимо создавать продукт, который соответствует пожеланиям пользователей, а не просто пытаться выпустить программу.

Интересно, что чем больше времени и ресурсов тратится на исправление дефектов или улучшение качества каким-либо образом, тем больше средств будет сэкономлено в долгосрочной перспективе. Работа с системой, в которой мало дефектов или, по крайней мере, существуют дефекты, о которых вы знаете, окажется значительно более легкой, чем когда вы захотите добавить новые функции в программу, которая периодически перестает работать и делает это как будто случайным образом. Система, проверяемая хорошим набором автоматизированных тестов, позволит вам вносить изменения и быть уверенным, что вы не создали новые дефекты в процессе разработки. Это парадокс качества программного обеспечения — вы можете потратить *меньше* денег и времени на создание *лучшего* продукта путем правильного тестирования.

2.3. Предварительное определение дефекта

Важно понимать, что не всякая найденная в системе проблема становится дефектом. Дефект является проблемой, которая либо нарушает функциональность системы в ее текущем понимании, либо не соответствует требованиям программы. Если программа работает нормально и соответствует всем требованиям, значит, у нее нет дефектов. Если программа не соответствует требованиям или не работает нормально, значит, можно говорить о том, что обнаружен дефект.

Например, рассмотрим компанию, создающую совершенно новую версию игры "крестики-нолики". Требования следующие:

1. Игровая доска должна быть три на три квадрата, всего девять смежных квадратов.
2. Первый игрок может поставить значок "X" (крестик) в один любой квадрат.
3. Второй игрок может поставить значок "O" (нолик) в любой открытый (т. е. еще не занятый крестиками или ноликами) квадрат, тем самым завершая первый ход первого игрока.

4. Затем игроки должны по очереди размещать крестики и нолики (первый игрок, а затем соответственно второй игрок) в свободные квадраты. Игра идет либо до тех пор, пока не останется незаполненных квадратов, и при этом ни в одном ряду, колонке или диагонали не окажутся проставленые значки одного типа (в этом случае объявляется ничья); либо до тех пор, пока целый ряд, колонка или диагональ не окажутся заполненными значками одного типа, и в этом случае выставлявший значки данного типа (крестик для первого игрока, нолик для второго) оказывается победителем, а его соперник проигравшим.

Это довольно понятные правила игры в крестики-нолики. Теперь давайте рассмотрим случай, когда первый игрок, который должен поставить значок "X", начинает игру с "O". Это дефект, потому что это нарушает требование 2. Даже если продолжить игру (скажем, второй игрок поставит "X"), все равно это дефект, потому что это нарушает требование.

Теперь давайте представим, что после бета-тестирования пользователь заявляет, будто игра нечестная, потому что она вынуждает игрока использовать "X", а ему этот значок не нравится. Пользователь предлагает заменить "X" на "W", потому что последняя является гораздо более красивой буквой. Это дефект или улучшение?

Это **улучшение**, потому что система соответствует всем требованиям и работает нормально. То, что пользователю не нравится значок, не является дефектом! И очень важно внести эти изменения — возможно, даже гораздо более важно, чем исправлять существующие дефекты. Улучшения не являются плохими, бесполезными или каким-то видом жалоб, они просто предполагают модификацию существующих требований системы.

Другим примером дефекта стало бы то, если бы игровое поле исчезло после того, как игрок поставил значок в центральный квадрат. Нет никаких особенных требований по этому поводу, но существуют изменяющиеся "неявные требования" к программам, такие как запрет на аварийное завершение (им нельзя вылетать) и зависание, они должны выводить на экране актуальное изображение, реагировать на воздействие и т. д. Эти неявные требования могут изменяться в зависимости от вида системы: например, видеоигра должна реагировать на воздействие в течение 99% всего времени, в то время как для программы прогноза погоды (в которой данные обновляются каждые 30 минут) "реагирование" заключается в том, чтобы просто вывести необходимую информацию.

Могут возникнуть разногласия по поводу того, является ли текущая проблема дефектом или улучшением. Основная причина этих разногласий — как раз эти неявные требования. Если персонаж видеоигры реагирует спустя три секунды после нажатия клавиши, можно сказать, что это довольно долго, даже если нет отдельных требований по производительности. Игроку не очень понравится каждый раз ждать по три секунды. Но какое ожидание после нажатия клавиши является приемлемым? Две секунды? Одна? Сто миллисекунд? Подобным образом можно задаться вопросом: допустимо ли для программы зависать и терять данные, если в системе закончилась память? Для единственного запущенного приложения в вашем телефоне — возможно. Это допустимо принять как достаточно редкое событие, причи-

няющее небольшой урон обычному пользователю, поэтому добавление обработчика подобной ситуации станет улучшением. Для компьютера-мейнфрейма, на котором запущено ПО, работающее с банковскими переводами, это определенно окажется дефектом. Предотвращение потери данных, даже если это не указано явно в требованиях, является крайне важным для этой области. Понимание **тестируемой системы** и области ее применения позволяет вам применять **интуитивное тестирование** ("seat of your pants" testing), т. е. тестирование поведения без формального описания, но основанное на вашем знании системы и области ее работы.

В некоторых случаях различие между дефектом и улучшением может оказаться очень значительным. Если ваша компания разрабатывает авиационный софт для нового истребителя, то очень вероятно, что у вас в скрупулезно рассматривается, что именно является улучшением, а что — дефектом. В таком случае имеются подготовленные требования, принимающие решения арбитры и люди, чьей работой является проектирование и разъяснение требований. Если компания подписала контракт для разработки программы и тщательно придерживается каждой буквы этого контракта, то ее сотрудники будут на всякое обращение заказчика отвечать, что это не дефект, а нечто не покрытое требованиями, а значит, улучшение.

В других случаях граница между дефектами и улучшениями оказывается размытой. Давайте предположим, что вы работаете на некий стартап, в котором недостаточно средств на разработку программного обеспечения, а единственным реальным требованием выступает негласное правило "делайте то, что хочет клиент, или мы банкроты". В этом случае, если клиент хочет что-то, значит, над этим нужно работать без каких-либо вопросов.

Хотя решение, над каким дефектом или улучшением следует работать, почти всегда находится в сфере управления проектом, а не обеспечения качества, взаимодействие с командой QA часто будет полезным. Тестирование ПО всегда позволяет определить влияние найденных дефектов, а также потенциальные риски исправления дефектов, внесения улучшений или же просто найти обходные пути. Обладание этим знанием поможет менеджерам проекта принимать осознанные решения о направлении развития продукта.

2.4. Пример тестирования в реальной жизни

Представим, что мы тестируем новую программу "Уменьшатель", которая берет строку и делает ее буквы строчными. Заказчик не предоставил никакого дополнительного описания задачи, потому что она кажется очевидной — входной текст может быть в нижнем регистре, а может и не быть, данные на выходе должны быть той же строкой, но все прописные буквы должны стать строчными. Метод, решающий эту задачу в программе, имеет следующую сигнатуру:

```
public String lowerify(String S)
```

Заказчик настаивает, что больше ничего не нужно знать, чтобы начать тестирование. Если бы вам было поручено такое тестирование, какие бы вопросы вы задали

для того, чтобы создать тест-план? Другими словами, какие требования вы бы постарались вытянуть из заказчика?

1. Какая кодировка символов будет использоваться — UTF-9, ASCII, EBCDIC или что-то другое?
2. Какая максимальная длина обрабатываемых данных предполагается? То, что хорошо работает на нескольких словах, может начать вести себя по-другому, если на вход подать несколько терабайт текста.
3. Что должно происходить, если входной текст написан на языке, отличном от английского? И что делать, если в этом языке нет понятия прописных и строчных букв?
4. Что должна делать программа, если во время ее работы пользователь нажмет комбинацию клавиш <Ctrl>+<C> или выполнит любую другую команду остановки программы?
5. Должна ли эта программа уметь читать данные из сети? Если да, то что нужно делать в случае проблем с сетью — попытаться прочитать данные снова, прекратить работу, показать ошибку или что-то еще?

Уверенность в том, что у вас есть правильные ответы на эти вопросы, является частью валидации программы. Вы хотите тестировать не то, что пользователь хочет получить от программы.

Вы разобрались, что хочет заказчик, но тем не менее еще остается над чем поработать. Вам нужно убедиться, что программа работает в нормальных условиях и обрабатывает разнообразные входные данные. Вот несколько вариантов входных данных, позволяющих протестировать работу программы в различных случаях:

1. Строка со всеми прописными буквами, например `"ABCDEFG"`.
2. Строка, в которой все буквы уже переведены в нижний регистр, например `"lmnop"`.
3. Строка с небуквенными символами, например `"78&^%0()[]"`.
4. Строка, в которой смешаны буквы в верхнем и нижнем регистрах, например `"VwXyZ"`.
5. Строка со спецсимволами, такими как возврат каретки и нулевой символ, т. е. `\r\n\o`.
6. Пустая строка.
7. Очень длинная строка — скажем, текст из большой книги, оцифрованной в рамках "Проекта „Гутенберг“".
8. Исполняемый код.
9. Бинарные данные.
10. Строка, внутри которой встречаются маркеры конца файла (EOF).

Можете ли вы придумать какие-либо другие возможные входные данные, которые способны вызвать ошибку или неправильный результат? Внешние факторы также могут рассматриваться.

Что произойдет, если...

1. У системы закончится память во время обработки текста?
2. Процессор обрабатывает множество различных процессов, и система из-за этого не отвечает на запросы?
3. Сетевое соединение оборвано в самый разгар обработки данных?

Важно отметить, что практически невозможно **исчерпывающе протестировать** все комбинации входных данных. Даже если бы мы собирались протестировать входные данные, ограниченные 10 символами из цифр и букв, и проигнорировать все внешние факторы, тогда бы у вас получилось более трех квадриллионов тест-кейсов. Так как строки могут быть произвольной длины (их ограничивает объем памяти компьютера), и существует множество внешних факторов, которые можно было бы учесть, выполнение всеобъемлющего тест-плана для этой функции заняло бы миллиарды лет! Даже из этого простого примера легко понять, что тестирование может быть очень сложным процессом, полным неопределенности и трудных решений о том, на чем следует сфокусироваться. Тестировщику предстоит не только решать эти неопределенности, но и устанавливать, сколько усилий и времени следует тратить на них. Чем больше усилий вы вложите в какую-то часть тестирования, тем меньше времени останется на другое. Помните это, разрабатывая тестовую стратегию, — время, за которое вы должны выполнить проект, может изменяться, но оно всегда конечное, и всегда существуют различные приоритеты, которыми вам придется жонглировать, чтобы обеспечить качество ПО.

Помните, что причиной проведения тестирования ПО являются оценка и, если возможно, уменьшение риска для заинтересованных лиц. Понимание возможных рисков само по себе может уменьшить риск. В конце концов, непротестированное программное обеспечение, которое никогда не запускалось, может быть идеальным (теоретически, по крайней мере) или не работать совсем. Тестирование помогает рассчитать, где между этими двумя экстремумами *на самом деле* находится программное обеспечение. Это поможет нам понять, являются ли проблемы с ПО тривиальными, или же из-за них следует отложить выпуск программы, потому что бóльшая часть функционала не работает. Помогая определить уровень риска, тестировщики позволяют другим участвующим заинтересованным лицам принять соответствующие решения.

Глава 3

Зачем тестировать программы?

Теперь, когда мы знаем, что такое тестирование программного обеспечения, имеет смысл спросить: почему кто-то может захотеть заниматься этим? В конце концов, вы определенно добавляете дополнительную работу, т. к. придется писать тесты. Также вам нужно будет убедиться, что созданный вами код может быть протестирован, и, возможно, создать тестовые фреймворки для своей системы. В зависимости от того, как вы тестируете, не исключено, что вам потребуется изучить другие фреймворки или даже языки программирования.

3.1. Тестировать или не тестировать

Давайте представим, что вы стали главой компании Rent-A-Cat, Inc. Подающий надежды молодой менеджер проекта подбегает к вам в коридоре, с его прекрасно уложенных волос капает пот, а сам он вцепился в распечатку из Excel.

"Мэм (или сэр)! — кричит менеджер проекта. — Я открыл способ снизить расходы нашего проекта на десятки тысяч долларов! Всё, что необходимо, — это убрать связанные с тестированием ресурсы из команды. У меня отличные разработчики программ, и они никогда не сделают ошибку. Таким образом, мы, в конце концов, сможем купить ту самую позолоченную раковину для туалета руководства!"

Здесь у вас есть два варианта на выбор:

1. Неистово захохотать и представить ощущение дистиллированной воды, текущей на ваши наманикюренные руки в этой единственно подходящей для вас царской раковине.
2. Объяснить менеджеру проекта причины тестирования и доводы, и почему важно тестировать ПО перед его выпуском.

Поскольку вы читаете книгу по тестированию ПО, я предполагаю, что вы выберите второй вариант. Хотя, на первый взгляд, вы могли подумать, что имело бы смысл не тестировать вашу программу. С организационной точки зрения, фирма существует для того, чтобы принести доход своим владельцам. Если вы можете снизить стоимость разработки программы путем исключения части процесса разработки, тогда вы заставите основательно задуматься владельцев фирмы над тем, имеет ли смысл содержать команду разработчиков.

Помните, что глава компании отвечает за всю организацию, а не только за инжиниринг, IT или любой другой отдел. Даже для технической компании отдел тестирования в большинстве случаев будет сравнительно крохотным. И при этом другие отделы будут просить ресурсы, часто обоснованно. Маркетинговый отдел запросит больше маркетологов, ведь компания, которая не генерирует новые идеи для продуктов, однажды разорится. Отделу техподдержки требуются сотрудники горячей линии, ведь если люди не разберутся, как пользоваться продуктом, они прекратят покупать его, и компания разорится. Все эти и другие отделы компании могли бы использовать финансовые ресурсы, которые в настоящее время идут персоналу, занятому тестированием.

Но даже с учетом вышесказанного у вас, как главы компании, найдется множество причин отклонить план менеджера проекта.

3.2. Ни один из разработчиков не совершенен

Поднимите руку, если вы когда-либо писали неправильный программный код. Если ваша рука поднята, тогда вы уже знаете, зачем нужно тестировать программы. (Если ваша рука опущена, тогда я могу предположить, что вы никогда ранее не программировали.) Неплохо помнить, что разработка программного обеспечения является одним из самых интеллектуально сложных процессов, которым занимаются люди, и при этом максимально задействуются способности человеческого разума. И если так рассуждать, ошибка на единицу при получении индекса может показаться не такой уж серьезной.

Согласно данным Национального института стандартов и технологии (National Institute of Standards and Technology), ошибки в ПО стоили американской экономике примерно 60 млрд долларов в 2002 году[1]. Это примерно 0,6% внутреннего валового продукта страны. И так как пишется всё больше и больше программ, а наша повседневная жизнь всё больше и больше оказывается завязаной на них, то эта цифра наверняка значительно выросла к сегодняшнему дню. Даже если рассматривать низкое значение 2002 года, можно понять, что дефекты в программах вызвали проблемы, стоимость которых оказалась примерно равна трети американского сельскохозяйственного производства.

3.3. Обнаружить дефекты раньше, чем позже

«Золотое правило» тестирования гласит, что вы должны обнаружить дефекты настолько раньше, насколько возможно. Если вы найдете проблему в программе сравнительно рано, в большинстве случаев разработчику достаточно будет внести простое исправление, и никто вне команды не узнает, что программа падала, когда вводились цифры вместо имени. Если с подобным дефектом пользователь столк-

[1] The Economic Impact of Inadequate Infrastructure for Software Testing, National Institute of Standards and Technology, 2002. См. **http://www.nist.gov/director/planning/upload/report02-3.pdf**.

нется во время эксплуатации программы, последствия могут оказаться куда более плачевными — компания потеряет деньги, т. к. приложение будет неработоспособным, а потребители разочарованными. Кроме того, заниматься исправлениями на этом этапе окажется гораздо сложнее и дороже. Программисты в состоянии стресса попытаются "заклеить скотчем", т. е. решить проблему как можно быстрее, и не станут заниматься надежными исправлениями. Подобно заклеиванию протекших труб скотчем, это решение окажется худшим из возможных.

Тестирование программ позволяет вам найти ошибки еще до того, как пользователи увидели их, а ваша команда сможет эффективнее с ними справиться, уменьшив в конечном итоге влияние этих ошибок.

3.4. Стабильность

Разработка программного обеспечения является сложным процессом. Иногда в некоторых компаниях она может даже превратиться в хаос. Команда тестировщиков может облегчить ситуацию, обеспечивая стабильность. Подстраховка разработчиков в том, что они продолжают работу над содержащей меньше дефектов программой, позволит этой программе стать более устойчивой. Дополнительный функционал маловероятно может быть построен на основе ненадежного фундамента.

Команда обеспечения качества может также следить за существующими дефектами и осуществлять расстановку приоритетов. Разработчикам и менеджерам не следует бросать всё и заниматься только что обнаруженными дефектами или, наоборот, игнорировать новые проблемы и акцентироваться на старых — команда тестировщиков сможет определить, чем следует заниматься в текущий момент. Это делает процесс разработки более выверенным и равномерно идущим.

3.5. Защита пользователя

Программисты, менеджеры и все, кто работает над созданием программ, очень часто имеют свои причины для работы над проектом помимо того, что им платят за эту работу. Программисты могут хотеть изучить другой язык программирования; дизайнерам будет интересно попробовать создать принципиально новый пользовательский интерфейс; менеджеры проекта могут попробовать организовать свою работу так, чтобы успевать всё в срок. История разработки программного обеспечения просто замусорена проектами, которые были технически интересны или выпущены в срок, но при этом не отвечали требованиям пользователей.

У QA-инженеров есть особая роль — они действуют как представители потребителей и пользователей программы. Эта роль позволяет убедиться, что потребители получат программу высокого качества, которая им нужна. Фактически в некоторых организациях у тестировщиков есть полномочия остановить релиз или переопределить ресурсы для того, чтобы пользователь в итоге получил необходимую ему программу, которая разработана нужным образом. Эти полномочия зависят от области, в которой работает компания; в компаниях, создающих критически важные или

жизненно важные программы, у тестировщиков обычно больше полномочий. Но вне зависимости от размера компании или области, в которой она действует, наличие кого-то, чья работа заключается в представлении интересов заказчика, является мощной силой. Использованная с умом, она поможет создать программу, которая приведет пользователя в восхищение.

3.6. Независимый взгляд на всю систему

Разработчики сфокусированы на малой части системы, с которой они близко знакомы. Редко кто из разработчиков понимает функционирование всей разрабатываемой системы. У тестировщиков может не быть глубокого понимания каждой составляющей программы, но они, как правило, имеют широкое видение всей системы. Они тестируют различные возможности программы, используют новую функциональность и устанавливают программу в различных системах.

Такой охват всей системы оказывается ценной противоположностью глубокому погружению в какую-то одну из составляющих программы. Понимание, как совместно работают различные подсистемы и как реализована функциональность с точки зрения пользователя, позволяет сотрудникам QA предоставить информацию о статусе системы *как системы*. Это также делает тестировщиков ценным ресурсом при добавлении новой функциональности или модификации существующей, т. к. они будут понимать, как это может повлиять на систему в целом.

В то время как другие заинтересованные лица будут напрямую заинтересованы в разработке программы, роль тестировщика будет заключаться в обеспечении независимого взгляда на программу. Рассматривая систему изолированно и без влияния (вольного или невольного) разработчиков и менеджеров, тестировщики ПО могут предоставить более реалистичный и объективный статус системы.

3.7. Обеспечивая качество

Хотя тестирование ПО может предоставить вам множество ценных преимуществ, оно не является единственным способом улучшить качество вашей программы. В одной из самых авторитетных книг по написанию программ "Совершенный код" Стива Макконнела (Steve McConnell "Code Complete") приведена оценка нахождения дефектов разработчиками, использующими различные техники. Анализ кода (code review), формальные инспекции, моделирование программного обеспечения способствуют улучшению качества программы. **Парное программирование**, когда два человека работают одновременно за одним компьютером, также показало значительное улучшение качества программы. Так как человеку довольно легко пропустить свои ошибки, другой человек, изучающий код или текст независимо, довольно часто замечает, что что-то не так. Я знаю, о чем говорю, — я пропустил множество нелепых опечаток в этой книге, которые были обнаружены сразу же, как я отправил книгу на редактирование[1].

[1] См. **https://github.com/laboon/ebook/pull/15/**.

Хотя качество означает не только "уменьшение количества дефектов", это определенно является важной составляющей. Выявляя дефекты, тестировщики улучшают качество программы. Пользователи программы получают лучший продукт.

Выделение соответствующего времени на тестирование, программирование и прочие составляющие цикла разработки программного обеспечения также поможет повысить качество. Очень немногие части программы написаны безупречно, а созданная в условиях недостатка времени программа окажется неприемлемого качества. Предоставление адекватного времени и ресурсов инженерам для разработки программы в целом повысит уровень качества.

Выбор языка программирования, фреймворка и дизайн программы также могут оказать большое влияние на качество программы. Хотя у каждого языка есть свои сторонники, определенно существуют причины, по которым большинство веб-приложений не пишется на ассемблере, а встраиваемое ПО не создается на Ruby. Разные языки, библиотеки и другие средства разработки имеют различные преимущества, и использование тех, что подходят для вашей создаваемой системы, принесет свои дивиденды в терминах качества.

Хотя эта книга фокусируется на тестировании, следует признать, что оно выступает только одной из составляющих качественного ПО. Качество в современном проекте по разработке ПО является обязанностью всех участников команды, не только тестировщиков.

3.8. Риск

Сущность тестирования сводится к минимизации рисков для всех вовлеченных: потребителей, пользователей, разработчиков и т. д. Независимое тестирование ПО позволит провести объективный анализ качества системы. Оно снижает риск путем предоставления информации о статусе системы как на высоком уровне (т. е. "система готова к релизу"), так и на низком (т. е. "если имя пользователя содержит символ !, то система сломается"). Разработка ПО является сложным и рискованным процессом. И если руководитель компании хочет убедиться, что риски сведены к минимуму, необходимо, чтобы тестировщики ПО являлись частью команды.

Глава 4

Основы тестирования

Перед тем как мы погрузимся в написание тестов, нам надлежит убедиться, что мы с вами одинаково понимаем теорию и терминологию тестирования. В этой главе мы расширим свой словарный запас и получим теоретическую основу для обсуждения тестирования.

4.1. Классы эквивалентности и поведение

Представьте, что вам необходимо протестировать новый дисплей для датчика давления автомобильных колес. Давление считывается с внешнего датчика, и гарантируется, что значение давления будет передано на наш 32-битный дисплей. Если давление больше 35 фунтов на квадратный дюйм (pounds per square inch, PSI), должен загореться сигнал "ИЗБЫТОЧНОЕ ДАВЛЕНИЕ", а все остальные сигналы должны быть отключены. Если давление находится в пределах от 0 до 20 PSI, должен загореться сигнал "НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ", и все остальные сигналы должны быть отключены. Если значение давления оказывается отрицательным, должен загореться сигнал "ОШИБКА", и все остальные сигналы должны быть отключены.

Этот тест должен быть довольно простым. Имеется только одно входное значение, его тип известен, и выходные значения тоже известны. Мы исключаем экзогенные факторы, хотя тестировщику "железа" будет интересно узнать, что случится, если, скажем, провод между датчиком и дисплеем оборвется, или произойдет скачок напряжения, или... в общем, используйте свое воображение.

С чего надо начать подобное тестирование? Вам нужно подготовить некоторые входные значения и ожидаемые выходные (например, "отправить 15 PSI → увидеть, что горит сигнал „НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ“ и все остальные сигналы выключены"). Вы можете исполнить тест и посмотреть, совпадает ли то, что происходит, с тем, что вы ожидали увидеть. Это суть тестирования — сверка **ожидаемого поведения** с **наблюдаемым поведением**, т. е. обеспечение того, что программа делает именно то, что вы ожидаете от нее в определенных обстоятельствах. Можно внести корректировки, дать советы и предостережения, но основа всего тестирования заключается в сравнении ожидаемого поведения с наблюдаемым.

Ваш менеджер хотел бы протестировать эту систему как можно быстрее и поручает вам создать четыре теста. Вооруженные знанием, что вам нужно сравнить ожидаемое поведение с наблюдаемым, вы решаете отправить значения –1, –111, –900 и –5, чтобы увидеть сигнал "ОШИБКА" в каждом случае, и при этом остальные сигналы не должны загораться. Волнуясь от того, что вы написали свои первые четыре теста, вы показываете их менеджеру, который хмурится и говорит: "Ты тестируешь только один класс эквивалентности!"

Класс эквивалентности (или эквивалентное разбиение) является набором входных данных, которые соответствуют одному выходному значению. Вы можете представить их как различные "группы" входных значений, которые делают что-то схожее. Это дает возможность тестировщикам создавать тесты, которые покрывают все составляющие функциональности и позволяют избежать избыточного тестирования только одной части (как в вышеприведенном примере, где класс эквивалентности "ОШИБКА" был протестирован четыре раза, в то время как другие ни одного).

Какие другие классы эквивалентности имеются в данном случае? Для того чтобы найти ответ, представьте все возможные варианты, которые вы можете получить на выходе:

1. Сигнал "ОШИБКА" загорается для PSI, равного –1 или меньше.
2. Сигнал "НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ" загорается для PSI между 0 и 20 включительно.
3. Сигналы не загораются для PSI между 21 и 35 включительно — нормальные условия работы.
4. Сигнал "ИЗБЫТОЧНОЕ ДАВЛЕНИЕ" загорается для PSI от 36 и выше.

Математически вы можете связать группу входных значений и ожидаемое состояние на выходе:

1. [MININT, MININT + 1, ..., –2, –1] → только сигнал "ОШИБКА".
2. [0, 1, ..., 19, 20] → только сигнал "НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ".
3. [21, 22, ..., 34, 35] → нет сигналов.
4. [36, 37, ..., MAXINT – 1, MAXINT] → только сигнал "ИЗБЫТОЧНОЕ ДАВЛЕНИЕ".

(Здесь MAXINT и MININT являются максимальным и минимальным значениями 32-битного целого числа соответственно.)

Мы только что **разбили** наши эквивалентные классы. Разбиение является определением наших эквивалентных классов и гарантированием того, что они не перекрывают друг друга, но при этом покрывают все входные значения. Другими словами, они должны поддерживать **строгое разбиение**. Например, из-за плохих или неправильно понятых требований мы сгенерировали следующее разбиение эквивалентных классов:

1. [–2, –1, 0, 1, 2] → только сигнал "ОШИБКА".
2. [3, 4, ..., 21, 22] → только сигнал "НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ".

3. [20, 21, ..., 34, 35] → нет сигналов.

4. [36, 37, ..., 49, 50] → только сигнал "ИЗБЫТОЧНОЕ ДАВЛЕНИЕ".

Здесь есть две проблемы. Первая заключается в том, что все значения меньше –2 и больше 50 не привязаны к классам эквивалентности. Каким должно быть ожидаемое поведение, если датчик отправит значение 51? Это также рассматривается как ошибка? Или это "ИЗБЫТОЧНОЕ ДАВЛЕНИЕ"? В данном случае это **не определено**. Очень часто неопределенное поведение встречается при тестировании довольно сложных программных комплексов, но тестировщик ПО должен помогать в нахождении пробелов в покрытии и узнавать, что должно происходить (или, по крайней мере, происходит) в таких ситуациях.

Вторая и гораздо более неприятная проблема заключается в противоречивости принадлежности значений 20, 21 и 22. Они одновременно принадлежат к классам эквивалентности "НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ" и "нет сигналов". Какое ожидаемое поведение для входного значения 21? В зависимости от того, какой эквивалентный класс вы рассматриваете, это может быть отсутствие сигналов или сигнал "НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ". Это нарушение строгого разбиения, и вы легко можете увидеть, насколько проблематичным оно может быть.

Важно отметить, что эквивалентные классы не должны состоять из случаев, которые дают одинаковое выходное значение! Например, представим, что вы тестируете интернет-магазин. При стоимости заказа 100 долларов и менее предоставляется скидка 10%, а при заказе на 100,01 доллара и более — скидка 20%. И хотя здесь есть широкий диапазон выходных значений, поведение на выходе будет одинаковым для всех значений от 100 долларов и меньше и для всех значений от 100,01 доллара и больше. Они образуют два эквивалентных класса, и не будет отдельных классов для каждого индивидуального выходного значения (т. е. $10,00 → $9,00; $10,10 → $9,01 и т. д.).

Теперь, когда наши эквивалентные классы определены, можно написать тесты, которые покрывают всю функциональность дисплея. Мы можем отправить значение –2, чтобы протестировать класс эквивалентности "ОШИБКА", значение 10, чтобы протестировать класс эквивалентности "НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ", значение 30 для тестирования класса эквивалентности "НЕТ СИГНАЛОВ" и значение 45, чтобы протестировать класс эквивалентности "ИЗБЫТОЧНОЕ ДАВЛЕНИЕ". Конечно, эти значения были выбраны, скорее, произвольно. В следующей главе мы рассмотрим, как выбирать определенные значения, чтобы увеличить шансы нахождения дефектов.

4.2. Внутренние и граничные значения

В тестировании существует аксиома, что дефекты, скорее всего, могут быть найдены на границах двух классов эквивалентности. Эти значения — "последнее" одного класса эквивалентности и "первое" следующего класса эквивалентности — называются **граничными значениями** (boundary values). Значения, которые не являются граничными, называют **внутренними значениями** (interior values). Например,

рассмотрим очень простую математическую функцию, в которой вычисляется абсолютная величина целого значения. У нее есть два класса эквивалентности:

1. [MININT, MININT + 1, ..., –2, –1] → для входного значения *x* на выходе получим –(*x*).
2. [0, 1, ..., MAXINT – 1, MAXINT] → для входного значения *x* на выходе получим *x*.

Граничные значения здесь — это –1 и 0; они являются разделительной линией между двумя классами эквивалентности. Любое другое значение (например, 7, 62, –190) будет внутренним, окажется "в середине" класса эквивалентности.

Теперь, когда мы понимаем, что такое граничные и внутренние значения, можно задаться вопросом: почему случай с использованием граничных значений более вероятно окажется дефектным? Причина в том, что более вероятно, что в коде окажется ошибка на граничном значении, потому что классы эквивалентности оказываются близки. Давайте рассмотрим пример с функцией, вычисляющей абсолютную величину:

```
public static int absoluteValue (int x) {
    if (x > 1) {
        return x;
    }
    else {
        return -x;
    }
}
```

Вы видите ошибку в коде? Здесь простая ошибка на единицу в первой строке метода. Поскольку проверяется, что аргумент больше единицы, подача на вход 1 вернет –1. Так как граничные значения очень часто явно упоминаются в коде, это еще одна причина, по которой они могут вызвать ошибку или попасть не в "тот" класс эквивалентности. Давайте перепишем метод правильно, с учетом найденной ошибки:

```
public static int absoluteValue (int x) {
    if (x >= 1) {
        return x;
    } else {
        return -x;
    }
}
```

Гораздо лучше. Однако представьте, как трудно будет переписать этот метод, чтобы он выдавал ошибку, только когда вы передаете 57 — не 56 и 58, а именно 57. Такое возможно, конечно, но маловероятно. Так как очень редко имеется возможность исчерпывающе протестировать все входные значения для программы (или даже для одиночной функции), имеет смысл сфокусироваться на значениях, которые помогут вскрыть дефекты.

Вернемся к нашему дисплею, отображающему информацию о давлении, — наш тест-менеджер говорит, что у нас есть время протестировать больше значений. Мы

хотим удостовериться, что как минимум мы тестируем все граничные значения и достаточный набор внутренних значений. Сперва мы рассчитаем все граничные значения, а затем создадим тест-план, в котором будут все граничные значения и некоторые из внутренних.

Граничные значения:

1. –1, 0 (граница между "ОШИБКА" и "НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ").
2. 20, 21 (граница между "НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ" и "НОРМАЛЬНО").
3. 35, 36 (граница между "НОРМАЛЬНО" и "ИЗБЫТОЧНОЕ ДАВЛЕНИЕ").

Значения, которые необходимо протестировать:

1. Внутренние значения, "ОШИБКА": –3, –100.
2. Граничные значения, "ОШИБКА"/"НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ": –1, 0.
3. Внутренние значения, "НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ": 5, 11.
4. Граничные значения, "НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ"/"НОРМАЛЬНО": 20, 21.
5. Внутренние значения, "НОРМАЛЬНО": 25, 31.
6. Граничные значения, "НОРМАЛЬНО"/"ИЗБЫТОЧНОЕ ДАВЛЕНИЕ": 35, 36.
7. Внутренние значения, "ИЗБЫТОЧНОЕ ДАВЛЕНИЕ": 40, 95.

Можно также рассмотреть **неявные граничные значения**. В отличие от **явных граничных значений**, которые являются результатом требований (подобно тем, что рассчитаны выше), неявные значения определяются тестируемой системой или средой, в которой работает система. Например, MAXINT и MININT — неявные граничные значения; добавление единицы к MAXINT вызовет превышение максимально допустимого целого значения, а присвоение переменной значения MININT и последующее его уменьшение на единицу приведут к тому, что переменная примет значение MAXINT. В каждом из этих случаев класс эквивалентности изменится.

Неявные граничные условия могут зависеть от условий исполнения. Предположим, что у нас есть система с 2 Гбайт памяти, и мы запускаем в памяти обработку базы данных. Классы эквивалентности для тестирования функции, которая вычисляет количество добавленных строк, могут быть следующими:

1. Отрицательное число строк → ошибочное условие.
2. Ноль строк, или таблица не существует → возвращается NULL.
3. Одна строка или более → возвращается количество добавленных строк.

Существует неявная граница между тем количеством строк, которые можно добавить в память, и тем, которые нельзя. Тот, кто составлял требования, мог не подумать об этом, но вы, как тестировщик, должны держать в уме неявные граничные значения.

4.3. Базовые случаи, граничные случаи, угловые случаи

Продолжим наше изучение сенсорного датчика давления. Пройдя через различные тестовые ситуации (тест-кейсы), мы можем понять, что они различаются по своей распространенности. Бóльшую часть времени давление либо будет нормальным, либо немного меньше, либо немного больше. Каждый из таких случаев является **базовым** (base case) — система работает с ожидаемыми параметрами в обычном режиме.

Ситуация, когда входные значения лежат за границами нормальных рабочих параметров или приближаются к тем пределам, которые еще может обработать система, называется **граничным случаем** (edge case). Граничным случаем является дыра в шине и падение давления до нуля. Другим случаем станет то, что кто-то забыл про подключенный насос, и из-за этого давление подскочило до 200 PSI, что является максимально допустимым значением для шины.

Угловые случаи (corner case, иногда называемые **патологическими случаями**) относятся к ситуациям, когда сразу несколько составляющих работают неправильно в одно и то же время, или значение, грубо говоря, совершенно не попадает в диапазон ожидаемых значений. В качестве примера можно привести датчик значения шины, получающий значение 2 млрд PSI, что значительно выше давления внутри ядра Земли. В качестве другого примера можно привести шину, лопнувшую в тот момент, когда датчик ломается и пытается отправить сообщение об ошибке.

Хотя я использовал простую функцию с относительно хорошо заданными входными и выходными значениями, базовые случаи, граничные случаи и угловые случаи могут быть определены и изучены через другие виды операций. Рассмотрим интернет-магазин. Для тестирования корзины покупателя возможны следующие базовые случаи:

1. Добавить товар в пустую корзину.
2. Добавить товар в корзину, в которой уже присутствует товар.
3. Удалить товар из корзины, в которой уже присутствует товар.

Это всё **счастливые пути** — данные являются валидными, обычными, а проблемы отсутствуют. Нет ошибок и не возникают исключения, скорее всего, их создаст пользователь, система работает нормально и т. д. Теперь давайте рассмотрим граничные случаи:

1. Пользователь пытается добавить 1000 единиц товара в корзину одновременно.
2. Пользователь пытается нажать кнопку **Удалить** в корзине, которой нет товаров.
3. Пользователь открывает и закрывает корзину множество раз, ничего при этом больше не делая.
4. Пользователь пытается добавить в корзину товар, которого нет на складе.

Все эти случаи могут произойти, но они не "нормальные". Они могут потребовать специальных обработчиков ошибок (таких, как попытка удалить товары из пустой

корзины или добавить товар, которого нет в продаже), иметь дело с большими числами (например, добавление 1000 предметов) или нагружать систему странным способом (открывать и закрывать корзину снова и снова).

В итоге угловые случаи являются случаями, в которых возникают самые разрушительные проблемы или задействуются очевидно плохие данные. Несколько примеров:

1. Товар, который был в наличии на момент загрузки страницы, закончился на складе перед тем, как пользователь нажал кнопку **Добавить в корзину**.
2. Система получает запрос добавить 10^{80} товаров (что примерно соответствует количеству атомов во Вселенной) в корзину.
3. Память, в которой хранилось содержимое корзины, оказалась повреждена.

Угловые случаи часто включают в себя катастрофические сбои (потерю сетевого соединения, поломку ключевой подсистемы), генерацию полностью неправильных данных или множественные сбои, произошедшие одновременно.

4.4. Успешные и неуспешные случаи

При обсуждении тест-кейсов предполагается, что у теста может быть два варианта завершения. Во-первых, это может быть **успешный случай** (также называемый **позитивным тест-кейсом**), в котором результат соответствует ожидаемому при определенных данных. В общем, тесты, следующие по счастливому пути, когда пользователь делает то, что обычно должен делать, являются успешными случаями.

Неуспешными случаями (также называемые **негативными тест-кейсами**) являются случаи, когда мы ожидаем "падение" системы по каким-либо причинам, например при попытке осуществить запись на диск, доступный только для чтения, вычислить квадратный корень из отрицательного числа (в системах, которые не работают с комплексными числами) или при добавлении неправильного имени пользователя в систему. В неуспешных случаях вместо того, чтобы прийти к корректному результату, система выполнит... что-то другое. Это "что-то другое" зависит от теста и от того, какая функциональность тестируется. Это может быть ошибка кода, возврат значения по умолчанию, исключение, закрытие системы или просто вывод информации об ошибке в лог-файл или в консоль.

4.5. Тестирование черного, белого и серого ящиков

Существуют различные способы тестирования системы, каждый со своими достоинствами и недостатками. Сейчас мы познакомимся с тремя различными видами тестирования, а более подробно рассмотрим эти три парадигмы в следующих главах.

Возможно, самым легким видом тестирования для понимания является **тестирование черного ящика**. В этом случае у тестировщика нет знаний о том, как работает

система изнутри, и взаимодействовать с ней он может как обычный пользователь. Другими словами, тестировщик не знает, какая используется база данных, какие существуют классы или даже на каком языке написана программа. Вместо этого тестирование проводится так, как будто тестировщик является обычным пользователем программы.

Рассмотрим приложение для работы с электронной почтой. Взявший на тестирование это приложение тестировщик черного ящика будет проверять, может ли оно принимать и отправлять почту, проверять грамотность слов в письме, сохранять файлы и т. п. Этот тестировщик не станет проверять, что был вызван какой-то метод некоего класса, что объекты загружены в память или как осуществляются вызовы определенных функций. Если, например, тестировщик хочет убедиться, что письма могут быть правильно отсортированы по именам отправителей, в соответствующем тесте черного ящика нужно нажать кнопку или выбрать в меню команду **Сортировать по именам отправителей**. Тестировщик черного ящика может не знать, что программа написана на Java или Haskell, использовалась ли сортировка слиянием, быстрая сортировка или пузырьковая сортировка. Хотя тестировщика *будут* беспокоить результаты, полученные благодаря выбранным методам. Тестировщик черного ящика фокусируется на том, работает ли тестируемая система, как должна, с точки зрения пользователя, и нет ли в ней дефектов, с которыми может столкнуться пользователь.

Особенности системы, такие как вид примененного алгоритма или тип выбранной схемы использования памяти, могут предполагаться тестировщиком черного ящика, но он в первую очередь должен сфокусироваться на результатах работающей системы. Например, тестировщик черного ящика может обратить внимание, что система замедляется, когда начинает сортировать тысячи писем в почтовом ящике пользователя. Тестировщик черного ящика может предположить, что для сортировки использовался алгоритм типа $O(n^2)$, и отметить это как дефект. Тем не менее он не будет знать, какой алгоритм применялся или какие особенности программного кода вызвали замедление.

По знанию работы тестируемой системы **тестирование белого ящика** является противоположностью тестированию черного ящика. В тестировании белого ящика у тестировщика есть глубокое представление об исходном коде, и он напрямую тестирует этот код. Тестировщик белого ящика может протестировать отдельные функции кода, очень часто изучая особенности работы системы гораздо более детально, чем в тестах черного ящика.

Продолжая рассматривать пример с почтовой программой, обратим внимание, что тесты белого ящика могут проверять функцию сортировки (`EmailEntry[] emails`), подавая на вход различные значения и изучая то, что эта функция возвращает или делает. Тестировщиков белого ящика будет беспокоить, что именно произойдет, если были переданы массив нулевой длины или ссылка на нулевой элемент, в то время как тестировщики черного ящика будут озабочены только тем, что случится, если они попытаются отсортировать пустой список сообщений в самом приложении. Тестировщики белого ящика обращаются с кодом как с кодом — проверяют,

что возвращаемое значение функции корректное, объекты инициализированы правильно и т. д. — вместо того чтобы смотреть на систему с точки зрения пользователя.

Тестирование серого ящика, как подразумевает его название, является гибридом тестирования белого ящика и тестирования черного ящика. Тестирование серого ящика включает обращение тестировщика к системе в роли обычного пользователя (как в тестировании черного ящика), но со знаниями программного кода и устройства системы (как в тестировании белого ящика). Обладая этими знаниями, тестировщик серого ящика может создавать более глубокие тесты черного ящика.

Давайте предположим, что наш тестировщик серого ящика собирается протестировать функцию сортировки в почтовом приложении. Изучая код, тестировщик понимает, что система использует метод сортировки вставками. Известно, что у сортировки вставками наихудшая производительность, когда необходимо отсортировать данные, уже отсортированные в обратном порядке. Таким образом, тестировщик серого ящика может добавить тест, который проверяет, что система способна отсортировать список электронных писем, которые расположены в обратном порядке. Другой пример — тестировщик может обратить внимание, что в функции поиска нет обработки пустых данных, и проверить, приведет ли простое нажатие клавиши <Enter> без ввода данных к разыменованию нулевого указателя в программном коде, к поиску `""` и отображению всех писем в результатах поиска, или к другому непредсказуемому поведению.

4.6. Статическое и динамическое тестирование

Другим способом систематизации тестов является их группировка на **статические** и **динамические тесты**. В динамических тестах тестируемая система работает и код исполняется. Фактически все тесты, которые мы рассматривали ранее, были динамическими. Даже если мы не видим сам код, компьютер работает, принимая какие-то данные на входе, обрабатывает их и выдает выходные данные.

Статический тест, наоборот, не исполняет код. Скорее, он пытается протестировать особенности системы без запуска самой системы. Примерами статического тестирования могут быть запуск **линтера** (который помечает "дурно пахнущий код" — скажем, в котором к переменной обращаются еще до присваивания ей какого-нибудь значения) или когда кто-то проверяет код вручную без его запуска.

На первый взгляд, преимущества статического тестирования могут быть неочевидны. В конце концов, какую дополнительную выгоду можно получить от тестирования программы без ее запуска? Вы как будто осознанно отстраняетесь от прямого воздействия и наблюдаете всё со стороны. Однако, поскольку статический анализ напрямую изучает код вместо результатов исполнения этого кода, он может помочь найти проблемные участки самого кода.

В качестве примера давайте рассмотрим два метода, которые принимают на входе строковую переменную `toChirp` и добавляют в конце ее строку "CHIRP!". Например, передача значения `foo` вернет `fooCHIRP`! в обоих методах:

```
public String chirpify(String toChirp) {
    return toChirp + "CHIRP!";
}

public String chirpify(String toChirp) {
    char[] blub = toChirp.toCharArray();
    char[] blub2 = char[blub.length + 6];
    blub2[blub.length + 0] = (char) 0x43;
    blub2[blub.length + 1] = (char) 0110;
    blub2[blub.length + 2] = (char) 73;
    blub2[blub.length + 3] = (char) (01231);
    blub2[blub.length + 4] = (char) (40*2);
    blub2[blub.length + 5] = "!";
    String boxer99 = String.copyValueOf(data2);
    return boxer99;
}
```

Оба этих метода будут возвращать один результат для определенного входного значения. Их выходные данные могут быть изучены динамическими тестами и сверены с ожидаемыми значениями. Однако можно ли сказать, что второй метод лучше? Он довольно запутанный; сложно понять, что он делает; имена переменных бессмысленные. Поиск всего того, что неправильно в коде, оставим в качестве упражнения читателю. В случае с динамическими тестами может оказаться сложно или невозможно определить разницу между этими двумя методами. Но, используя такой метод статического тестирования, как обзор кода (code review), довольно просто найти проблемы во втором методе.

Глава 5

Требования

Запомните, что валидация ПО заключается в гарантировании, что мы создаем правильное программное обеспечение; другими словами, убеждаемся, что создаем то, что хотят пользователи и/или потребители. Для того чтобы провалидировать программу, мы должны знать, что она должна делать, по мнению пользователей, и эта задача может оказаться гораздо более сложной, чем вы думаете. Пользователи часто не способны четко сформулировать, что именно они хотят. Они могут думать, что знают точно свои желания, но когда они видят реализацию, то сразу же понимают, что это не то. Или у них может вообще не быть никакой идеи — они просто хотят от вашей команды, чтобы вы создали "что-то вроде социальной сети, но для покупок и продаж. Что именно? Что-то типа хот-догов. А для кошек можно?".

Одним из способов определить, какое ПО создавать, является определение **требований** к программному обеспечению. Требования являются формулировками, определяющими, что именно эта часть программы должна делать в определенных условиях. Так более или менее принято в зависимости от области работы и вида создаваемого программного обеспечения. Например, при создании программы для мониторинга ядерного реактора должны быть очень точные и хорошо проработанные требования. Если вы работаете для социально-медийного стартапа (надеюсь, не того, который "продажа и покупка, типа того, например, хот-догов... для кошек", потому что среди них конкуренция), тогда ваши "требования" могут быть нацарапаны вашим CEO на салфетке после нескольких бутылок вина и после нескольких встреч с реальными потребителями.

Требования гарантируют, что разработчики знают, что создавать, а тестировщики знают, что тестировать. Хотя требования очень важны, они не являются священными! Здравый смысл должен преобладать при изучении требований, и требования могут быть изменены. Обратите внимание, что помимо требований существуют другие способы определения, какое ПО создавать, например пользовательские истории (user stories).

В нашем примере с датчиком давления колес из главы по основам тестирования у нас были сравнительно простые требования, хотя мы не оговаривали, чем именно они являются:

1. Сигнал "ОШИБКА" включается для PSI, равного –1 или меньше.
2. Сигнал "НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ" включается для PSI от 0 до 20.

3. Сигналы не включаются для PSI от 21 до 35 (нормальные рабочие условия).
4. Сигнал "ИЗБЫТОЧНОЕ ДАВЛЕНИЕ" включается для PSI от 36 и выше.

Эти "неформальные требования" показывают, что должно происходить с системой при определенных входных значениях. Классический способ написания требований заключается в том, чтобы сказать, что система "должна" сделать. При тестировании вы можете мысленно перевести "должна" как "обязана". То есть если требование говорит, что система должна сделать что-то, значит, система обязана сделать что-то для того, чтобы вы как тестировщик могли сказать, что система соответствует требованию. Требования должны быть написаны точно, и следует избегать неоднозначности любой ценой. Давайте переведем эти неформальные требования в нечто, больше соответствующее требованиям, которые пишутся в реальном мире.

- *REQ-1.* Если дисплейным датчиком получено значение давления, равное –1 или меньше, тогда сигнал "ОШИБКА" *должен* быть включен, а все остальные сигналы *должны* быть отключены.
- *REQ-2.* Если дисплейным датчиком получено значение давления, лежащее в пределах от 0 до 20 (включительно), тогда сигнал "НЕДОСТАТОЧНОЕ ДАВЛЕНИЕ" *должен* быть включен, а все остальные сигналы *должны* быть отключены.
- *REQ-3.* Если дисплейным датчиком получено значение давления, лежащее в пределах от 21 до 35 (включительно), тогда все сигналы *должны* быть отключены.
- *REQ-4.* Если дисплейным датчиком получено значение давления, равное 36 или больше, тогда сигнал "ИЗБЫТОЧНОЕ ДАВЛЕНИЕ" *должен* быть включен, а все остальные сигналы *должны* быть отключены.

Обратите внимание, насколько подробными эти требования стали по сравнению с приведенными выше неформальными. Инжиниринг требований является настоящей инженерной дисциплиной, и может оказаться очень сложно описать большую и/или сложную систему. Также обратите внимание, насколько более плотным и бóльшим стал текст в попытке избежать неоднозначности. Определенно, на написание формальных требований уйдет гораздо больше времени, чем накалякать общий набросок программы на салфетке. Теперь и изменения вносить станет сложнее. Компромисс может быть найден, а может и нет, в зависимости от области работы и тестируемой системы. Гарантирование того, что авиационное программное обеспечение при всех условиях правильно контролирует полет самолета, вероятно, потребует тщательно проработанной **спецификации требований** (списка всех требований системы), в то время как для упомянутого выше медийно-социального сайта подобное может не понадобиться. Жесткость может обеспечить очень хорошее определение системы, но ценой гибкости.

Кстати, если вы задумывались, почему адвокатам платят так много, то можно сказать, что описанное выше применимо к тому, чем они занимаются каждый день. Можно представить, что законы — это набор требований, которые человек должен соблюдать, чтобы быть законопослушным; представить, какие наказания применяются в случае, если человек нарушает закон, как создаются законы и т. д.:

1. Если нарушитель переходит оживленную дорогу не по пешеходному переходу, и имеются приближающиеся машины, и по крайней мере одной машине необходимо затормозить, то нарушитель *должен* быть обвинен в нарушении "Пешеход не уступил дорогу".
2. Если человек переходит оживленную дорогу, когда на светофоре пешеходного перехода не горит зеленый свет и нет приближающихся машин, нарушитель *должен* быть обвинен в нарушении "Пешеход не соблюдает сигналы светофора".

И в законе, и в тестировании возможно "спуститься в кроличью нору", стараясь определить, что именно означает текст. Английский язык полон неоднозначностей. Например, возьмем совершенно разумное требование, которое может быть прочитано различными способами, один из которых:

- *UNCLEARREQ-1.* Основная сигнализационная система должна звучать, если обнаружен нарушитель с помощью визуального дисплейного модуля.

Должно ли это означать, что сигнализация должна зазвучать, если нарушитель пользовался помощью визуального дисплейного модуля? Или сигнализация должна звучать, если нарушитель обнаружен благодаря визуальному дисплейному модулю системы? Старайтесь избегать такой неоднозначности при написании требований. Хотя тестировщики редко пишут требования самостоятельно, их частенько просят просмотреть их. Гарантирование того, что вы понимаете требование и это требование может быть протестировано, сохранит время и позволит позднее избежать головной боли в процессе разработки.

При написании требований важно держать в голове, что требования обозначают, что системе следует делать, но не как ей это следует делать. Другими словами, не надо определять детали реализации, а лишь как система или подсистема взаимодействует с окружающим миром и воздействует на него. Это легче понять на примерах из новой межзвездной космической игры.

Хорошие требования:

- *GOOD-REQ-1* — когда пользователь нажимает кнопку **Скорость**, текущая скорость космического корабля должна быть отображена на главном экране;
- *GOOD-REQ-2* — система должна быть способна поддерживать скорость 0,8*c* (80% скорости света) по крайней мере в течение трех дней (7200 часов) без потребности в техобслуживании;
- *GOOD-REQ-3* — система должна сохранять последние 100 координат мест, в которых брались образцы.

Плохие требования:

- *BAD-REQ-1* — когда пользователь нажимает кнопку **Скорость**, система должна обратиться к памяти по адресу 0x0894BC50 и отобразить ее значение на экране;
- *BAD-REQ-2* — система должна моделировать реакцию "антивещество — вещество" для того, чтобы достигать скорости 0,8*c*;
- *BAD-REQ-3* — система должна использовать реляционную базу данных, чтобы сохранять последние 100 координат мест, в которых брались образцы.

Обратите внимание, что все плохие требования говорят о том, *как* что-то должно быть выполнено, а не *что* система должна делать. Что произойдет, если изменится расположение памяти? BAD-REQ-1 должен измениться, как и другие требования, в которых данные зависят от расположения в определенной области памяти. Почему важно использовать именно реактор антиматерии-материи? В конце концов, ключевым моментом является то, что космический корабль может двигаться с определенной скоростью. И в итоге, важно ли то, что нужно использовать реляционную базу данных для хранения координат? Если посмотреть с точки зрения пользователя, то беспокоить его должно лишь то, что координаты должны быть сохранены.

Для сложных или критически важных с точки зрения безопасности систем (таких как реальный космический звездолет) требования могут определять реализацию. В этих случаях не только важно, что система делает нечто, но и то, что она делает это проверенным и определенным способом. Для большинства систем, однако, такие требования являются излишними и значительно ограничат гибкость в процессе разработки программного обеспечения. Также станет более сложным тестировать эти требования, поскольку тестировщику надо определять не только соответствие наблюдаемого поведения ожидаемому, но и как наблюдаемое поведение происходило.

Создавая требования по особенностям реализации, вы исключаете возможность тестирования черного ящика. Не зная программного кода, как вы можете быть уверены, что система отображает содержимое памяти по адресу 0x0895BC40? Это невозможно (по крайней мере, если у вас нет невообразимой суперсилы, позволяющей заглянуть внутрь чипа памяти и понять, что там хранится и где). Все тесты окажутся тестами белого ящика.

5.1. Тестируемость

С точки зрения тестировщика, одним из самых важных аспектов требований является то, могут ли они быть протестированы или нет. С точки зрения цикла разработки программного обеспечения формулировка "тестируемые требования" является синонимом "хороших требований". Невозможно доказать, что требование, которое нельзя протестировать, выполнено. Давайте рассмотрим пример с двумя требованиями и постараемся определить, какое из них лучше. Обратите внимание, что оба они семантически и синтаксически верные и содержат это крайне важное слово "должен":

1. Система должна увеличивать счетчик ПОСЕТИТЕЛЕЙ на единицу каждый раз, когда сенсор ТУРНИКЕТА активируется без ошибок.
2. Система должна работать со счетчиком каждый раз, когда кто-то проходит.

Заметим, что первое требование очень определенное; оно обозначает, что должно быть сделано, какие входные данные отслеживать и какого поведения ожидать. А именно: каждый раз, когда датчик ТУРНИКЕТА активируется, мы ожидаем, что счетчик ПОСЕТИТЕЛЕЙ (который может быть переменной, дисплеем или чем-то

еще — это следует определить в описании полных требований) увеличивается на единицу. Второе требование очень неоднозначное сразу по нескольким причинам. Что делать со счетчиком? Как он узнает, что кто-то прошел? Что означает, что кто-то проходит? Невозможно создать тест для такого требования.

Теперь, когда мы увидели примеры тестируемых и нетестируемых требований, можем ли мы подробно описать, что значит для требования быть тестируемым?

Для того чтобы требования были тестируемыми, они должны соответствовать пяти критериям, каждое из которых мы рассмотрим отдельно. Им следует быть:

1. полными;
2. последовательными;
3. недвусмысленными;
4. количественными;
5. выполнимыми для тестирования.

Требования должны покрывать всю работу системы. Это то, что мы подразумеваем, когда говорим, что спецификация требований должна быть **полной**. Всё, что не покрыто требованиями, будет интерпретировано по-разному разработчиками, дизайнерами, тестировщиками и пользователями. Если что-то важно, оно должно быть определено точно.

Требования должны быть **последовательными**, т. е. они не должны противоречить друг другу или законам Вселенной (или той области, в которой вы работаете). Требования, которые не противоречат друг другу, являются **внутренне последовательными**; требования, которые не противоречат миру вне системы, называются **внешне последовательными**.

Вот пример группы требований, которые не являются внутренне последовательными:

- *INCONSISTENT-REQ-1* — система должна отображать сообщение: "ВНИМАНИЕ: ИЗБЫТОЧНОЕ ДАВЛЕНИЕ" на консоли, когда давление составляет 100 PSI или более;
- *INCONSISTENT-REQ-2* — система должна отключить консоль и не отображать информацию до тех пор, пока давление меньше 200 PSI.

Что должна делать система, если давление лежит в границах от 100 до 200 PSI? В данном случае вы можете использовать разбиение на классы эквивалентности, чтобы определить, что требования не являются внутренне последовательными.

Требованиям необходимо быть **недвусмысленными**, т. е. они должны определять всё настолько точно, насколько возможно для той области программного обеспечения, с которой вы работаете. Допустимый уровень однозначности будет значительно отличаться в зависимости от того, какой тип программного обеспечения вы разрабатываете. Например, если вы создаете игру для детей, возможно, достаточно оговорить, что некая площадь должна быть "красной". Если же вы описываете требования к интерфейсу ядерного реактора, для предупредительного сигнала должен быть указан точный оттенок красного цвета в цветовой модели Pantone.

С учетом вышесказанного не загоните себя в угол слишком строгими требованиями. Например, если ваши требования говорят, что какая-то страница должна быть определенного размера в пикселах, у вас могут возникнуть трудности с конвертацией ее для мобильных устройств. Тем не менее требования не должны принимать следующую форму:

- *AMBIGUOUS-REQ-1* — система должна выключать всё, когда нажата кнопка **Выключение**.

Что значит "выключать всё"? Разговаривая с друзьями или коллегами, мы можем использовать такие неопределенные термины, потому что человеческий мозг удивительно хорош в распознании двусмысленностей. Однако двусмысленные требования могут привести к тому, что разработчики или другие заинтересованные лица могут интерпретировать их по-разному. Классический пример — провал запущенной NASA миссии Mars Climate Orbiter, когда часть разработчиков использовала имперскую систему мер, а другая — метрическую. Обе группы думали, что правильный путь получения результатов очевиден, но они использовали различные реализации.

Насколько это возможно, требования должны быть **количественными** (как противоположность качественным), т. е. если нужно использовать в требованиях численные значения, то используйте их. Следует избегать любых субъективных терминов вроде "быстрый", "легко реагирующий", "практичный" или "офигенный". Если вам необходимо указать, что система должна сделать, — указывайте. Например, следующее требование качественное, но не количественное:

- *QUALITATIVE-REQ-1* — система должна возвращать результат предельно быстро.

Что мы хотим этим сказать? Тестировать это требование невозможно без определения, что мы понимает под "предельно быстро" и какой результат должен быть получен быстро.

В итоге должен присутствовать некий здравый смысл при написании требований. Можно написать требование, которое теоретически можно протестировать, но в реальной жизни по разным причинам этого сделать нельзя. Такое требование **невыполнимо** для тестирования. Скажем, у нас есть следующие требования для тестирования нашего датчика давления:

- *INFEASIBLE-REQ-1* — система должна быть способна выдерживать давление до $9{,}5 \cdot 10^{11}$ фунтов на квадратный дюйм;
- *INFEASIBLE-REQ-2* — при нормальных рабочих условиях (определенных где-то в другом разделе) система должна оставаться работоспособной в течение 200 лет непрерывного использования.

Оба эти требования, конечно, можно протестировать. В первом случае просто поместите систему в эпицентр относительно мощного термоядерного взрыва и определите, ухудшилось ли качество системы после детонации. Второе требование еще проще: просто нужно ездить с датчиком давления воздуха в колесах 200 лет. Так как продолжительность жизни человека заметно меньше, вам, вероятно, понадобятся несколько поколений тестировщиков. Передавать ли должность "тестировщик датчика давления" по наследству или использовать подход "мастер — ученик", зависит только от вас.

Когда дело касается физических явлений, становится очевидным, что глупо тестировать такие требования. (Если они не кажутся глупыми вам, сколько тестировщиков вы знаете, у которых в распоряжении имеется ядерное оружие? Вероятно, меньше пяти.) Однако зачастую более сложно определить, что требование является невыполнимым, когда вы имеете дело с программным обеспечением. Функционируя в реальном физическом мире, человеческий мозг очень часто сталкивается с трудностью определения, насколько выполнимым является требование к программе (существующей в виртуальном мире), которое нужно протестировать.

5.2. Функциональное против нефункционального

Функциональные требования определяют, что система должна *делать*; нефункциональные требования определяют, какой система должна *быть*.

Требования, которые мы обсуждали ранее, являлись функциональными, т. е. они говорили, что система должна выполнять конкретное действие при определенных условиях. Например:

1. Система должна показывать сообщение "Пользователь не найден", если пользователь попытается авторизоваться и введенное имя пользователя не существует в системе.
2. После извлечения записи из базы данных, если какое-то из полей неправильное, система должна возвращать строку "Неверное значение" для этого поля.
3. После запуска система должна отобразить сообщение "ДОБРО ПОЖАЛОВАТЬ В СИСТЕМУ" в пользовательской консоли.

Функциональные требования являются (относительно) простыми для тестирования; они говорят, что определенное поведение должно происходить при заданных условиях. Очевидно, будут какие-то сложности и вариации при тестировании некоторых требований, но общая идея очевидна. Например, для второго требования тесты могут проверять каждое поле базы данных и различные виды неверных значений. Это может быть довольно запутанный процесс, но существует план, позволяющий разработать тесты, которые напрямую вырастают из требований.

Нефункциональные требования описывают общие характеристики системы, в отличие от определенных действий, которые выполняются при определенных условиях. Нефункциональные требования часто называют **атрибутами качества**, потому что они описывают качество системы в противоположность тому, что она должна делать конкретно. Некоторые примеры нефункциональных требований:

1. Система должна использоваться опытным пользователем компьютера после не более 3-часового обучения.
2. Система должна быть способной работать с одной сотней пользователей одновременно.
3. Система должна быть надежной с временем непредвиденного простоя меньше 1 часа в месяц.

Нефункциональные требования зачастую более сложны для тестирования, чем функциональные, потому что ожидаемое поведение оказывается гораздо более неопределенным. Именно поэтому особенно важно ясно определить само требование.

Одним из основных путей четко определить нефункциональные требования является использование в них количественных значений.

5.3. Замечание о наименовании требований

В былые времена, когда примитивные инженеры долбили бинарные деревья каменными топорами, существовал только один путь написания требований. Это был путь, по которому требования писали их праотцы, а до этого писали праотцы праотцов, и так до самых истоков цивилизации. Этот священный метод наименования требований был таков.

ТРЕБОВАНИЯ:

1. Система должна делать X.
2. Система должна делать Y.
3. Система должна делать Z, если происходит событие A...

Это было довольно просто для маленьких проектов. Но по мере того, как проект становился всё больше и больше, в этой схеме начинали возникать проблемы. Например, что произойдет, если требование станет нерелевантным? Тогда возникнут "пропавшие" требования; в списке требований могут быть разделы 1, 2, 5, 7, 12 и т. д. Наоборот, что если нужно добавить требования к программе? Если нумерация списка требований возрастает линейно, эти новые требования должны быть размещены в конце или втиснуты между существующими (требование 1.5).

Другой проблемой была необходимость запоминать требования, основываясь исключительно на номерах. Для обычного человека не очень сложно держать в голове список из нескольких требований, но он не сможет это делать, когда требований к программному обеспечению окажется сотни, если не тысячи.

Было разработано несколько способов для разрешения этой проблемы. Один из них заключался в группировке всех требований по различным секциям (например, "DATABASE-1", "DATABASE-2") и продолжении в них традиционной нумерации. По крайней мере, в такой схеме новое требование для базы данных DATABASE не понадобится размещать в конце общего списка требований, зато можно будет поместить его вместе с аналогичными требованиями. Также из названия можно догадаться, о чем говорит это требование.

Другой способ — использовать соответствующие аббревиатуры в названиях требований. Префиксы и суффиксы вроде "FUN-" для функциональных и "NF-" для нефункциональных являются довольно обычными.

При этом следует отметить, что еще более важно использовать одинаковый подход к наименованиям требований внутри команды!

Глава 6

Тест-планы

Теперь, когда вы узнали про требования, базис теории и терминологию тестирования программного обеспечения, пришло время заняться **тест-планами**. В тест-планах содержится полный план тестирования тестируемой системы. Затем они выполняются, т. е. человек или автоматизированный инструмент тестирования проходит все шаги тест-плана, и его результаты сравниваются с тем, что получено на самом деле. Иными словами, мы будем проверять соответствие ожидаемого поведения системы (то, что мы написали в нашем тест-плане) тому поведению, которое мы наблюдаем (то, что на самом деле выполняет система).

6.1. Базовая схема тест-плана

Тест-план по своей сути является просто набором **тест-кейсов**. Тест-кейсы — это отдельные тесты, которые составляют тест-план. Давайте предположим, что вы создаете тест-план для тестирования нового приложения, которое говорит вам, является ли ваш кофе слишком горячим, чтобы его пить. Маркетологи нашего гипотетического приложения любят холодный кофе и решили не создавать никакой функционал, сообщающий о том, что температура кофе слишком низкая. Имеются два требования:

- *FUN-COFFEE-TOO-HOT* — если измеренная температура кофе составляет 80 °C или выше, на экране приложения должно отображаться сообщение "СЛИШКОМ ГОРЯЧО";
- *FUN-COFFEE-JUST-RIGHT* — если измеренная температура кофе составляет меньше 80 °C, на экране приложения должно отображаться сообщение "САМОЕ ТО".

Как мы может разработать тест-план для нашего приложения по измерению температуры кофе? Есть одно входное значение — измеренная температура кофе — и два возможных выходных, одно из набора ["СЛИШКОМ ГОРЯЧО", "САМОЕ ТО"]. Мы проигнорируем, что большинство людей посчитают, что кофе температуры 7 °C далеко не "САМОЕ ТО".

Единственное входное значение и одно из двух возможных выходных значений являются простым случаем разбиения класса эквивалентности, поэтому давайте разобьем эти классы эквивалентности:

- *JUST-RIGHT* — [–INF, –INF + 1, ..., 79, 80] → "САМОЕ ТО";
- *TOO-HOT* — [81, 82, ..., INF – 1, INF] → "СЛИШКОМ ГОРЯЧО".

Нашими граничными значениями являются 80 и 81, т. к. они отмечают разделение между двумя классами эквивалентности. Давайте также использовать два внутренних значения: 57 °C для класса "САМОЕ ТО" и 93 °C для класса "СЛИШКОМ ГОРЯЧО". Для этого конкретного примера тест-плана мы проигнорируем неявные граничные значения бесконечности и отрицательной бесконечности (или, в представлении системы, MAXINT и MININT).

Используя эти значения и общее представление, что мы хотим протестировать, мы можем начать создавать тест-кейсы. Хотя в разных инструментах и компаниях применяют различные шаблоны для ввода тест-кейсов, существует относительный стандарт, который может быть использован или модифицирован для большинства проектов:

1. **Идентификатор**, такой как "16", "DB-7" или "DATABASE-DROP-TEST", который однозначно идентифицирует тест-кейс.
2. **Тест-кейс** — описание тест-кейса и определение, что он тестирует.
3. **Предусловия** — любые предусловия для состояния системы или мира перед началом теста.
4. **Входные значения** — любые значения, напрямую подаваемые на вход в тесте.
5. **Шаги исполнения** — фактические шаги теста, которые должны быть выполнены тестировщиком.
6. **Выходные значения** — любые значения, полученные на выходе теста.
7. **Постусловия** — любые постусловия состояния системы или мира, которые должны быть истинны после выполнения теста.

Не беспокойтесь, если у вас возникли какие-либо вопросы по поводу этих определений. В следующих главах мы рассмотрим их более глубоко и приведем примеры.

6.1.1. Идентификатор

Как и у требований, у тест-кейсов тоже есть идентификаторы. Они позволяют коротко и быстро сослаться на тест-кейс. Во многих случаях это просто числа, но можно также использовать более сложные системы наименований, подобные той, что была описана в разделе об именах для требований.

Тест-планы обычно не такие большие, как требования для программы; когда они становятся достаточно объемными, отдельные тест-планы группируют в бо́льшие тестовые наборы (тест-сьюты, test suites). Зачастую идентификаторы являются обычными числами. Если вы используете автоматизированное ПО для работы с тестами, то обычно оно выполняет нумерацию за вас.

6.1.2. Тест-кейс (или краткое изложение)

В этом разделе приводится краткое изложение того, что и как должно тестироваться в тест-кейсе. Таким образом, некто, изучающий тест-план, сможет с ходу сказать, для чего нужен этот тест-кейс и почему он включен. Обычно это можно понять после внимательного изучения предусловий, входных значений и шагов исполнения, но для человека проще всего лишь прочитать, что должен делать этот тест.

Примеры:

1. Убедиться, что товары со скидкой могут быть добавлены в корзину и при этом их цена автоматически уменьшится.
2. Убедиться, что передача нечисловых значений приведет к тому, что функция вычисления квадратного корня вернет исключение `InvalidNumber`.
3. Убедиться, что когда система обнаружит достижение внутренней температуры 66 °C, на дисплее отобразится сообщение об ошибке и произойдет выключение в течение 5 секунд.
4. Убедиться, что если операционная система переключается между временны́ми зонами во время путешествия, то для отображения результата вычислений используется начальная временная зона.

6.1.3. Предусловия

Для теста зачастую необходимо, чтобы для его запуска система находилась в определенном состоянии. Хотя можно теоретически рассмотреть приведение системы в данное состояние как часть исполнения теста (*см. разд. 6.1.5*), чаще гораздо более разумно сделать, чтобы определенные **предусловия** были реализованы перед запуском теста. Это необходимо для многих тестов, которые проверяют не математические чистые функции.

Примеры предусловий:

1. Система запущена.
2. В базе данных содержится запись о пользователе Joe с паролем EXAMPLE.
3. Для флага SORT_ASCEND установлено значение true.
4. В корзине покупателя уже находятся три товара.

Давайте рассмотрим последний пример и поймем, почему лучше иметь выполненное предусловие, чем включать его в шаги теста. Мы хотим протестировать, что добавление товара в корзину, в которой уже находятся три товара, приведет к отображению информации о четырех товарах в корзине. Читая описание этого теста, вы можете понять, что добавление первых трех товаров не упоминается. С точки зрения этого теста шаги по добавлению этих товаров не относятся к делу; всё, что важно, — это то, что в начале теста три товара находятся в корзине.

С прагматической точки зрения способность устанавливать предусловия добавляет гибкость и краткость тестам. Предположим, что вместо размещения предусловия о наличии в корзине трех товаров, мы включим в тест следующие шаги:

1. Найти товар "1XB".
2. Выбрать товар "1XB".
3. Нажать кнопку **Добавить в корзину** три раза.

Это может сработать, когда вы запускаете тест в первый раз. Но этот тест оказывается очень **хрупким** — существует множество способов разрушить его, если система вдруг меняется. Что, если товар "1XB" больше не существует? Что, если у функции поиска имеется дефект и предметы, название которых начинается с "1", не могут быть найдены? Что, если кнопка **Добавить в корзину** поменяла название?

Также существует недостаток с точки зрения краткости. Мы только что добавили три шага в тест, где всего одно предусловие. Краткость, помимо того, что она сестра таланта, очень полезна в гарантировании того, что фокусирование будет идти на важных частях теста. Шаблонный тест — враг внимания и фокусирования.

Разделительная линия между предусловиями и шагами выполнения иногда может оказаться скорее искусством, чем наукой. Вообще, чем более критична с точки зрения безопасности наша рабочая область, тем более точными будут предусловия. Например, допустим, вы тестируете хостинг изображений, где все изображения являются общедоступными и видимыми для всех пользователей. Тест-кейс включает в себя проверку, что определенное изображение показывается на экране, когда пользователь открывает соответствующий URL. Для этого теста могут быть достаточны следующие предусловия:

1. Пользователь залогинился.
2. Изображение было загружено по URL **/pictures/foo**.

Однако если мы тестируем банковское ПО и используем изображение для предупреждения о неправильной транзакции, то здесь, вероятно, будет больше предусловий, а те, что уже имеются, будут более точными:

1. Пользователь X залогинился с паролем Y.
2. У пользователя X в аккаунте нет предупреждений или сообщений о необходимости проверки.
3. Сберегательный счет пользователя X содержит $0,00.
4. Расчетный счет пользователя X содержит $50,00.
5. У пользователя X нет других счетов в банке, кроме сберегательного и расчетного.
6. Пользователь X попытался снять $50,01 с расчетного счета.

В обоих случаях шаги исполнения будут одинаковыми или, по крайней мере, очень похожими — перейти по URL-адресу и проверить, что отображается определенное изображение. Однако состояние системы может быть значительно более детализированно в случае использования банковского программного обеспечения. И не только из-за того, что такая система гораздо более сложная, но и из-за того, что последствия ошибки для банка окажутся более значительными, чем для хостинга изображений. В этом случае имеет смысл точно определить, что должно происходить и что должно быть создано перед шагами выполнения. Чем более точно вы

опишете предусловия, тем легче будет воспроизвести определенную ситуацию, и, как мы говорили, воспроизводимость является ключом к разрешению проблемы.

6.1.4. Входные значения

В то время как предусловия являются теми аспектами системы, которые должны быть установлены перед запуском теста, входные значения являются теми значениями, которые передаются напрямую в тестируемую функциональность. Это различие может быть трудноуловимым, поэтому давайте рассмотрим несколько примеров.

Представим, что у нас есть алгоритм сортировки billSort, который предположительно в 20 раз быстрее, чем любой другой алгоритм. Не принимая на веру то, что billSort всегда дает правильный результат, мы разрабатываем тесты для него. В одном из них используется глобальная переменная `SORT_ASCENDING`. В зависимости от того, какое значение принимает эта булева переменная — `true` или `false`, сортировка будет идти либо по возрастанию (от меньшего значения к большему, т. е. `"a"`, `"b"`, `"c"`), либо по убыванию (от большего значения к меньшему, т. е. `"c"`, `"b"`, `"a"`). Если мы собираемся протестировать этот алгоритм сортировки, установка флага будет считаться предусловием, т. к. это то, что должно быть установлено перед тестом. Массив `["a", "b", "c"]` будет являться входными значениями; эти значения подаются напрямую для тестирования.

Другой способ понять разницу между входными значениями и предусловиями — представить себе тесты как методы. В любом случае это хорошее упражнение для вас — ведь мы займемся этим, когда дойдем до главы о юнит-тестах!

```
public boolean testArraySort() {
    // ПРЕДУСЛОВИЯ
    SORT_ASCENDING = true;
    // ВХОДНЫЕ ЗНАЧЕНИЯ
    int[] vals = [1, 2, 3];
    // Новый улучшенный метод billSort! :)
    billSorted = billSort(vals);
    // Старый унылый метод сортировки из Java :(
    NormalSorted = Arrays.sort(vals);
    if (Arrays.equals(billSorted, normalSorted) {
        // Наши массивы равны, тест пройден
        return true;
    } else {
        // Наши массивы не равны, тест провален
        return false;
    }
}
```

Обратите внимание, что вы не используете флаг `SORT_ASCENDING` в тестируемой функциональности (в частности, в методе `billSort()`), и, значит, он не является входным значением. Однако массив `vals` передается в метод `billSort`, и поэтому этот массив рассматривается как входное значение.

Но разве нельзя перестроить систему так, чтобы можно было отправлять флаг в качестве входного значения в метод `billSort()`?

```
// Аргументы = массив vals, флаг SORT_ASCENDING
billSorted = billSort(vals, true)
```

Это, конечно, возможно, и тогда можно рассматривать `SORT_ASCENDING` как входное значение, а не предусловие. Является ли что-то предусловием или входным значением — очень часто зависит от реализации программы. Если бы мы программировали на таком языке, например, как Haskell, где побочные эффекты для большинства функций крайне ограничены, то функции, подобные нашей, никогда бы не имели никаких предусловий кроме "программа запущена".

Сортировка является чисто математической идеей, но многое из того, что тестирует тестировщик, не настолько однозначно. В таких случаях зачастую сложно определить, что считается входными значениями, а что предусловиями. Эвристически, если кто-то выбирает или вводит значение во время выполнения теста, то это следует рассматривать как входное значение, в ином случае это предусловие. Например, если тест-кейс проверяет авторизацию пользователя с разными именами-логинами, тогда такое имя будет входным значением. Если тест проверяет способность добавлять предметы в корзину после того, как залогинился определенный пользователь, тогда имя пользователя будет предусловием, т. к. оно не вводится напрямую в тесте, а авторизация должна быть выполнена перед началом теста.

6.1.5. Шаги выполнения

Теперь, когда и предусловия, и входные значения для тест-кейса определены, пришло время на самом деле запустить тест-кейс. Шаги, предпринятые во время выполнения теста, называются **шагами выполнения**, и они являются тем, что фактически делает тестировщик. Шаги выполнения очень часто невероятно специфичны, и весьма критично следовать им в точности. Сравните это с предусловиями, где достаточно получить конечный результат любыми средствами.

Начнем с простого примера. Мы тестируем интернет-магазин и проверяем, что добавление одного товара в пустую корзину отобразит "1" в качестве количества товаров в корзине. Предусловие заключается в том, что в корзине содержится нулевое количество товаров. Это может быть достигнуто разными путями: пользователь никогда не логинился раньше; пользователь уже залогинился и купил какое-то количество товаров, что привело к обнулению счетчика; или же любой из находившихся в корзине товаров был удален без совершения покупки. В данном случае не имеет значения, как этот результат (то есть корзина с нулевым количеством товаров) был достигнут, важно только то, что это выполнено.

С другой стороны, фактические шаги выполнения должны быть изложены предельно понятно:

1. Найти товар "SAMPLE-BOX" путем выбора текстового поля поиска, ввода SAMPLE-BOX и нажатия кнопки **Поиск**.

2. Должен отобразиться товар с названием SAMPLE-BOX. Нажмите кнопку с текстом "Добавить товар в корзину", расположенную рядом с картинкой SAMPLE-BOX.
3. Изучите надпись "Количество товаров в корзине = x" в верхней части экрана.

Обратите внимание, что эти шаги относительно явные. Очень важно записать шаги достаточно детально, чтобы в случае возникновения проблемы она была легко воспроизводима. Если в шагах выполнения было упомянуто "Добавьте товар", то наш тестировщик мог выбрать любой товар из доступных. Если проблема в выбранном товаре (скажем, добавление SAMPLE-PLANT, в отличие от SAMPLE-BOX, никогда не увеличивает количество товаров в корзине), то будет сложно разобраться, в чем же именно проблема. Соответствующий отчет о дефекте поможет смягчить эту проблему, но она может быть полностью предотвращена, только если гарантировано, что шаги выполнения определены правильно. Конечно, и здесь можно переусердствовать:

1. Переместите курсор на пиксел (170, 934) путем перемещения правой руки на 0,456 дюйма от предыдущего положения с использованием компьютерной мыши. Эта локация должна соответствовать текстовому полю с меткой "Искать".
2. Примените давление в течение 200 мс к левой кнопке мыши использованием указательного пальца правой руки.
3. После 200 мс быстро прекратите надавливать на левую кнопку мыши. Убедитесь, что курсор теперь видим и мигает с частотой 2 Гц в текстовом поле... (и т. д.).

В общем, лучше всего установить уровень спецификации, соответствующий способностям и знаниям людей, которые фактически будут выполнять тесты (или в случае автоматизированных тестов, программ, которые фактически будут выполнять тесты). Если тестировщики вашей компании хорошо знакомы и с программным продуктом, и с областью, в которой он работает, может оказаться вполне достаточным сказать: "Установи *фробинатор* в положение ‘FOO’ с использованием основного диска". Это достаточно определенно, чтобы знакомый с системой пользователь смог однозначно выполнить шаги. Однако не все знакомы с системой так, как автор тестов. Зачастую те, кто выполняет тесты, являются наемными сотрудниками, аутсорсерами или просто новичками в проекте. Для незнакомого с "фробинизацией" стороннего тестировщика (удивительно, что есть немного незнакомых с ней людей) может быть необходимо определить, что должно быть выполнено в деталях:

1. Откройте основную управляющую панель путем выбора **Набор > ... > Основной** в меню в верхней части экрана.
2. Выберите фиолетовый диск, обозначенный FROBINATOR. Переместите диск вправо относительно его начальной позиции до тех пор, пока в поле STATUS не появится текст FOO.

Да, и в конце надо заметить, что *фробинаторов* или *фробинизации* не существует.

6.1.6. Выходные значения

Значения, возвращаемые в ходе тестирования функциональности, называются **выходными значениями**. Когда дело касается исключительно математических функций, то их очень легко определить — математическая функция по определению принимает некоторое входное значение (или значения) и отдает некоторое выходное значение (или значения). Например, функция абсолютной величины принимает некоторое число x; и если $x < 0$, то она возвращает $–x$; в противном случае она возвращает x. При тестировании функции с –5 и проверке того, что она возвращает 5, является очевидным, что входным значением является –5, а выходным — 5. Нет никаких предусловий; подача –5 должна всегда возвращать 5, и не имеет значения, какие глобальные переменные установлены, не имеет значения, что именно в базе данных, не имеет значения, что отображается на экране. Нет никаких постусловий; функция не должна ничего отображать на экране, записывать что-то в базу данных или устанавливать глобальную переменную.

Но компьютерные программы не состоят исключительно из математических функций, и поэтому мы должны научиться различать постусловия и выходные значения.

6.1.7. Постусловия

Постусловием является любое условие, которое должно быть проверено после завершения шагов выполнения, и при этом не являться выходным значением. На постусловия тестируемая функциональность может не влиять напрямую, но постусловия напрямую вызываются тестируемой функциональностью. Предупреждающее сообщение на экране, какая-то записанная в базу данных информация, установленная глобальная переменная, продолжение работы системы или то, что поток не был уничтожен, — всё это примеры постусловий.

6.1.8. Ожидаемое поведение и наблюдаемое поведение

Хотя мы обсудили отличия между выходными значениями и постусловиями, на самом деле зачастую эти отличия не так важны, или же для понимания этих отличий нужно настолько погрузиться в теорию, что становится очевидным — оно того не стоит. Подобное можно сказать и о предусловиях и входных значениях.

Главное, что нужно держать в голове при написании тест-кейса:

```
Когда система находится в состоянии X
И выполняется действие Y,
Я ожидаю, что произойдет Z
```

Это значение `Z` и является сутью теста — это ожидаемое поведение. Невозможно протестировать что-то, если вы не знаете, что должно произойти. Как сказал Льюис Кэрролл, "если вы не знаете, куда вы идете, то вас приведет любая дорога". Аналогично при написании тест-кейса вам нужно знать, куда должен прийти тест-кейс, в противном случае невозможно проверить, что система оказалась там, где она должна быть.

6.2. Разработка тест-плана

Перед тем как приступать к написанию любого тест-плана, необходимо задуматься о конечной цели. Насколько детализированным должен быть тест-план? Какие виды граничных условий должны быть проверены? Какие существуют потенциальные риски неизвестных дефектов? Ответы на эти вопросы будут сильно различаться в зависимости от того, тестируете ли вы детскую онлайн-игру или же программу для мониторинга ядерного реактора. В зависимости от контекста и области тестируемого программного обеспечения даже для программ со схожими требованиями могут понадобиться различные стратегии создания тест-планов.

Простейший — и зачастую лучший — путь разработки детализированного тест-плана заключается в чтении требований и определении того, как их можно протестировать по отдельности. Обычно довольно много мыслей вкладывается в разработку требований, и поскольку целью системы является удовлетворение требований, имеет смысл убедиться, что все требования были реально протестированы. Это также открывает простой путь к созданию тест-плана. Итак, первое требование — напишем тест-кейсы; второе требование — напишем еще тест-кейсы; и будем повторять до тех пор, пока все требования не будут покрыты.

Для каждого требования необходимо продумать хотя бы один "счастливый путь" и создать хотя бы один тест-кейс для этого пути. То есть надо определить, в какой ситуации при рабочих параметрах будет удовлетворено это требование? Например, если у вас есть требование, чтобы некая кнопка была доступна для нажатия, только если значение меньше 10, и недоступна, если 10 и больше, то минимально вам понадобится создать тесты, проверяющие, что кнопка активна, если значение меньше 10, и неактивна, если значение больше либо равно 10. Этим тестируются оба класса эквивалентности требования (значение < 10 и значение ≥ 10).

Вы также можете продумать случаи, в которых помимо классов эквивалентности тестируются различные граничные условия. Продолжая рассматривать вышеприведенный пример, давайте предположим, что у вас есть тест-кейсы для значения 5 и для значения 15, что гарантирует вам использование по крайней мере одного значения для каждого класса эквивалентности. Если вы захотите добавить тест-кейсы, проверяющие границу между двумя классами эквивалентности, можете добавить тест-кейсы для 9 и 10.

Сколько таких граничных и угловых кейсов вы добавите и насколько широко вы протестируете внутренние и граничные значения, будет зависеть от времени и ресурсов, которые у вас есть для тестирования, от области ПО и уровня риска, который допустим для вашей организации. Помните, что исчерпывающее тестирование во всех отношениях невозможно. Существует скользящая шкала времени и энергии, которые можно потратить на написание и исполнение тестов, и не существует правильного ответа, что именно нужно выбрать. Нахождение компромисса между скоростью разработки и обеспечением качества является одной из ключевых задач тестировщика программного обеспечения.

Создание тест-кейсов для нефункциональных требований (атрибуты качества) к системе зачастую может быть сложным. Вам следует попытаться убедиться, что

требования сами по себе являются тестируемыми, и оценить количество тест-кейсов, которые нужны для этих требований.

К сожалению, просто наличие соответствия между требованиями и тест-кейсами для каждого их них не всегда означает, что вы разработали хороший тест-план. Вам, возможно, потребуется добавить дополнительные тесты, чтобы убедиться, что требования работают в тандеме, или проверить с точки зрения пользователя те ситуации, которые не связаны напрямую с требованиями или не вытекают из них. Более того, вам необходимо научиться понимать контекст, в котором существует программа. Обладание профильными знаниями в области работы поможет вам понять основные рабочие ситуации, то, как система взаимодействует с внешним миром, возможные проблемы, и как пользователь может ожидать восстановления системы после возникновения этих проблем. Если никто в команде не понимает область работы программы, возможно, следует обсудить вопросы со специалистом (subject matter expert, SME) перед написанием тест-плана.

Понимание программной среды, в которой создается ПО, также может способствовать написанию тест-плана, хотя технически это приведет к тестированию серого ящика, в отличие от тестирования черного ящика, потому что вы, как тестировщик, будете знать некоторые особенности внутренней реализации, но это может дать ценное понимание того, где могут скрываться потенциальные ошибки. Позвольте мне привести пример. В Java деление на ноль, как показано ниже, выбросит исключение `java.lang.ArithmeticException`:

```
int a = 7 / 0;
```

Не имеет значения, какое у нас делимое, потому что если делитель равен нулю, выбрасывается исключение `java.lang.ArithmeticException`:

```
// Во всех этих случаях выбрасывается одно и то же исключение
int b = -1 / 0;
int c = 0 / 0;
int d = 999999 / 0;
```

Следовательно, при тестировании написанной на Java программы вы можете предположить, что деление на ноль, по существу, является одним классом эквивалентности; если это произошло, то затем будет происходить одно и то же событие, каким бы оно ни было (например, возможно, что исключение перехвачено и сообщение "Ошибка деления на ноль" выведено в консоль).

JavaScript (да, технически я имею в виду ECMAScript 5 — для тех, кто хочет знать подробности) не выбрасывает исключение во время деления на ноль. Однако если знаменатель равняется нулю, то в зависимости от числителя вы можете получить разные результаты!

```
> 1 / 0
Infinity

> -1 / 0
-Infinity
```

```
> 0 / 0
NaN
```

Деление положительного числа на ноль возвращает бесконечность, деление отрицательного числа — "минус" бесконечность, а деление нуля на ноль — NaN (Not a Number — не число). Это означает, что деление на ноль, несмотря на то, что является одним "внутренним классом эквивалентности" для Java-программ, оказывается тремя разными классами для программ, написанных на JavaScript. Зная это, вы, возможно, захотите протестировать программу и убедиться, что она может обработать все эти возвращаемые значения, и не предполагать, что вы проверили все граничные случаи просто потому, что вы проверили деление на ноль. Это реальный пример из написанного мною тест-плана, и при его использовании было найдено несколько дефектов.

6.3. Тестовые фикстуры

Во время написания вашего плана вы можете захотеть протестировать ситуации, которые сложно воспроизвести. Например, вы можете захотеть проверить, что рассмотренное выше приложение для проверки температуры кофе работает при смене временны́х зон. Будет бессмысленно дорого реально перемещаться из одной временной зоны в другую. Вспомните, что вы повелитель этого мира тестирования! Вы можете просто изменить временную зону системы, в которой запущена программа. Если вы тестируете программу, которая будет работать в России, можете просто поменять настройки локали на Россию вместо того, чтобы срываться на рейс. Если вам нужны десять пользователей в базе данных для тестирования, можете просто добавить их вручную. Хотя эти фейковые ситуации могут не охватить все дефекты, способные произойти в реальности, они помогут обнаружить многие из них.

Скрипт или программа, используемые для перевода тестируемой системы в состояние готовности для тестирования, называется **тестовой фикстурой**. Текстовые фикстуры могут быть простыми и состоять из последовательности шагов, которые нужно добавить в программу, но ничто не ограничивает их сложность. Аппарат для тренировки посадки на Луну управлялся астронавтами на Земле, и в его работе использовался сложный механизм обратной связи для симуляции лунной гравитации. Для того чтобы изучить больше примеров о том, как тестирование и тестовые фикстуры помогали астронавтам добраться на Луну, обратитесь к книге Дэвида Минделла "Цифровой Аполлон: Человек и машина в космическом полете" (David Mindell "Digital Apollo: Human and Machine in Spaceflight").

Тестовые фикстуры часто используются для симуляции внешних систем. История из личного опыта: я тестировал подсистему, которая взаимодействовала с другими подсистемами через JSON. Поначалу эти другие системы настраивались вручную перед каждым тест-кейсом. Вскоре я осознал, что это отнимало много времени и зачастую приводило к ошибкам. Решением стало использование гема `simple_respond` языка Ruby, который принимал заданный JSON-файл и в ответ на любой запрос всегда возвращал данные этого файла. Вместо того чтобы заниматься настройкой

других подсистем, которые я не тестировал, я мог сконцентрироваться на том, как будет функционировать *моя* тестируемая подсистема. Это не только сэкономило мне время и уменьшило количество зависящих от человеческого фактора ошибок — тесты перестали зависеть от правильной работы других частей системы. Подобные тестовые фикстуры могут также быть переиспользованы при взаимодействии с внешними системами, когда нельзя модифицировать их состояние для заданного тест-кейса.

6.4. Выполнение тест-плана

Выполнение тест-плана называется **прогоном** (проходом, тест-раном). Прогон можно представить как эквивалент объекта и класса. Выполнение тест-плана создает прогон подобно тому, как при работе с классом мы можем создать объект. Тест-план является картой, показывающей, куда можно пойти, в то время как прогон подобен путешествию.

Выполнение тест-плана должно быть сравнительно простым процессом, предполагающим, что тест-план разработали надлежащим образом. В конце концов, вы потратили время, чтобы убедиться, что все предусловия выполнимы, что входные значения заданы, что шаги выполнения достаточно детализированы, а выходные значения и постусловия могут быть протестированы. На данном этапе выполнение тест-кейсов будет сравнительно механическим процессом (и это одна из причин появления автоматизированного тестирования). Вы можете отправить кого-то в помещение, где находится компьютер, на котором можно запустить программу, и спустя несколько часов, в зависимости от длительности тест-плана, этот человек выйдет из помещения с полностью протестированной системой.

К сожалению, это прекрасное видение не всегда становится реальным. В процессе выполнения тест-кейса этот тест-кейс может получать различные статусы. В итоге тест-кейс получит некий финальный статус, хотя во время прогона этот статус будет изменяться. Существуют также статусы "нулевой" или "не протестирован", которые означают, что этот данный тест-кейс еще не был выполнен.

Хотя не существует универсального хранилища статусов, можно привести репрезентативную выборку тест-кейсов, с которой вы можете встретиться в своей работе тестировщика. Названия могут меняться, но эти шесть типов обеспечивают хорошее покрытие ситуаций, в которых может оказаться ваш тест-кейс:

1. Пройден (Passed).
2. Неудавшийся (Failed).
3. Остановлен (Paused).
4. Запущен (Running).
5. Заблокирован (Blocked).
6. Ошибка (Error).

Пройденный тест является тем, в котором всё ожидаемое поведение (т. е. выходные значения и постусловия) соответствует наблюдаемому поведению. Проще говоря, это тест, где всё работает.

Напротив, **неудавшийся** тест является тем тестом, в котором по крайней мере одна из составляющих наблюдаемого поведения не соответствует ожидаемому поведению. Это различие может быть в выходных значениях или постусловиях. Например, если функция вычисления квадратного корня возвращает значение квадратного корня четырех, равное 322, то этот тест-кейс должен быть помечен как неудавшийся. Если у тест-кейса было постусловие, что на экране должно появиться сообщение "ОШИБКА: СЛОНЫ НЕ МОГУТ ТАНЦЕВАТЬ", а на экране сообщение об ошибке гласит "ОШИБКА: СЛОНЫ НЕ МОГУТ ВЫБРАСЫВАТЬСЯ ИЗ ОКНА", то этот тест-кейс также является неудавшимся. Всякий раз, когда тест-кейс помечается неудавшимся, должен быть зарегистрирован соответствующий дефект. Это может быть новый дефект или же известный дефект, вызывающий множество проблем, например ошибки для всех животных с утверждением, что они не могут выбрасываться из окна, в то время как правильным будет сообщение, что они не могут танцевать. Если нет дефекта, связанного с неудавшимся тест-кейсом, то либо тест-кейс не был достаточно важен для тестирования, либо найденный дефект недостаточно важен для документирования. Если это так, вам надо переосмыслить ваш тест-кейс!

Остановленный тест является тестом, который был запущен, но затем поставлен на удержание на какой-то период времени. Это позволяет другим тестировщикам и менеджерам знать статус теста и прогресс, которого достиг тестировщик. Это также гарантирует, что другой тестировщик не начнет выполнять тест, который уже выполняется. Тест-кейс может быть остановлен по банальным причинам — скажем, тестировщик отправился перекусить или занялся чем-то, связанным с тестируемой системой (например, покинул помещение для получения новых тестовых данных). В любом случае предполагается, что тестировщик продолжит работу над тестом после своего возвращения, но не означает, что тест сам по себе не может быть выполнен (это покрывается статусом "Заблокирован", рассматриваемым ниже).

Запущенный тест является тестом, который начат, но пока еще не завершен, и таким образом, конечный результат пока неизвестен. Данный статус обычно используется в случаях, когда выполнение теста занимает значительное время и тестировщик хочет дать понять другим тестировщикам, что тест находится в стадии выполнения. Хотя технически все тесты находятся в состоянии "запущен" короткий период времени (когда тестировщик выполняет шаги выполнения), и если нет какой-либо автоматизации, то этот статус обычно присваивается долго исполняемым тестам.

Иногда тест невозможно выполнить в данный момент. Причиной этого могут быть внешние факторы (например, в связи с недоступностью части тестового оборудования) или внутренние (скажем, часть функционала не доработана, или невозможно тестировать в связи с дефектами, присутствующими в системе). В таких случаях тест может быть помечен как **заблокированный**. Это означает, что тест не может быть запущен в настоящее время, хотя к нему можно вернуться во время будущих прогонов, когда разрешатся проблемы, препятствующие его запуску.

Наконец, в некоторых случаях тест-кейс просто не может быть выполнен сейчас или в будущем в связи с проблемой в самом тест-кейсе. В таких случаях статус тес-

та может быть отмечен как **"ошибка"**. Помеченные ошибочными тесты могут вступать в противоречие с требованиями: например, в требовании может говориться о том, что фон веб-страницы должен быть синим, в то время как тестируемое приложение является работающей с командной строкой консольной программой. Проблемой может быть, если, скажем, ожидаемое поведение говорит о том, что квадратный корень из 25 должен выдавать результат "пудель". Иногда причиной пометки теста ошибочным может оказаться простая опечатка, но чаще это указывает на фундаментальную проблему с пониманием командой разработчиков или командой тестировщиков разрабатываемого программного обеспечения. Тест-кейсы с пометкой "ошибка", в отличие от тест-кейсов, помеченных заблокированными, не должны запускаться, пока эта ошибка не будет устранена.

6.5. Отслеживание тестовых прогонов

Хотя вы можете выполнять тест-план для удовольствия или ради желания самосовершенствоваться, в большинстве случаев вам захочется записать результаты тест-плана. Это можно сделать при помощи специального программного обеспечения, электронной таблицы или даже блокнота. В некоторых случаях это требуется нормативами среды, но даже если не требуется, отслеживание того, какие тесты прошли, а какие нет, будет очень полезным.

При отслеживании тестового прогона есть несколько информационных разделов, которые вы захотели бы включить:

1. Дата исполнения теста.
2. Имя или другой идентификатор (т. е. логин или ID-номер) тестировщика.
3. Название или другой идентификатор тестируемой системы.
4. Указатель того, какой код тестировался. Это могут быть тег, ссылка, номер версии, номер сборки или какая-то другая форма идентификации.
5. Тест-план, к которому относится тестовый прогон.
6. Итоговый статус каждого тест-кейса. Обратите внимание, что временные статусы, такие как "Остановлен", должны быть изменены на итоговый статус перед завершением тестового прогона.
7. Список всех дефектов, задокументированных в результате выполнения тест-кейса в случае их обнаружения, или же разъяснение причины того, почему тест имеет иной статус, отличный от "Пройден".

Пример тестового прогона может выглядеть так:

```
Дата: 21 мая 2014 г.
Имя тестировщика: Jane Q. Tester
Система: Meow Recording System (MRS)
Номер сборки: 342
Тест-план: Тест-план Meow Storage Subsystem
```

```
Результаты:
TEST 1: Пройден
TEST 2: Пройден
TEST 3: Неудавшийся (записан дефект #714)
TEST 4: Заблокирован (дополнительная функция программы пока не реализована)
TEST 5: Пройден
TEST 6: Неудавшийся (причина в известном дефекте #137)
TEST 7: Ошибка (очевидная ошибка в тест-плане; необходимо проверить с отделом
системного инжиниринга)
TEST 8: Пройден
TEST 9: Пройден
```

Если тест не проходит, должно произойти одно из двух событий: либо заполняется информация о новом дефекте, либо отмечается, что причиной проблемы стал уже известный дефект. Дефект означает, что система не действует, как запланировано; ожидаемое поведение не соответствует наблюдаемому поведению. Больше о документировании дефектов вы узнаете в следующей главе.

Если тест заблокирован, должно быть указано, из-за чего произошла блокировка. Причина может лежать вне зоны ответственности команды тестирования (например это может быть из-за того, что новая возможность программы еще не реализована), или существует что-то, что может быть улучшено (например, при отсутствии соответствующего оборудования). Объяснение причины незавершенности теста и включение этого объяснения в результаты тестового прогона будет полезно не только при документировании статуса, но и позволит руководителям или другим сотрудникам решить проблему в будущем. Например, если часть тест-плана требует наличия какого-то особого оборудования, то задокументированная информация о том, что из-за его нехватки нельзя пройти тесты, может побудить руководство к закупке оборудования или инженеров к поиску альтернативных решений.

Хочется уповать на то, что тесты со статусом "Ошибка" будут редкостью. Впрочем, если найден ошибочный тест, тестировщику надлежит отметить, почему он или она считает этот тест ошибочным. Соображения по исправлению (или по крайней мере по получению дополнительной информации) следует включить в результат данного тест-кейса.

Отслеживание тестовых прогонов позволяет вам увидеть, где падают ваши тесты. Если вы заметите, что какая-то часть программы падает чаще других, возможно, вам следует сфокусироваться на этом. И наоборот, если тесты всегда успешно проходят (и так продолжается годы), они могут быть недостойны вашего внимания. Также отслеживание поможет вам увидеть, где сбои проявляются время от времени, и таким образом вы узнаете, какие тест-кейсы нестабильны. Просмотр тестовых прогонов спустя некоторое время не только позволит вам оценить прогресс и качество программного обеспечения, но и даст понимание, как в дальнейшем лучше создавать следующие тест-планы.

6.6. Матрицы трассируемости

Теперь, когда у нас есть список требований и тест-план для их тестирования, что еще остается? Конечно, можно отправиться домой и насладиться любимым напитком после того, как целый день маялся в шахтах данных. Но есть кое-что еще, что можно обсудить в этом разделе. Мы неформально разработали тесты, которые, как мы полагаем, удовлетворяют требованиям, но мы можем перепроверить, что наши требования и тест-план синхронизированы, путем создания **матрицы трассируемости**.

Матрица трассируемости является простым способом определить, какие требования совпадают с тест-планами, и отобразить это на понятной диаграмме. Она состоит из списка требований (обычно просто идентификаторов требований) и списка номеров тест-кейсов, которые соответствуют этим требованиям (т. е. тех, что тестируют конкретные аспекты данного требования).

В качестве примера вернемся к спецификации требований для приложения по измерению температуры кофе. Вы обратите внимание, что список требований немного изменился — и это нормально при разработке программ!

- *FUN-COFFEE-TOO-HOT*. Если измеренная температура кофе составляет 80 °C и выше, то приложение должно отображать на дисплее сообщение "СЛИШКОМ ГОРЯЧО".
- *FUN-COFFEE-JUST-RIGHT*. Если измеренная температура кофе меньше 80 °C, но больше 55 °C то приложение должно отображать на дисплее сообщение "САМОЕ ТО".
- *FUN-COFFEE-TOO-COLD*. Если измеренная температура кофе составляет 55 °C и меньше, то приложение должно отображать на дисплее сообщение "СЛИШКОМ ХОЛОДНО".
- *FUN-TEA-ERROR*. Если жидкостью, температуру которой измеряют, является чай, то приложение должно отображать на дисплее сообщение "ИЗВИНИТЕ, ЭТО ПРИЛОЖЕНИЕ НЕ ПОДДЕРЖИВАЕТ РАБОТУ С ЧАЕМ".

Мы запишем идентификаторы требований и оставим пространство для идентификаторов тест-планов:

```
FUN-COFFEE-TOO-HOT:
FUN-COFFEE-JUST-RIGHT:
FUN-COFFEE-TOO-COLD:
FUN-TEA-ERROR:
```

Теперь давайте посмотрим на завершенный тест-план и определим, какие тест-кейсы соответствуют тестированию заданных требований. Для каждого подходящего тест-кейса запишем его идентификатор рядом с требованием:

```
FUN-COFFEE-TOO-HOT: 1, 2
FUN-COFFEE-JUST-RIGHT: 3, 4, 5
FUN-COFFEE-TOO-COLD: 6, 7
FUN-TEA-ERROR: 8
```

Легко увидеть, что для каждого требования есть по крайней мере один покрывающий его тест. Если бы было другое требование, скажем:

- *FUN-COFFEE-FROZEN*. Если кофе находится в твердом, а не жидком состоянии, то приложение должно отображать на дисплее сообщение "ЭТОТ КОФЕ МОЖЕТ БЫТЬ ТОЛЬКО СЪЕДЕН, НО НЕ ВЫПИТ",

и мы попытались бы создать матрицу трассируемости, то было бы легко увидеть, что для проверки этого требования тестов нет:

```
FUN-COFFEE-TOO-HOT: 1, 2
FUN-COFFEE-JUST-RIGHT: 3, 4, 5
FUN-COFFEE-TOO-COLD: 6, 7
FUN-TEA-ERROR: 8
FUN-COFFEE-TOO-FROZEN:
```

Точно так же матрицы трассируемости могут позволить нам определить, есть ли у нас "бесполезные" тесты, которые не тестируют ни одного из требований. Например, представим, что мы создали "Тест-кейс 9":

```
ИДЕНТИФИКАТОР: 9
ТЕСТ-КЕЙС: определить, правильно ли приложение показывает температуру пуделя.
ПРЕДУСЛОВИЕ: пудель живой и в добром здравии, с нормальной для пуделя температурой 38 °C.
ВХОДНЫЕ ДАННЫЕ: Нет.
ШАГИ ВЫПОЛНЕНИЯ: навести датчик на пуделя на пять секунд. Прочитать значение
с дисплея.
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет.
ПОСТУСЛОВИЯ: на экране демонстрируется сообщение "С пуделем всё в порядке".
```

В нашей матрице трассируемости снова появляется пробел, но на этот раз на стороне требований. Тест-кейс 9 не соответствует ни одному из требований, и это будет ненужный тест:

```
FUN-COFFEE-TOO-HOT: 1, 2
FUN-COFFEE-JUST-RIGHT: 3, 4, 5
FUN-COFFEE-TOO-COLD: 6, 7
FUN-TEA-ERROR: 8
???: 9
```

Время от времени в "реальном мире" могут найтись тесты, которые не подходят к какому-либо определенному требованию. Например, если системный инженер не создал требование по надежности, тест-план все равно может включать тест, позволяющий убедиться, что система работает, даже если она запущена целый день. Это определенно не наилучшая практика, но такое иногда встречается. Если это произошло, лучшим выходом станет создание требования по надежности, которое можно протестировать.

Конечно, матрица трассируемости дает очень простой обзор тестового покрытия. Тот факт, что каждое требование было протестировано, не означает, что каждое требование было протестировано тщательно. Например, что если у системы есть проблемы с чрезвычайно горячим кофе? Наибольшая температура, которую мы проверяли, составляла 93 °C, но проблема может возникнуть при температуре

94 °C. Также в матрице трассируемости нет проверки, что наши тесты хорошие. Если мы тестировали, соответствует ли система требованию FUN-COFFEE-TOO-HOT путем помещения системы в ледяную воду, и при этом утверждаем, что тест-кейс соответствует требованию FUN-COFFEE-TOO-HOT, то об этом никак нельзя сказать по матрице трассируемости.

Матрицы трассируемости являются хорошим способом перепроверить вашу работу и сообщить другим людям за пределами вашей команды, насколько хорошо покрыта система с точки зрения тестирования. Если время поджимает, вы или ваш менеджер можете решить, что определенные части или функции системы более важны для тестирования, чем другие, и поэтому вы можете даже не писать тесты для этих менее важных частей. Опять-таки, это не наилучшая практика, но, по крайней мере, вы можете использовать матрицу трассируемости для отслеживания пробелов в вашем тестовом покрытии.

Потребителям и менеджменту, особенно в таких зарегулированных отраслях, как оборона и медицина, также могут потребоваться матрицы трассируемости как способ доказать, что системы были протестированы по крайней мере на базовом уровне.

Глава 7

Ломая программу

До этого момента мы фокусировались на разработке тест-планов и лишь немного упомянули о вопросе, что именно надо тестировать помимо необходимости удостовериться в соответствии программного обеспечения требованиям. Хотя это не самый худший подход к тестированию ПО, он пропускает слишком много важных, связанных с верификацией аспектов, особенно по проверке незаметных граничных случаев.

Причина тестирования — в поиске дефектов; чтобы найти дефекты, вам нужно решиться сойти со счастливого пути, гарантирующего, будто все вводят данные в том формате, который вам необходим, у систем неограниченная память и ресурсы процессора, а ваши компьютерные сети никогда не подвержены сбоям. Вам необходимо переместиться в темные леса, где люди пытаются ввести "СОРОК СЕМЬ" вместо 47, где у программ заканчивается память в разгар вычислений, а злодей из фильмов ужасов держит в руках заточенный топорик около кабеля компьютерной сети.

Впрочем, последнее вряд ли возможно. Но вам все же придется подумать о путях тестирования системы, когда что-то может оказаться не совсем несовершенным. Рассматривайте эту главу как путь тренировки вашего мышления тестировщика по созданию проблем и нахождению трещин в тестируемой программе.

7.1. Ошибки, которые следует искать

1. **Логические ошибки.** Логическая ошибка является ошибкой в логике программы. Разработчик понимал, что нужно сделать, но в процессе преобразования системы от описания до имплементации что-то пошло не так. Это могло быть чем-то простым и вызванным случайной заменой "больше чем" (>) на "меньше чем" (<) или же сложным и вызванным запутанным взаимодействием между многочисленными переменными.

   Для поиска логических ошибок следует убедиться, что ожидаемый результат достигается для различных входных значений. Границы — явные и неявные — зачастую оказываются "золотой жилой" для поиска дефектов. Также попробуйте другие виды интересных входных значений, такие как спецсимволы, чрезвычай-

но длинные строки и неправильно отформатированные данные. Входные данные, поступающие от других систем или пользователей, часто содержат странные или некорректные данные. Неплохо знать, что произойдет, когда ваша система получит их.

Для сложных выходных значений (например, генерация большой веб-страницы из шаблона) может оказаться невыполнимой проверка этих значений напрямую. Однако вы можете проверить наличие допустимых свойств выходных значений, и то, что эти свойства ожидаются. Например, что данные сгенерированной страницы могут быть разобраны, и что она отображает информацию корректно. Вы также можете взглянуть на данные с более высокого уровня абстракции — вместо проверки, что в HTML-коде тег `<strong>` оборачивает каждое встречающееся слово "кошка", может оказаться проще взглянуть на страницу, найти слова "кошка" и убедиться, что все они выделены полужирным.

2. **Ошибки на единицу.** Хотя технически это логические ошибки, они встречаются настолько часто, что их стоит отметить отдельно. Ошибка на единицу — это ситуация, когда программа делает что-то неправильно, потому что значение неверно всего на единицу. Это было причиной внимания при определении граничных значений в предыдущей главе — граничные значения являются целенаправленным методом поиска ошибок на единицу. Почему они такие распространенные? Представим очень простой метод, который определяет, является ли человек несовершеннолетним:

```
public Boolean isMinor(int personage) {
    if (personage <= 18) {
        return true;
    } else {
        return false;
    }
}
```

Вы заметили ошибку? По крайней мере, в Соединенных Штатах на момент написания вы больше не считались несовершеннолетним в момент, когда вам исполнилось 18 лет. Из-за использования `<=` вместо `<` метод вернет, что человек несовершеннолетний, даже если ему исполнилось 18. Подобная незначительная ошибка может возникнуть в самых разных случаях — из-за того, что кто-то решил, что индексы массива начинаются с 1, а не с 0, что кто-то перепутал "больше или равен" с "больше", кто-то использовал `++i` вместо `i++` и т. п. Такие ошибки зачастую менее заметны, чем другие, т. к. они проявляются только в случае конкретных значений. При тестировании проверяйте граничные значения, и вы, скорее всего, обнаружите какие-то из "ошибок на единицу".

3. **Ошибки округления и ошибки плавающей запятой (точкой).** Компьютерные системы часто используют переменные с плавающей запятой для представления десятичных чисел (например, `1.1`). Зачастую это гораздо более эффективно, чем использовать значения произвольной точности или рациональные числа, но эффективность достигается ценой потери точности. Например, представим, что

тестируемая система использует стандарт IEEE 754 для хранения числа одинарной точности с плавающей запятой (т. е. 32-битного значения). Указывание `1.1` в качестве значения не означает, что оно и хранится как `1.1`. Вместо этого оно сохраняется как `1.100000023841857910156625`, потому что существует буквально бесконечное количество дробных чисел, и все их нужно сохранить в 32 битах. Если вы попытаетесь связать неограниченное количество значений с ограниченным пространством, то окажется, что некоторые значения имеют одинаковое представление. Если вы помните дискретную математику, то это принцип Дирихле (или "принцип ящиков и голубей" — pigeonhole principle) в действии. Это значение оказалось наиболее близким к `1.1`. Сопоставление различных значений одному и тому же представлению по сути говорит, что значения будут округлены.

"И хорошо, — можете подумать вы. — Это очень небольшое различие между настоящим значением и его представлением. Я не могу представить себе, чтобы оно могло как-то оказать влияние". А что произойдет, если вы умножите его на другое число с плавающей запятой? Это различие может увеличиться. А затем вы умножите результат еще на одно число с плавающей запятой, а потом еще... И в итоге вы будете иметь дело со значениями, которые будут совсем не близки к тем, которые должны быть! Хотя постепенный дрейф значений может и не произойти, но поскольку некоторые значения-представления могут быть больше или меньше действительных значений, такая ситуация вполне вероятна.

Это является одной из причин, почему во многих языках программирования присутствует тип данных `Currency` (финансовый); представьте, если в банке подобные калькуляции окажутся неверными. Использование значений с плавающей запятой будет стоить им немало средств. И всё может оказаться даже еще хуже, и тут не надо ничего придумывать: почитайте про трагедию с ракетой Patriot в 1991 году. Накопившиеся ошибки плавающей запятой привели к неспособности американской ПВО во время войны в Персидском заливе обнаружить атаку ракетой Scud, что стало причиной гибели 28 человек. Еще около сотни получили ранения.

В некоторых случаях допустима осознанная потеря точности из-за отбрасывания ряда чисел или когда программа при расчетах округляет число вверх или вниз. Например, в какой-то момент программе может потребоваться сконвертировать десятичную дробь в целое число, и у программиста есть несколько способов, как сделать это. Преобразование может идти вниз (т. е. `4.8` становится `4` путем отбрасывания всего, что находится справа от десятичного разделителя), оно может идти вверх (т. е. преобразовывая `4.3` в `5`), или же можно воспользоваться хорошо знакомым со школы округлением, когда `4.5` или большее округляется вверх, а меньшее, наоборот, вниз. Какой путь использовать? Это зависит от программы и ожидаемого результата, и довольно легко выбрать неправильный путь.

Для того чтобы осуществить проверку на ошибки округления и ошибки плавающей запятой, попробуйте провести тестирование с другими десятичными значениями в качестве входных данных. Убедитесь, что конечные выходные

значения фактически соответствуют ожидаемым выходным значениям с учетом допустимой погрешности. Если входные данные могут взаимодействовать с другими данными, проверьте, что происходит при использовании больших объемов данных. Подобные виды ошибок будут накапливаться и таким образом становиться более уловимыми.

4. **Интеграционные ошибки.** Когда ошибки существуют в интерфейсе между двумя различными частями системы, они известны как интеграционные ошибки. Эти интерфейсы могут быть границами классов, границами пакетов или межпроцессными границами, вплоть до границ между большими мультикомпьютерными системами. Так как интерфейсы часто находятся там, где существуют разделения между командами или людьми, работающими с различными частями системы, эти интерфейсы окажутся средой, богатой на дефекты. В командах обычно лучше налажена коммуникация между членами команды; в конце концов, они работают над схожими частями системы и получают быструю обратную связь всякий раз, когда возникает проблема при работе с их конкретной подсистемой. При взаимодействии с другими командами возникновение недопонимания более вероятно, и это создает возможности для перепроверок, что система работает корректно, а сделанные предположения верны. Даже если имеется хорошо определенная спецификация интерфейса, существуют вероятности недопонимания спецификации или ошибок во время ее реализации.

   Для тестировщика фокус на тестировании того, как системы интегрированы между собой, принесет большие дивиденды. Разработчикам зачастую сложно гарантировать то, что системы корректно взаимодействуют между собой, т. к. они, как правило, заняты какой-то отдельной частью разрабатываемой программы и не имеют всеохватывающего взгляда на систему.

5. **Ошибки предположений.** Практически невозможно полностью описать большинство систем при помощи требований. Если бы вы определили систему настолько точно, то это значит, что вы практически бы написали программу. Поэтому разработчики часто делают предположения о том, как программа должна функционировать. Однако эти предположения могут не совпадать с запросами потребителя или с тем, какого поведения от этой системы ожидают другие системы. Можно привести несколько распространенных примеров для проверки.

   - Как системе следует отображать ошибки?
   - Как данные должны отображаться или выводиться другим способом?
   - Существуют ли какие-то требования по форматированию файлов?
   - Какие системы должны поддерживаться?
   - С какими системами будет осуществляться взаимодействие?
   - Какие виды интерфейсов необходимы?
   - Какими должны быть пользовательский опыт и пользовательский интерфейс?
   - Как будет осуществляться доступ к системе? Кем?
   - Какие будут использоваться терминология, акронимы и т. п.?

- Каковы допустимые диапазоны или пределы для данных?
- Как будут вводиться данные? В каких форматах?

Если вы предполагаете, что все веса вводятся в фунтах, что произойдет, если кто-то другой сделает предположение о килограммах? Было сделано предположение, что выходные данные отображаются в ASCII, но какая-то часть данных была введена в UTF-8, что привело невозможности отобразить часть информации на экране? Разработчик написал программу, в которой используется интерфейс командной строки, а заказчик хотел графический интерфейс? Выходные файлы записаны в формате CSV (значения, разделенные запятыми), а конечные потребители ожидали использования табуляции в качестве разделителя? Все эти сделанные в процессе разработки предположения могут привести к дефектам во время использования системы.

Также могут существовать общие требования для заданной области, о которых разработчик может не подозревать. В конце концов, разработчики, как правило, являются разработчиками, а не экспертами в той области, для которой они создают программное обеспечение. Конечно, это не всегда так, но очень часто будет существовать разрыв между знаниями человека, занимающегося разработкой системы, и представлениями конечного потребителя.

Как тестировщик ПО, вы можете и должны помогать сократить этот разрыв. Понимание требований пользователя и того, как разработчики пишут программы, может позволить вам увидеть расхождения и узнать, на что следует смотреть при разработке тест-планов. Это также поможет вам расставить приоритеты и продумать стратегию при определении того, какие функции и граничные случаи необходимо проверить. Например, если вы знаете, что размер некоего типа файлов, с которыми работает система, не превышает 50 Кбайт, вы можете сконцентрироваться на тестировании небольших файлов вместо того, чтобы браться за граничные случаи, в которых рассматриваются очень большие файлы.

6. **Ошибки отсутствующих данных.** Каждый раз, когда данные приходят из внешнего источника в программу (например, CSV-файла, сервиса API или напрямую от пользователя, который вводит данные в терминале), существует вероятность того, что необходимые данные будут отсутствовать. Это может произойти как по такой простой причине, когда пользователь случайно нажал клавишу <Enter>, так и по довольно сложной, например где-то внутри огромного JSON-ответа отсутствовал нужный атрибут. Хотя в любом случае система должна обрабатывать такие ситуации надлежащим образом. Однако всегда допускайте, что внешний источник может не предоставить все данные, которые вам необходимы.

   Дальнейшие действия, когда данные отсутствуют, зависят от программы и области ее применения. В некоторых случаях программа может благополучно проигнорировать такую ситуацию; в других случаях ситуацию нужно отметить или записать в лог-файл, или же можно показать предупреждающее сообщение пользователю; в редких случаях правильным решением может оказаться выключение всей системы. Однако вы должны знать, как система отреагирует на от-

сутствие данных, и убедиться, что она действует так при возникновении подобной ситуации.

7. **Ошибки плохих данных.** Ошибки плохих данных вызывают еще больше проблем, чем ошибки отсутствующих данных. В то время как при поиске последних бывает достаточно проверить один элемент на наличие определенного атрибута, существует бесконечное количество способов для данных оказаться "плохими". Эти данные могут быть сгенерированы внутри системы, но более вероятно появление плохих данных из внешних систем, которые имеют различные допущения, используют разные форматы, были повреждены или модифицированы каким-либо образом и т. д.

   - *Данные слишком большие.* Возможно, система может обработать данные до определенного размера. Что произойдет, если размер входных данных окажется больше?
   - *Данные слишком короткие.* С другой стороны, что если система странно обрабатывает малые объемы данных? Это не такая частая проблема, как проблема больших данных, но зачастую она является причиной неэффективности (например, система всегда резервирует мегабайт памяти по умолчанию для входных данных, даже если их размер составляет всего несколько байт).
   - *Данные отформатированы некорректно.* Что произойдет, если программа ожидает данные, разделенные запятой, а получает данные с табуляцией в качестве разделителя? Что произойдет, если ваша программа ожидает строку JSON, а получает XML?
   - *Данные находятся вне допустимого диапазона.* Что произойдет, если программу запросят информацию о 593 штате Америки? Что произойдет, если указанная температура составит –500 °C (что ниже абсолютного нуля и соответственно не встречается в этой Вселенной)?
   - *Данные были повреждены.* Существуют ли какие-то гарантии при приеме данных, позволяющие сказать, что эти данные правильные и не были модифицированы? Хотя проблема повреждения данных не так актуальна, как раньше, все равно возможно возникновение ошибок. Например, если кто-нибудь откроет файл в текстовом редакторе другой операционной системы, что приведет к конвертации символов переноса строки в неожиданные знаки?
   - *Несогласованные данные.* Что произойдет, если входные данные противоречат сами себе? Например, если вы получаете данные с двумя записями для ID 723, и в одной из них указывается имя John Doe, а в другой — Jane Doe? Вы знаете, какое должно быть ожидаемое поведение в этой ситуации?
   - *Данные не поддаются анализу.* Что произойдет, если вы получаете данные, которые не могут быть разобраны из-за отсутствия закрывающего символа > или) или в которых содержится циклическая ссылка? Выдаст ли система соответствующее сообщение об ошибке или поймет, что оказалась застрявшей в бесконечной петле?

Отслеживание проблем плохих данных может быть сложным, т. к. существует масса возможностей возникновения ошибок, и зачастую только весьма специфическая конфигурация плохих данных может вызвать проблему. Хотя, конечно, можно подготовить данные, содержащие всевозможные ошибки, но более эффективно использовать **нечеткое тестирование** (или **фаззинг**, от англ. *fuzz testing*) — отправлять случайно сгенерированные данные (которые могут соответствовать или не соответствовать ожидаемым входным данным) и убеждаться, что система продолжает работать правильно. *См. главу 18* о стохастическом тестировании, чтобы узнать больше о фаззинге.

8. **Ошибки отображения.** Даже если система вычислит корректное значение, оно может быть отображено неверно. Такая проблема может возникнуть, если число или строка слишком длинные для отображения полностью (например, результат деления 1 на 3) и часть символов отбрасывается. В других случаях отбрасывания может не быть, и данное значение может "наползти" на другой текст или вызвать искажения вывода на экране прочей информации. Если вы отображаете значения на диаграмме, выбросы по осям *x* и *y* могут не попасть на экран, или же если попадут, станут причиной проблем с отображением других значений. График может выводиться некорректно, также возможно использование неверной битовой карты или цветов. Попытка вывести некоторые символы на дисплей может привести к зависанию терминала, включению сигнала динамика или к каким-либо проблемам с дисплеем. Неправильно обработанный HTML-код станет причиной того, что ваша веб-страница перестанет отображаться.

   Каждый раз, когда отображаются данные, проверьте не только то, что значение было вычислено правильно, но и то, что оно отображается соответственно. Используя данные, которые содержат необычные символы либо чрезвычайно большие или малые значения либо вообще не содержат ожидаемых значений, вы можете быть уверены, что символ, который нужно отобразить на экране, — это действительно тот символ, который увидит пользователь.

9. **Ошибки внедрения (инъекций).** Ошибки внедрения, которые являются подмножеством ошибок плохих данных, являются исполняемым кодом или прочими инструкциями, которые передаются программе. Если программу можно обмануть и заставить исполнить эти инструкции, последствия могут оказаться ужасными и включать в себя потерю или повреждение данных, неавторизованный доступ к системе или просто вывод системы из строя.

   Наиболее часто такую проблему могут вызвать управляющие и необычные символы. Управляющий код, который не улавливается интерфейсом ввода данных, может быть распознан другой подсистемой, и наоборот. Символы могут использоваться различными подсистемами и языками для различных целей. Например, в Java строки могут содержать нулевые символы (null), в то время как для строк языка C нулевой символ означает конец строки.

   Перед проведением тестирования определите, что произойдет, когда различные виды кода передаются программе. Это необязательно должна быть Java (или тот язык, на котором написана система). Веб-приложение может исполнять JavaScript-код в браузере посетителей. Многие системы используют SQL для

запросов к базе данных и ее обновлений, и произвольный SQL-код может изменить или удалить данные. Внедряемый код может быть помещен в конце длинной строки — хитрость, часто используемая для того, чтобы воспользоваться переполнением буфера (см. главу о тестировании безопасности). Проверка всех этих потенциальных опасностей потребует тестирования с широким диапазоном входных значений, и существуют противники, которые выиграют, если вы пропустите хотя бы одну точку входа для их злоумышленных программ.

10. **Сетевые ошибки.** Хотя все компьютеры все больше и больше взаимодействуют друг с другом через сети, не везде сетевые соединения доступны или функционируют без сбоев. Система должна продолжать работу, даже если сетевое соединение временно потеряно. Конечно, некоторые системы требуют наличия сетевого соединения (работа в ssh-клиенте или веб-браузере покажется довольно скучной, если вы не можете подключиться к другим системам), но определенно они не должны падать или зависать без него.

    Возможно, самым драматичным примером тестирования сетевых ошибок является "тест топора" (упомянутый выше), в котором тестировщик берет топор и обрубает кабель, соединяющий систему с сетью. Это можно сымитировать без риска травм путем отключения кабеля или отключения сигнала Wi-Fi в самый разгар работы программы.

    Потеря сетевого соединения является не единственной возможной проблемой, которая способна возникнуть у работающей в сети программы. Вы можете также проверить, что именно произойдет в случае возникновения больших задержек при прохождении сетевых пакетов или чрезвычайно низкой пропускной способности. Зачастую программа может предположить, что существующего соединения достаточно, и продолжить работу, но в итоге работа системы окажется нестабильной, т. к. качество имеющегося соединения является очень низким. Согласно другому сценарию качество соединения может оказаться изменяющимся (частые разрывы и соединения), что может вызвать проблемы, которые не видны в случае одной долгой потери связи с сетью. Для сетевого соединения с высоким уровнем потери пакетов можно создать свой тест, особенно если вы используете UDP или другой протокол без установления соединения. Для реального теста вы можете добавить искажения в линию. Во всех этих ситуациях и в случае других сетевых проблем качество работы системы должно ухудшаться постепенно, а не приводить к мгновенному ее отключению.

11. **Ошибки ввода-вывода.** Данные внутри вашей программы живут в уютном маленьком подготовленном мире. Данные вне вашей программы злы и неконтролируемы, с клыками, когтями и не соблюдают никаких законов кроме законов Мерфи. Если вы читаете файл с диска, возможно, что этот файл не существует. Возможно, он существует, но не в том формате. Возможно, он правильного формата, но поврежден. Возможно, вы пытаетесь записать в него, но он только для чтения. Возможно, другой пользователь уже открыл его. Возможно, он в каталоге, доступа в который у пользователя нет. Возможно, у вас был доступ в этот каталог в момент запуска программы, но не теперь. Возможно, файл существует, но он пустой. Возможно, размер файла в сотни мегабайт, в то вре-

мя как вы предполагали, что всего несколько килобайт. Этот перечисление можно продолжать и дальше.

Вам следует проверять, что, если вашей системе нужен доступ к диску, она подготовлена к подобным случайностям. Один из возможных путей — иметь специальный подкаталог для тестирования, заполненный самыми разными необычными файлами, и когда новой функции потребуется прочитать файл, запустить ее на всех этих экземплярах. Эти необычные файлы могут реализовывать упомянутые выше проблемные ситуации, а также те случаи, которые связаны со сферой работы программы (например, файлы с самореферентной внутренней структурой, содержащие отсутствующие данные и т. п.).

12. **Ошибки интерфейса.** Системам часто приходится взаимодействовать с другими системами, и для этого им необходим какой-то интерфейс. Этот интерфейс может быть более или менее определенным, от простого приема текста на входе и выдаче текста на выходе (как в большинстве утилит UNIX, таких как more или grep) до работы со сложными бинарными форматами. Однако этот интерфейс должен быть определен на каком-то уровне, и существует вероятность, что определение было дано неоднозначно, или различные части определения противоречат друг другу, или же разные участники команды разработчиков интерфейса допускали разные предположения при его создании (см. выше об ошибках предположений). Есть даже вероятность, что были допущены ошибки во время программирования интерфейса!

Эти интерфейсы не обязательно должны быть межпроцессными или межкомпьютерными. Это может быть такая низкоуровневая модель, как интерфейс между двумя классами. Интерфейс метода обычно легко понять; например, в Java легко разобраться, что именно следующий метод принимает в качестве аргументов и что выдает, даже без всяких комментариев:

```
public boolean greaterThanTen(int a, int b) {
    return (a + b) > 10;
}
```

В данном случае вы можете увидеть, что метод принимает два целых значения, а возвращает булево. Довольно легко понять, что именно делает метод, даже без разъяснений. Хотя всегда существует вероятность (намеренно или случайно) написать запутанный код, положиться на побочные эффекты и глобальные переменные, но мозг среднестатистического программиста неплох в понимании вещей на уровне методов. Правда, как только вы, поднимаетесь на уровень классов, становится сложнее сказать, как и почему вещи взаимодействуют между собой. Вы больше не смотрите на сам язык, фокусируясь на таких ключевых словах, как for, if и return, которые знакомы вам, как ваше детское одеяло. Вместо этого вы видите "метаязык" класса, который окажется новым для вас. Вы работаете на более высоком уровне абстракции, и все будет становиться еще хуже по мере вашего продвижения вверх. Решением станет правильное определение и дизайн интерфейсов, но то, что понятно одному человеку, другого заставит вскинуть руки в отчаянии. Интерфейсы, как правило, проводят демар-

кационную линию между разработчиками или командами разработчиков. Тем самым они создают плодородную почву для непонимания, а там, где есть непонимание, также имеются и дефекты.

Во время тестирования вам необходимо потратить время на исследование интерфейсов системы. Существуют области, где коммуникации могли быть нарушены или где люди по разным сторонам сделали различные предположения. Что происходит, когда вы попытаетесь передать неожиданные значения? Что произойдет, если не передать ожидаемое значение, или передано больше значений, чем ожидается? Что произойдет, если данные лежат вне диапазона или слишком большие?

13. **Ошибки нулевого указателя (Null pointer).** Возможно, вы счастливчик и работали с языком, в котором отсутствует концепция нулевого указателя, но если вы программируете на Java, то наверняка знакомы с ней даже лучше, чем вам кажется. Каждый раз, когда объект может быть нулевым, должна осуществляться явная проверка, что он таковым не является, перед тем, как вы попытаетесь к нему обратиться. В 2012 году в разговоре об истории концепции нулевых объектов Франклин Чен (Franklin Chen) заметил, что в Java-подобных языках у любого объекта всегда *два* возможных состояния — сам объект и нулевая версия его самого[1]. Вы не можете предположить, например, что если у вас есть объект `Integer`, то он на самом деле является `Integer`; это может быть нулевой объект, маскирующийся как целый.

    Это сравнительно легко заметить, если вы осуществляете тестирование белого ящика и видите сам код. Если же вы осуществляете тестирование черного ящика, вам придется подумать также о случаях, когда объект может не существовать. Что произойдет, если вы попытаетесь искать объект в базе, которая не существует? Что произойдет, если вы ничего не введете в текстовое поле? Что произойдет, когда вы зададите неправильный ID? Для многих программ поведение при подобных ситуациях должно быть заранее явно определено. Каждый раз, когда обычное поведение должно быть проверено на однозначность, существует вероятность, что программист забыл это сделать или сделал неправильно.

14. **Ошибки распределенных систем.** Тестирование системы, которая запускается одновременно на нескольких серверах, имеет свои особенности. Во многих случаях не существует "истинной" копии данных (единого места, где, как предполагается, данные всегда корректные, в отличие от мест, содержащих "копии", которые могут быть устаревшими или неправильными). Вам предстоит позаботиться не только о тестировании системы с различными программными настройками и оборудованием, но и проверить различные сетевые топологии и разные комбинации настроек ПО и оборудования. Различные уровни пропуск-

[1] Для углубленного анализа "проблемы на миллиард долларов" нулевых указателей ознакомьтесь со статьей Франклина Чена "Исторические, теоретические, сравнительные, философские и практические перспективы" (Franklin Chen "Historical, Theoretical, Comparative, Philosophical, and Practical Perspectives") по адресу: **https://franklinchen.com/blog/2012/09/06/my-pittsburgh-ruby-talk-nil/**.

ной способности канала и задержек прохождения пакетов приведут к тому, что дефекты проявят себя. Время и отметки времени могут отличаться от системы к системе, несколько человек могут редактировать одни данные с разных машин, разные машины могут иметь разные концепции текущего состояния данных и т. д. Существует множество ситуаций, в которых определение ожидаемого поведения может оказаться сложным, а то и вообще невозможным.

При тестировании распределенной системы вам нужно потратить время на проверку того, что системы синхронизированы правильно; данные должны быть согласованными (по крайней мере, в конечном итоге). Убедитесь, что система работает, когда изменяется одна из ее составляющих, и особенно когда выходит из строя одна из отдельных систем (а это почти наверняка когда-то произойдет — чем больше машин выполняют ваш код, тем более вероятно, что по крайней мере одна из них выйдет из строя). Прочитайте статью Питера Дейтча "Восемь ошибок распределенных вычислений" (Peter Deutsch "The Eight Fallacies of Distributed Computing"[1]) — не беспокойтесь, читается очень быстро — и подумайте о тех предположениях, которые сделали вы и разработчики системы, а затем... сломайте эти предположения.

15. **Ошибки конфигурации.** Ошибки конфигурации могут проявляться двумя разными способами. Первый — когда администраторы системы могут сконфигурировать систему по-разному. Например, при настройке приложения Rails можно задать многочисленные параметры в различных конфигурационных YAML-файлах. У многих приложений имеются настройки конфигурации, ключи командной строки или другие способы изменения работы, и иногда они переопределяют друг друга или же взаимодействуют между собой самыми невообразимыми способами.

    Перейдем ко второму виду ошибок конфигурации. У работающих с онлайн-системой пользователей компьютеры могут быть настроены по-разному. Например, у них могут быть установлены разные браузеры — от мощных, выпущенных крупными компаниями и организациями, до небольших текстовых. У этих браузеров свои уровни совместимости со стандартами. Пользователи будут запускать эти браузеры на различных операционных системах, с разным железом и программным обеспечением, с различными плагинами. У некоторых пользователей JavaScript включен, у других — нет; некоторые используют блокировщики рекламы; у кого-то отключена загрузка изображений; у некоторых в браузерах включена функция запрета отслеживания, у некоторых нет — это перечисление можно продолжать. Всегда существует вероятность того, что проблема кроется в особой конфигурации.

    Будет ли ваша система работать соответствующим образом при всех возможных конфигурациях, которые пользователи используют для доступа (т. е. разные браузеры, отключенный JavaScript, без картинок и т. д.)? Выдаст ли система соответствующую информацию об ошибке, если она (система) сконфигури-

[1] См. **https://blogs.oracle.com/jag/resource/Fallacies.html**.

рована неправильно? Имеются ли у нее разумные настройки по умолчанию (или предупреждает ли она пользователя в зависимости от требований и области применения) в случае, если какое-то значение в настройках отсутствует или неверное? Имеются ли какие-то настройки, которые перекрывают другие неочевидным образом или вызывают проблемы при определенных комбинациях?

16. **Ошибки доступности.** Часто системы будут работать правильно, когда пользователь использует стандартные системы ввода и вывода, но не когда пользователь попытается задействовать нестандартные устройства. Они зачастую необходимы для тех, кто не может работать с системой при помощи стандартного набора "клавиатура — мышь — монитор"; например, слепые пользователи, которые используют дисплей Брайля для чтения выходных данных с компьютера. Если ваша программа не работает корректно с такими системами, значит, зависящие от них пользователи не смогут пользоваться вашей программой.

    Убедитесь, что вы предоставляете несколько способов ввода и вывода или как минимум можете принимать и выводить обычный тест. Не все пользователи могут использовать мышь или видеть изображения. Не предполагайте, что ваши настройки подойдут для всех, кто использует программу.

7.2. Список продолжается и продолжается

Эта глава не является всеохватывающим перечнем того, как сломать программу и найти дефекты. Это даже не сравнительно полный список. Дело в том, что компьютеры выполняют в точности то, что вы говорите им сделать, и одна из сложностей написания программ — это разъяснение компьютеру, что делать при самых разных обстоятельствах. В то время как многие из этих обстоятельств являются сравнительно общими для программ (например, как обрабатывать ситуации с отсутствующими файлами или пропаданием сетевого соединения), многие другие ошибки будут специфичными для области, в которой вы работаете, или для программы, которую вы пишете. Как тестировщик, вы должны смотреть на вещи шире и постоянно думать о путях, при которых проявятся дефекты конкретной тестируемой программы.

И когда вы это делаете, помните, что вы не пользователь. Предполагается, что вы, как тестировщик программного обеспечения, должны быть знакомы с тестируемой системой — если не сразу, то со временем. Во многих случаях вы будете более технически подкованы, чем человек, пользующийся программой. Пользователи не будут работать с программой так, как это делаете вы, и то, что кажется очевидным вам, будет совсем не очевидным для них. Важно представлять виды ошибок, которые пользователи будут совершать, что именно они будут вводить в качестве входных данных и что конкретно ожидают увидеть.

Глава 8

Прохождение тестового плана

Давайте пройдемся по разработке тест-плана для заданного списка требований к системе по взвешиванию котов *catweigher* (если вы хотите использовать умные названия для программ, то в этой главе вы их не найдете). Эта чрезвычайно полезная программа будет принимать один аргумент в виде веса[1] кота в килограммах и показывать нам, какой вес у этого кота — недостаточный, нормальный или избыточный:

```
$ catweigher 1.7
Cat Weighing System
Cat is underweight
// (вес кота недостаточный)

$ catweigher 83
Cat Weighing System
Cat is overweight
// (вес кота избыточный)
```

По мере разработки тест-кейсов обратите внимание на компромиссы, которые были сделаны, и как принимались решения о том, какие тест-кейсы включать. Также обратите внимание, как используются идеи, которые мы изучали в предыдущих главах, особенно связанные с разделением классов эквивалентности и проработкой неуспешных кейсов, чтобы создать всесторонний тест-план.

8.1. Изучение требований

1. *FUN-PARAMETER.* Система должна принимать один параметр `CATWEIGHT`, который должен быть либо положительным значением с плавающей запятой, либо положительным целым числом. Если параметр не соответствует ни одному из этих двух типов или если параметр не один, система должна немедленно пре-

[1] Естественно, что в килограммах измеряется не вес кота, а масса кота (как и любого другого объекта). Однако в обиходе мы часто используем термин "вес", подразумевая массу. Далее понятие "вес" употребляется именно в этом, бытовом, плане. — *Прим. ред.*

кратить работу с единственным сообщением: "Пожалуйста, введите правильное значение".

2. *FUN-STARTUP-MESSAGE.* При запуске система должна вывести в консоли сообщение "Cat Weighing System" ("Система взвешивания кота").
3. *FUN-UNDERWEIGHT.* Если `CATWEIGHT` меньше 3 кг, тогда в консоли должно быть выведено сообщение "Cat is underweight" ("Вес кота недостаточный").
4. *FUN-NORMALWEIGHT.* Если `CATWEIGHT` равен или больше 3 и меньше 6 кг, тогда в консоли должно быть выведено сообщение "Cat is normal weight" ("Вес кота нормальный").
5. *FUN-OVERWEIGHT.* Если `CATWEIGHT` больше или равен 6 кг, тогда в консоли должно быть выведено сообщение "Cat is overweight" ("Вес кота избыточный").
6. *NF-PERF-TIME.* Система должна отображать соответствующее сообщение в течение 2 секунд во время исполнения программы.

Хотя это сравнительно простая программа с небольшим набором требований, здесь уже присутствует некая неоднозначность. Можно предполагать, что в реальных приложениях ее будет больше. Зачастую это станет причиной дискуссий с системными инженерами, ответственными за требования аналитиками и/или потребителями для устранения различных неоднозначностей. Конкретная неоднозначность в этом случае находится в разделе FUN-PARAMETER, где говорится, что система должна немедленно прекратить работу с *единственным* сообщением "Пожалуйста, введите правильное значение" в случае, если введено неверное значение. Следующее требование FUN-STARTUP-MESSAGE говорит, что сообщение "Cat Weighing System" должно демонстрироваться при запуске; в нем не упоминается, должны ли учитываться случаи, когда ранее вводились неверные значения. Другими словами, ожидаемое поведение должно быть таким:

```
$ catweigher мяу
Пожалуйста, введите правильное значение
```

или таким:

```
$ catweigher мяу
Cat Weighing System
Пожалуйста, введите правильное значение
```

Нам следует определить ожидаемое поведение перед началом написания тест-плана. Это может быть выполнено проверкой соответствующих требований аналитиками, системными инженерами, владельцами продукта (product owners) и всеми, кто отвечает за требования. Если вы работаете с менее формальной командой, правильным решением будет сделать предположение. Однако эти предположения должны быть отмечены в тест-плане! Если кто-то делает предположения, они должны быть по крайней мере где-то ясно описаны. Но в целом вы должны избегать предположений; вы должны знать ожидаемое поведение настолько точно, насколько это возможно. В этом случае давайте предположим, что мы отправились к менеджеру проекту (конечно, у *catweigher* есть менеджер проекта) и определили,

что сообщение "Cat Weighing System" не должно отображаться, если значение было некорректным. Другими словами, первый вариант вывода является правильным.

8.2. Разрабатывая тест-план

Я убежден, что нисходящий подход требований программ наиболее полезен при создании тест-плана. Под подходом "сверху вниз" я понимаю, что сперва создается общий план, а затем он заполняется конкретными деталями. Это контрастирует с подходом "снизу вверх", когда сперва описываются конкретные особенности небольших разделов, и чем больше и больше малых особенностей описывается, тем яснее становится общая картина.

Рассматривая требования, я могу мысленно разделить их на три секции:

1. Ввод (задание параметра):
   - FUN-PARAMETER.
2. Вывод (отображение сообщений и результатов):
   - FUN-STARTUP-MESSAGE;
   - FUN-UNDERWEIGHT;
   - FUN-NORMALWEIGHT;
   - FUN-OVERWEIGHT.
3. Выполнение:
   - NF-PERF-TIME.

Как я определил, как с этим разобраться? Я посмотрел на требования, которые были связаны друг с другом, и поместил каждое из них в "кластер". У программы, с которой вы работаете, скорее всего, не будет точно таких кластеров (ну если только вы не создаете программу по взвешиванию котов — в этом случае вам повезло). Для больших программ эти кластеры будут часто развиваться вокруг конкретных особенностей (например, для онлайнового магазина это будут корзина покупок, отображение товаров, оплата) или различных подсистем (например, для видеоигры это будут пользовательский интерфейс, искусственный интеллект врагов и графика). Не существует правильного ответа, как кластеризовать требования для тестирования, и улучшение понимания системы может привести к изменению кластеризации. Однако имея общий план, вы начнете разбираться с тем, как тестировать систему в целом.

Рассматривая тот набросок тест-плана, который мы здесь создали, мы понимаем, что у второй секции (вывод), безусловно, больше всего требований, и это необязательно означает, что ее тестирование займет больше всего времени или потребует больше всего работы. Фактически FUN-STARTUP-MESSAGE является очень простым и неизменяемым, а три других требования включают математически чистые функции, и написать тесты для них будет сравнительно легко. В повседневной жизни тестирование некоторых требований может занять намного больше времени, чем других! Тестирование требований производительности и безопасности мо-

жет оказаться значительно сложнее, чем может показаться по длине их описаний.

8.3. Заполняя тест-план

Давайте начнем с первой секции — ввода. Я замечу, что здесь есть несколько возможных случаев:

1. Пользователь ввел один допустимый (valid) параметр.
2. Пользователь не ввел параметров.
3. Пользователь ввел один параметр, но он недопустимый.
4. Пользователь ввел два параметра (на один больше).
5. Пользователь вводит гораздо больше одного параметра.

В последних четырех случаях ожидаемое поведение одинаково — система прекращает работу, а на экране выводится сообщение "Пожалуйста, введите правильное значение". В первом случае система продолжит работу и ее поведение будет определяться другими требованиями. Вы уже видите на данном этапе, что важно иметь понимание всей системы, а не только рассматривать требования исключительно сами по себе. Это становится всё более и более сложным по мере возрастания количества требований и усложнения тестируемой системы.

У случаев 1, 3, 4 и 5 имеются варианты, которые могут быть протестированы. Существует множество значений для допустимых параметров в первом случае, которые вызовут различное поведение, как перечислено в требованиях. Существует практически бесконечное значение одиночных недопустимых параметров — от отрицательных значений до мнимых чисел и случайных строк любой длины. В четвертом случае может быть два допустимых параметра, два недопустимых, или же один допустимый и один недопустимый. Для пятого случая единственным ограничителем того, сколько параметров может ввести пользователь, является операционная система; он или она может ввести два, три, четыре и более. Во втором случае нет вариантов; не существует различных способов не ввести параметр.

В зависимости от важности программы по взвешиванию котов мы можем решить, хотим ли мы тестировать все возможные случаи, и сколько именно вариантов конкретного случая. Вероятно, что как минимум вы захотите протестировать счастливый путь, т. е. тот ожидаемый путь пользователя, когда он вводит одно допустимое значение. Поскольку пользователи часто забывают формат аргументов, мы также должны проверять неверный ввод с правильным количеством аргументов. Мы, возможно, также захотим протестировать граничные значения в случае правильного количества аргументов — так будут чаще всего обнаруживаться сбои для этого требования. Пользователи могут забыть ввести аргумент или ввести дополнительный. Изредка пользователь может подумать, что требуется гораздо больше аргументов. Все эти случаи помимо счастливого пути могут рассматриваться как неуспешные, т. к. ожидаемое поведение заключается в прекращении работы после ввода недопустимого значения или нескольких значений:

ИДЕНТИФИКАТОР: VALID-PARAMETER-TEST
ТЕСТ-КЕЙС: Запустить программу с допустимым параметром
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: 5
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher 5"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа завершает работу и отображает правильное выходное значение для пятикилограммового кота. Программа не выводит сообщение "Пожалуйста, введите правильное значение"

Для требования FUN-PARAMETER имеются четыре неуспешных случая, для которых я бы хотел добавить тесты. Они находятся вне счастливого пути, т. к. указывают, что пользователь работает с программой некорректно. Однако в этих случаях нам все еще необходимо гарантировать, что система следует требованиям. Давайте добавим дополнительные тест-кейсы для трех неуспешных способов:

ИДЕНТИФИКАТОР: INVALID-PARAMETER-TEST
ТЕСТ-КЕЙС: Запустить программу с недопустимым параметром
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: "dog"
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher dog"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа выводит сообщение "Пожалуйста, введите правильное значение" и завершает работу, не выводя больше никакой информации

ИДЕНТИФИКАТОР: NO-PARAMETER-TEST
ТЕСТ-КЕЙС: Запустить программу без параметра
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: Нет
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа выводит сообщение "Пожалуйста, введите правильное значение" и завершает работу, не выводя больше никакой информации

ИДЕНТИФИКАТОР: TWO-PARAMETER-TEST
ТЕСТ-КЕЙС: Запустить программу с двумя параметрами
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: 1 2
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher 1 2"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа выводит сообщение "Пожалуйста, введите правильное значение" и завершает работу, не выводя больше никакой информации

ИДЕНТИФИКАТОР: TOO-MANY-PARAMETER-TEST
ТЕСТ-КЕЙС: Запустить программу с большим количеством параметров (в данном случае с четырьмя)
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: 5 6 7 8
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher 5 6 7 8"

```
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа выводит сообщение "Пожалуйста, введите правильное значение"
и завершает работу, не выводя больше никакой информации
```

На данный момент имеется разумное тестовое покрытие для этого требования. Переходим к следующему — FUN-STARTUP-MESSAGE, которое оказывается очень простым для тестирования. Здесь две возможности: системе передали допустимые параметры и не передали допустимые параметры. В первом случае должно отображаться стартовое сообщение, а во втором не должно (как мы определили ранее, разбираясь с неоднозначностью в требовании к сообщению):

```
ИДЕНТИФИКАТОР: STARTUP-NO-MESSAGE-TEST
ТЕСТ-КЕЙС: Запустить программу без параметра, стартовое сообщение не должно
отобразиться
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: Нет
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа не выводит сообщение "Cat Weighing System" на экран
перед завершением работы

ИДЕНТИФИКАТОР: STARTUP-MESSAGE-TEST
ТЕСТ-КЕЙС: Запустить программу с допустимым параметром, должно отобразиться стартовое
сообщение
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: 5
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher 5"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа выводит сообщение "Cat Weighing System" на экран
перед отображением статуса веса кота
```

Теперь у нас покрыты две возможности. Обратите внимание, что постусловия фокусируются на специфических аспектах вывода, которые нужно протестировать вместо того, чтобы проверять весь вывод. Например, STARTUP-MESSAGE-TEST проверяет только то, что сообщение отображается, а не информацию о статусе веса пятикилограммового кота. Хотя добавление каких-то деталей кажется пустяком — и, возможно, даже поможет отловить дефект или парочку, — это заведет нас в ловушку в случае более сложных программ. Представьте, что мы в будущем решим изменить определение термина "избыточный вес". Если вы считаете, что это надуманный вопрос, то с людьми это уже однажды случалось — по крайней мере, в Соединенных Штатах. В 1998 году десятки миллионов людей вдруг оказались с "избыточным весом" в связи с тем, что Национальные институты здравоохранения (National Institutes of Health) изменили определение термина[1]. А подобное может

[1] Для того чтобы узнать больше, ознакомьтесь со статьей на сайте CNN по адресу **http://www.cnn.com/HEALTH/9806/17/weight.guidelines/** или обратитесь к статье на сайте National Institutes of Health "Clinic Guidelines on the Identification, Evaluation, and Treatment of Overweight and Obesity in Adults" по адресу **https://www.ncbi.nlm.nih.gov/books/NBK2003/**.

произойти и с нашими бедными котами! И если произойдет, нам придется пройтись по всем тестам и убедиться, что даже сторонние тесты случайно не пострадали от изменения определения статуса веса. Сохраняя фокусировку наших тестов на конкретном ожидаемом поведении, мы гарантируем, что наш набор тестов не становится хрупким.

Легко добавить больше граничных случаев, например дополнительные допустимые и недопустимые входные значения, запуск на разных операционных системах, запуск, когда уже работают другие программы... Этот список можно продолжать. Но, как говорится, благоразумие — лучшая часть доблести. Трата слишком большого количества времени на подобное простое требование, вероятно, окажется неоптимальным использованием ресурсов. Конечно, всегда есть вероятность, что демонстрация сообщения является крайне важной задачей (например, если речь идет о правовой или связанной с безопасностью информации). В таких случаях очевидно, что данным задачам следует уделять больше внимания. Хотя может показаться, что я избегаю простых правил о том, сколько внимания уделять каждому требованию, я хочу донести мысль о том, что нет простых правил. Как тестировщику, вам придется решать, как фокусироваться на каждом требовании, подсистеме, функции или другом аспекте тестируемой системы. Этот зависит от области работы и конкретного программного продукта, и вы можете ошибиться. Я могу привести вам примеры и эвристику, но ни одна книга не заменит вам серое вещество между ушей.

8.4. Определяя фокус

Давайте перейдем к трем требованиям по весу: FUN-UNDERWEIGHT, FUN-NORMALWEIGHT и FUN-OVERWEIGHT. Сейчас самое время разделить входные и выходные значения на классы эквивалентности, как было объяснено раньше:

- < 3 кг $\rightarrow$ недостаточный вес;
- ≥ 3 кг и < 6 кг $\rightarrow$ нормальный вес;
- ≤ 6 кг $\rightarrow$ избыточный вес.

Давайте предположим, что котов взвешивают с инкрементом в одну десятую килограмма. Это можно уточнить, обсудив с системными инженерами и другими заинтересованными лицами. Мы можем выбрать явные граничные значения: `2.9`, `3.0`, `5.9` и `6.0` кг. Теперь давайте добавим внутреннее значение из каждого класса эквивалентности: `1.6` кг для недостаточного веса, `5.0` кг для нормального веса и `10` кг для избыточного веса. Мы также хотим добавить неявные граничные значения — скажем, `0` и `1000` кг. Последнее значение предполагает, что `1000` кг является теоретической верхней границей для кота перед тем, как он превратится в черную дыру в соответствии с моим пониманием физики (замечу, что автор не является физиком). В завершение давайте проверим некоторые угловые случаи: отрицательное число (`-13`), нечисловую строку (`quackadoodle_doo`). Посмотрите, как много внимания уделяется определению множества входных значений для этих требований по сравнению с начальным сообщением. Так как определение статуса веса кота является

сутью приложения, ему должно уделяться больше внимания с точки зрения тестирования:

```
ИДЕНТИФИКАТОР: UNDERWEIGHT-INTERNAL
ТЕСТ-КЕЙС: Запустить программу и передать в качестве параметра недостаточный вес кота,
составляющий 1.6 кг
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: 1.6
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher 1.6"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа выводит на экран сообщение "Вес кота недостаточный"

ИДЕНТИФИКАТОР: UNDERWEIGHT-LOWER-BOUNDARY
ТЕСТ-КЕЙС: Запустить программу и передать в качестве параметра вес невесомого кота,
составляющий 0 кг
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: 0
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher 0"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа выводит на экран сообщение "Вес кота недостаточный"

ИДЕНТИФИКАТОР: UNDERWEIGHT-UPPER-BOUNDARY
ТЕСТ-КЕЙС: Запустить программу и передать в качестве параметра недостаточный вес кота,
составляющий 2.9 кг
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: 2.9
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher 2.9"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа выводит на экран сообщение "Вес кота недостаточный"

ИДЕНТИФИКАТОР: NORMALWEIGHT-INTERNAL
ТЕСТ-КЕЙС: Запустить программу и передать в качестве параметра нормальный вес кота,
составляющий 5 кг
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: 5
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher 5"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа выводит на экран сообщение "Вес кота нормальный"

ИДЕНТИФИКАТОР: NORMALWEIGHT-LOWER-BOUNDARY
ТЕСТ-КЕЙС: Запустить программу и передать в качестве параметра нормальный вес кота,
составляющий 3 кг
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: 3
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher 3"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа выводит на экран сообщение "Вес кота нормальный"
```

ИДЕНТИФИКАТОР: NORMALWEIGHT-UPPER-BOUNDARY
ТЕСТ-КЕЙС: Запустить программу и передать в качестве параметра нормальный вес кота, составляющий 5.9 кг
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: 5.9
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher 5.9"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа выводит на экран сообщение "Вес кота нормальный"

ИДЕНТИФИКАТОР: OVERWEIGHT-INTERNAL
ТЕСТ-КЕЙС: Запустить программу и передать в качестве параметра избыточный вес кота, составляющий 10 кг
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: 10
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher 10"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа выводит на экран сообщение "Вес кота избыточный"

ИДЕНТИФИКАТОР: OVERWEIGHT-LOWER-BOUNDARY
ТЕСТ-КЕЙС: Запустить программу и передать в качестве параметра избыточный вес кота, составляющий 6 кг
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: 6
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher 6"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа выводит на экран сообщение "Вес кота избыточный"

ИДЕНТИФИКАТОР: OVERWEIGHT-UPPER-BOUNDARY
ТЕСТ-КЕЙС: Запустить программу и передать в качестве параметра избыточный вес кота, составляющий 1000 кг
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: 1000
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher 1000"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа выводит на экран сообщение "Вес кота избыточный"

ИДЕНТИФИКАТОР: WEIGHTSTATUS-INVALID-NEGATIVE
ТЕСТ-КЕЙС: Запустить программу и передать в качестве параметра отрицательный вес кота, составляющий -13 кг
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: -13
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher -13"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа не отображает информацию о статусе веса кота и прекращает работу

ИДЕНТИФИКАТОР: WEIGHTSTATUS-INVALID-STRING
ТЕСТ-КЕЙС: Запустить программу и передать в качестве параметра неправильный строковый аргумент

```
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: "quackadoodle_doo"
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher quackadoodle_doo"
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Нет в наличии
ПОСТУСЛОВИЯ: Программа не отображает информацию о статусе веса кота и прекращает
работу
```

Может показаться, что слишком много текста для этих тестов с их относительной важностью! Впрочем, запомните, что объем требуемой документации будет зависеть от вашей компании, от правовых требований к тестируемой системе и т. д. Хотя мы создали относительно проработанный тест-план, в отважном стартапе могут ограничиться простым списком различных проверяемых значений и неформально ожидаемых поведений программы. Ответственность в определении нужного поведения может оказаться на тестировщике. И хотя это потребует от него больше работы и умственных усилий, а также готовности к возможным ошибкам, компромисс будет состоять в том, что тестовый план окажется более гибким, а разработка его пройдет быстро. В тестировании программного обеспечения, как и во всей сфере разработки, очень редко встречаются "абсолютно правильные" ответы, и часто приходится выбирать правильные компромиссы.

Также имеется некоторое пересечение между двумя последними тест-кейсами (WEIGHTSTATUS-INVALID-NEGATIVE и WEIGHTSTATUS-INVALID-STRING) и предыдущими тест-кейсами по проверке требования FUN-PARAMETER. И хотя они рассматривают немного отличающиеся аспекты системы, в работе все они используют методологию черного ящика. Можно было бы сказать, что они не являются необходимыми, хотя тестирование белого ящика может показать, что различные части системы тестируются различными тест-кейсами.

В зависимости от культуры компании и того, что ожидается от документации тест-плана, мы могли бы сжать тест-кейсы таким образом, чтобы каждый из них работал со множеством значений. И хотя я не рекомендую такой подход — проще копировать и вставлять, а большее количество тест-кейсов позволяет им быть более конкретным — это экономит время при написании. Это также позволяет каждому тест-кейсу покрыть бóльшую "территорию", но добавляет дополнительный шаг для определения того, где кроется проблема, если тест-кейс завершился неуспешно. Опуская эту оговорку, давайте рассмотрим пример сжатия трех тест-кейсов о недостаточном весе в один:

```
ИДЕНТИФИКАТОР: UNDERWEIGHT-INTERNAL
ТЕСТ-КЕЙС: Запустить программу и передать ей в качестве параметра значения
недостаточного веса кота
ПРЕДУСЛОВИЯ: Нет
ВХОДНЫЕ ЗНАЧЕНИЯ: 0, 1.6, 2.9
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher n", где n равняется
одному из входных значений
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Не доступны
ПОСТУСЛОВИЯ: Для каждого из входных значений программа выводит на экран сообщение
"Вес кота недостаточный"
```

8.5. Тест-кейсы для нефункционального требования

Впереди будет целая глава о тестировании производительности, но пока давайте пройдемся через один очень простой тест производительности. Мы хотим убедиться, что система выполняет расчет и выводит информацию на дисплей за 2 секунды. Легкий способ проверить это — использовать утилиту `time` операционной системы UNIX, которая покажет, сколько времени заняло выполнение команды. Хотя этот стандартный инструмент выдаст вам несколько различных результатов, сосредоточьтесь только на "реальном" (отмечен как `real`), который измеряет, сколько времени заняло выполнение чего-либо в соответствии с часами на стене (другие виды времени, которое измеряет команда `time`, будут обсуждаться в главе о тестировании производительности).

Мы хотим проверить, что будут рассчитаны различные значения и программа завершит работу в течение 2 секунд. В случае, если вычисления для одного класса эквивалентности занимают гораздо больше времени, чем другие, тестируется несколько значений:

```
ИДЕНТИФИКАТОР: PERFORMANCE-RUNTIME
ТЕСТ-КЕЙС: Запустить программу с различными входными значениями, измерить время
выполнения, убедиться, что каждая итерация занимает меньше двух секунд.
ПРЕДУСЛОВИЯ: У программы нет известных функциональных дефектов
ВХОДНЫЕ ЗНАЧЕНИЯ: 0, 1.5, 5, 7.5, 10
ШАГИ ВЫПОЛНЕНИЯ: В командной строке запустите "catweigher n", где n равняется одному
из входных значений
ВЫХОДНЫЕ ЗНАЧЕНИЯ: Не доступны
ПОСТУСЛОВИЯ: Для каждого из входных значений реальное время выполнения, измеренное
командой time, будет меньше 2.000 секунд.
```

Обратите внимание на появление предусловия. Этот тест-кейс не будет рассматриваться допустимым, пока не устранены все функциональные дефекты. В конце концов, неважно, сколько времени выполняется программа, если она не дает правильную информацию о весе кота! Это не означает, что тест-кейс нельзя выполнить заранее, но результаты его выполнения не могут рассматриваться валидными до тех пор, пока программное обеспечение не будет функционально правильным. Это одна из больших проблем тестирования производительности; часто им сложно заниматься до того, как программа станет функционально полной или по крайней мере будет близка к этому состоянию.

Глава 9

Дефекты

До сего момента мы потратили немало времени, обучаясь нахождению дефектов. Это разумно, т. к. является одной из ключевых целей тестирования. Однако мы еще не определили полностью, что является дефектом, и не обсудили, что с ними делать в случае обнаружения.

9.1. Что такое дефект?

Дефект — это любая ошибка в программе, которая становится причиной того, что тестируемая система выполняет одно из следующих условий:

1. Не соответствует заданным требованиям (функциональным или нефункциональным).
2. Возвращает неправильный результат.
3. Неожиданно прекращает работу (устойчивость системы является неявным требованием всех тестируемых систем).

Самый очевидный вид дефекта — система не соответствует требованиям. Если требования определяют, что программа должна делать что-то, а она это не делает, то это дефект. Хотя, как обсуждалось в посвященной требованиям главе, все написанное на естественном языке обязательно имеет некоторую неоднозначность. Разработчик, реализовавший новую функцию программы, мог иначе понимать требования, нежели тестировщик.

Примером системы, возвращающей неправильный результат, могла бы стать электронная таблица, показывающая, что 2 + 3 равняется 23, или графический редактор, в котором рисование начнется синим цветом после того, как пользователь выбрал красный цвет. Это может быть не определено требованиями (и, таким образом, может и не пересекаться с первым видом дефекта из списка выше).

Последний вид дефекта является общим для всех программ — программа не должна неожиданно прекращать выполнение (вылетать). То есть если целью пользователя, создателя или установщика программы является то, что система должна работать в определенный момент времени, то эта программа должна работать в этот момент времени.

Программное обеспечение может неожиданно прекратить работу, и это необязательно будет дефектом. Например, отправка сигнала SIGKILL (посредством команды `kill -9` или подобной) процессу UNIX вызовет прекращение работы программы без запуска ее подпрограмм, отвечающих за завершение работы. Однако она вылетела не неожиданно — пользователь хотел этого и даже отправил сообщение, говорящее, что надо выполнить именно это! Причиной того, что пользователю пришлось отправить сигнал SIGKILL процессу, мог быть дефект, но то, что программа прекратила работу в таких условиях, дефектом не является. Если система вылетает из-за ошибки сегментации, неотловленного деления на ноль или разыменования нулевого указателя — все это рассматривается как дефекты. Они никогда не должны появляться в программе, даже если в требованиях не определяется, что "программа должна выполняться без каких-либо исключений нулевого указателя".

В конце отметим, что слово **"баг"** часто используется как взаимозаменяемое по отношению к слову "дефект". Они означают одно и то же, но "баг" является более разговорным. В этой книге будет использоваться слово "дефект", за исключением тех случаев, где автор забыл об этом.

9.2. Жизненный цикл дефекта

После обнаружения дефекта тестировщик (или тот, кто нашел дефект) должен **доложить** об этом. Доклад может иметь различные значения в зависимости от организации и серьезности дефекта. По сути, это означает, что особенности дефекта должны быть отмечены там, где он будет рассматриваться в будущем заинтересованными лицами проекта — другими тестировщиками, разработчиками, менеджментом и т. д. В большинстве случаев у команд имеются программы по отслеживанию дефектов, но доклад о дефекте может быть простым, как пометка ошибки при помощи старомодной ручки и бумаги.

При тестировании программного обеспечения по множеству причин важно отслеживать всю информацию, которая практически необходима. Во-первых, чем больше информации доступно, тем более вероятно, что этот дефект удастся воспроизвести. Представьте, что у вас отказывает система только при установке определенных переменных среды, но вы не уточняете их в докладе о дефекте. Когда разработчик, которому поручено заниматься исправлением дефекта, начнет работу, он может пометить его комментарием "у меня всё работает", поскольку система работала, как и задумано, вне зависимости от того, что делал разработчик. Без знания этих переменных среды воспроизвести дефект не удастся.

С другой стороны, можно зайти слишком далеко. Обычно нет необходимости описывать каждый запущенный в системе процесс в случае обнаружения опечатки. Степень подробности описания дефекта будет зависеть от самого дефекта и области, в которой вы работаете. Однако существуют определенные данные, которые окажутся полезными во многих случаях. В следующем разделе будет описан шаблон дефекта для того, чтобы эти данные не забывались. Напоминая регистрирующему дефект о том, что именно нужно включить в сообщение, позволит ему не пропустить важные шаги или информацию и тем самым минимизировать чрезмер-

ную коммуникацию по данному дефекту в процессе его исправления. Чек-листы (от англ. *checklist* — котрольный список) и подобные шаблоны являются очень важными мощными инструментами в ситуациях, где цена ошибки в работе крайне высока.

Когда получен доклад о дефекте, его жизнь только начинается. Подобно тому как в разработке программного обеспечения существует "жизненный цикл ПО" — начинающийся с требований, дизайна и т. д., продолжающийся эксплуатацией и поддержкой и заканчивающийся "концом жизни" программы, — дефекты тоже имеют свой жизненный цикл. Он выглядит следующим образом:

1. Открытие.
2. Доклад.
3. Сортировка/назначение.
4. Исправление.
5. Верификация.

Когда тестировщик или другой пользователь впервые сталкивается с дефектом и распознает его, это стадия "Открытие". В некоторых случаях после этого может ничего не произойти — пользователь проигнорирует дефект или решит, что он не стоит дальнейшего изучения. Однако для профессионального тестировщика такой путь неприемлем — работа тестировщика заключается в определении качества программного обеспечения, и это включает в себя обнаружение дефектов и сообщение о них.

Вторая стадия — заполнение информации о дефекте, как правило, стандартизованным способом. Для этого нужно потратить некоторое время и выяснить, что именно сделал тестировщик, чтобы выявить проблему, что должно было произойти и что именно ожидалось. Помните, что роль тестировщика — в нахождении проблем и их воспроизведении, а не в выяснении, какой именно участок кода стал причиной. Этим тоже можно заняться — и часто приходится, если у тестировщика имеются необходимые технические знания, — но это не является его основной задачей. Если тестировщик погружается слишком глубоко в код, он или она испытает соблазн написать, что нужно для исправления дефекта, вместо того, чтобы сфокусироваться на том, что собой представляет дефект. Помните, как и в требованиях, дефект — это *что именно* неправильно, а не то, *как* это исправить.

После обнаружения дефекта кто-то должен принять решение, тратить ли ресурсы на его исправление, и если сто́ит это делать, то как расставить приоритеты в случае, когда дефектов окажется слишком много. Это называется **сортировкой**. В английском языке это слово (triage) происходит от медицинского термина, определяющего приоритет в обслуживании пациентов в зависимости от степени их ранения или заболевания. Так часто делается, когда единовременно появляется слишком большое количество жертв, чтобы медицинский персонал позаботился о них. Например, после сильного стихийного бедствия больница может быть переполнена пациентами: у некоторых из них легкие травмы, у других — серьезные, а у третьих — настолько тяжелые ранения, что пострадавших невозможно спасти. В таких случаях

госпиталь может начать сортировать поступающих пациентов и немедленно обслуживать пострадавших с серьезными, но излечимыми ранениями, и устанавливать более низкий приоритет для тех, у кого небольшие травмы, и тех, кого навряд ли спасут, даже если приложат все усилия.

Хотя подобная драматичная сортировка редка в медицинском мире, в сфере разработки программного обеспечения она встречается необычайно часто. Как правило, оказывается, что найденных дефектов гораздо больше, чем времени у разработчиков на их исправление. Соответствующие заинтересованные лица решают, какие дефекты должны быть исправлены с учетом имеющихся ресурсов, и устанавливают приоритеты в зависимости от того, как быстро эти дефекты могут быть исправлены и сколько боли они доставляют пользователям программы. В некоторых организациях ответственность за выбор дефектов, которыми нужно заниматься в первую очередь, лежит на разработчиках — именно они решают, какие дефекты наиболее легки для исправления, и снимают основную боль у пользователей программы. Такой тип работы хорош тогда, когда разработчики глубоко понимают нужды пользователя, например когда они работают над инструментом для разработки программного обеспечения и сами являются его пользователями.

После того как дефект передан разработчику или группе разработчиков, этот разработчик исправляет его. Зачастую после этого добавляется автоматизированный тест, чтобы покрыть ту ситуацию, при которой произошел дефект, и избежать повторения подобного в будущем. И хотя разработчик непосредственно близок к создаваемому им программному обеспечению, окончательное решение о том, был ли дефект исправлен, принимается не им. Подобно тому как некоторые люди не могут увидеть ошибки в написанных ими текстах, разработчик может не заметить совершенно другую ошибку, которая появилась из-за исправления, или то, что некоторые граничные случаи не были покрыты. Обязательный анализ кода (code review) поможет несколько улучшить ситуацию, но обычно окончательное решение по поводу того, что дефект исправлен, принимает тестировщик. Так как одним из ключевых аспектов работы тестировщика является независимый обзор качества ПО, он может предоставить более объективное и беспристрастное определение того, был ли исправлен дефект надлежащим образом или нет.

Таким образом, после исправления дефекта программу необходимо вернуть команде тестирования для проверки данного исправления. Разработчик или разработчики могли не протестировать все граничные случаи или же могли создать другие проблемы своим исправлением. Тестировщик может независимо проверить, что исправление действительно устранило дефект и не стало причиной возникновения других дефектов. Конечно, в некоторых случаях могут появиться другие дефекты, особенно если первый блокировал появление последующих. Например, если имеется опечатка в приветственной странице, которая появляется после авторизации, а дефект не позволяет пользователям авторизоваться, тестировщик все равно должен проверить исправление, позволяющее пользователям пройти авторизацию. То, что опечатка на приветственной странице была обнаружена после возможности залогиниться, не связывает сущности этих дефектов. Даже дефект, который вызывает связанные дефекты, иногда может быть верифицирован. В продолжение нашего

примера с неработающей формой входа: если исправление позволит пользователям авторизоваться, но при этом не будет проверяться пароль, ситуация может рассматриваться как улучшение, и регистрируется второй дефект по проблеме с паролем.

9.3. Стандартизованный шаблон дефекта

Это не отраслевой стандарт шаблона сообщения о дефекте, но я нашел его очень полезным. Он гарантирует, что были отмечены все основные аспекты обнаруженного дефекта. Выполняя роль своего рода чек-листа того, что должно быть записано о дефекте, он помогает гарантировать, что тестировщик ничего не забудет. Обратите внимание, хотя здесь не приведен никакой идентификатор, он часто добавляется автоматически той программой по работе с отслеживанием дефектов, которую вы используете. Если вы не применяете никакой программы по отслеживанию дефектов и не хотите использовать, можете добавить поле ИДЕНТИФИКАТОР.

Итак, вот шаблон:

```
КРАТКОЕ ОПИСАНИЕ:
ОПИСАНИЕ:
ШАГИ ВОСПРОИЗВЕДЕНИЯ:
ОЖИДАЕМОЕ ПОВЕДЕНИЕ:
НАБЛЮДАЕМОЕ ПОВЕДЕНИЕ:
ВЛИЯНИЕ:
СЕРЬЕЗНОСТЬ:
РЕШЕНИЕ:
ЗАМЕТКИ:
```

9.3.1. Краткое описание

Краткое описание — это одно предложение или чуть больше, кратко говорящее о найденном дефекте. Оно полезно для людей, которые просматривают длинные списки дефектов, или для того, чтобы убедиться, что эти люди поняли суть дефекта. Можно привести несколько хороших примеров кратких описаний:

1. Фон страницы красный, а должен быть синий.
2. Калькулятор приложения показывает –1 при расчете квадратного корня из –1.
3. Система завершает работу с ошибкой SEGFAULT после выбора пользователя.
4. Добавление трех товаров или более в корзину единовременно удаляет из корзины другие товары.

Все эти примеры описывают проблему сжато, но точно, с одной необходимой подробностью, показывающей проблему. В этом разделе нет чрезмерных комментариев. Помните, что на краткое описание дефекта среди сотен других кратких описаний зачастую лишь бросят взгляд, особенно если вы представляете их руководству. По этой причине большинство современных книг имеют краткие названия, в отличие от таких, как "Яйцо, или Мемуары Грегори Джидди, эсквайра: с литературными

произведениями господ Фрэнсиса Флимси, Фредерика Флорида и Бена Бомбаста, к которым добавляются личные мнения Пэтти Пут, Люси Люсиус и Присциллы Позитив, а также мемуары достопочтенного Щенка, задуманные известной Курицей и представленные публике знаменитой Кормушкой" ("The Egg, Or The Memoirs Of Gregory Giddy, Esq: With The Lucubrations Of Messrs. Francis Flimsy, Frederick Florid, And Ben Bombast. To Which Are Added, The Private Opinions Of Patty Pout, Lucy Luscious, And Priscilla Positive. Also The Memoirs Of A Right Honourable Puppy. Conceived By A Celebrated Hen, And Laid Before The Public By A Famous Cock-Feeder"). Да, это настоящая книга, опубликованная анонимным автором в 1772 году.

Вам следует быть очень осторожным, чтобы не сделать этот раздел слишком общим. Если пользователь не может найти одну конкретную книгу, не надо делать краткое описание "Поиск сломался". Если одно вычисление дает достоверно неправильный результат, не надо описывать дефект как "Расчеты не работают". Перед регистрацией дефекта попробуйте разобраться в том, как далеко простирается проблема.

9.3.2. Описание

В разделе **"Описание"** дефект описывается более подробно. Его цель — позволить глубоко понять проблему и выяснить причины, почему, собственно, такое поведение является дефектом. Обычно это несколько предложений, хотя если дефект является сложным или требует разъяснительной информации для понимания, оно может быть длиннее.

В этом разделе вы также можете больше углубиться в детали о границах проблемы, чем в "Кратком описании". Например, вы могли отметить в "Кратком описании", что "налоги не рассчитываются правильно для товаров стоимостью более 100 долларов". В "Описании" можно рассказать, что вы тестировали товары стоимостью $100,01; $200,00 и $1000,00, и в каждом случае налоги оказывались меньше половины, чем следовало бы. Это позволит другим узнать, что вы пробовали, и лучше понимать границы дефекта.

9.3.3. Шаги воспроизведения

Помните о том, что найденный вами дефект, возможно, кто-то захочет исправить. Это можете быть вы или кто-то другой, но первым шагом для исправления чего-то является способность воспроизвести это. **Шаги воспроизведения** являются конкретными шагами, которые надо выполнить для того, чтобы проявился дефект.

Когда кто-то передает разработчикам проблему, которую необходимо устранить, порядок действий должен быть следующим:

1. Разработчик пытается повторить дефект.
2. Если разработчику не удается повторить дефект, он делает пометку на докладе о дефекте "на моей машине работает" и прекращает заниматься дефектом.
3. В противном случае разработчик пишет код, который, как он надеется, приведет к исправлению дефекта.

4. Разработчик снова выполняет шаги воспроизведения, которые вызвали дефект.
5. Тестировщик отмечает в докладе, был ли исправлен дефект. Если нет, осуществляется возврат к пункту 3.

Вы могли заметить, что в пункте 2 был своего рода преждевременный выход для ситуации, когда разработчик не может воспроизвести дефект. Вы также могли заметить первое появление проклятия многих тестировщиков — отговорки, что "на моей машине работает"! Компьютеры разработчиков и пользователей сильно различаются установленными программами, библиотеками, возможно, даже операционными системами. Дефект, проявляющийся в продуктивной[1] или в тестовой среде, но не в системе разработчика, является обычной и очень распространенной проблемой. Так же часто, впрочем, оказывается, что тестировщик просто не конкретизировал шаги воспроизведения настолько точно, насколько это возможно. Как тестировщик, вы должны минимизировать проблему ответов "на моей машине работает!" путем написания шагов воспроизведения предельно детализированными и однозначными.

Это включает описание всех соответствующих шагов и предусловий перед теми шагами, которые напрямую вызвали появление дефекта, например, если необходимо предварительно установить некие переменные среды для запуска программы, или был передан определенный флаг сборки мусора, или даже что система должна быть запущена за 6 часов перед выполнением конкретных шагов тестирования. Определение того, что является соответствующим, а что нет — сложный процесс. По мере накопления опыта тестирования проектов в вашей области и большем понимании тестируемой системы, вы сможете использовать ваше тестировочное "шестое чувство" для решения этого вопроса. И перед тем как вы достигнете этой точки, обычно имеет смысл ошибиться в пользу "слишком много информации", чем предоставить ее недостаточно.

Обязательно записывайте все конкретные шаги и значения, особенно, если существует несколько путей выполнения. Никогда не пишите, что кому-то нужно "Ввести неправильное значение", вместо этого укажите "Введите –6 после появления знака >". Обратите внимание, что в последнем примере тестировщик предоставляет конкретный пример неправильного значения, а также говорит, где и когда его нужно ввести. Как и при написании тест-кейсов, вам необходимо представить, что человек, выполняющий их, является автоматом с общим пониманием системы. Скажите этому автомату точно, что именно надо делать, и устраните неоднозначность насколько это возможно.

9.3.4. Ожидаемое поведение

Ключевой концепцией тестирования, и на нее мы будем ссылаться снова и снова в этой книге, является проверка того, что ожидаемое поведение системы соответствует наблюдаемому поведению. Не имеет смысла тестировать нечто, если вы не

[1] То есть у пользователя. — *Прим. пер.*

знаете, что должно происходить. В поле **ожидаемого поведения** тестировщик отмечает, какой должна быть система после прохождения шагов выполнения.

И снова — чем более точно описано поведение, тем легче будет разработчикам и другим тестировщикам воспроизвести проблему и в итоге исправить ее. Никогда не надо писать "Система должна возвращать правильное значение", вместо это напишите, что "Система должна возвращать 6".

Помимо спасения людей от необходимости самостоятельно рассчитывать правильные значения, другим преимуществом точного описания ожидаемого поведения является то, что тестировщик или тест-кейс могут не ожидать правильного поведения! В этом случае кто-то может посчитать, что имеется дефект, и определить, в чем его ошибочность — хотя программа работает, как задумано. Такое может произойти, когда требования неоднозначные или сообщивший о дефекте тестировщик просто совершил ошибку.

9.3.5. Наблюдаемое поведение

Противоположным полю ожидаемого поведения является поле **наблюдаемого поведения**. В нем описывается то, что в действительности произошло после прохождения шагов выполнения. Как и все остальные поля шаблона сообщения о дефекте, оно должно быть заполнено настолько точно, насколько возможно. Если необходимо, можно добавить какие-то пояснения, объясняющие, почему наблюдаемое поведение отличается от ожидаемого (например, если это длинная строка, у которой изменилась всего одна буква в середине).

9.3.6. Влияние

Тестировщику необходимо понимать, какое влияние дефект окажет на пользователей. Это должно быть разъяснено в поле **"Влияние"**.

Будьте осторожны, чтобы не впасть в разглагольствования и не сделать этот раздел чересчур общим. Скажем, фон в игре во время прыжка игрока меняется с голубого на фиолетовый. Фактически описание влияния должно быть таким: "Пользователь увидит изменение цвета фона, но на игровой процесс это не повлияет". Версия с разглагольствованиями может выглядеть так: "Пользователь возненавидит эту игру и всех, кто ее делал, потому что тупой фон меняет тупой цвет, когда тупой пользователь делает что-то тупое". В общем, если вы обнаружите использование слова "тупой" или других уничижительных слов в сообщениях о дефектах, возможно, существует лучший способ описать проблему.

Также имейте в виду, что по возможности нужно описывать влияние на всех пользователей, а не на каких-то конкретных. Например, если проблема проявляется на всех браузерах, это окажется более серьезным, чем та же проблема, которая оказывает влияние только на пользователей какого-то одного браузера. Если дефект проявляется лишь у опытных пользователей, редактирующих файлы конфигурации системы, то это, вероятно, менее важно, чем если бы подобное происходило с новыми пользователями системы. Опытные пользователи скорее поймут, что пробле-

ма возникла после модификации ими файлов, чем новые. Указание, как дефект влияет на пользователей, поможет эффективной сортировке дефектов.

9.3.7. Серьезность

Связанным с полем "Влияние" является поле **"Серьезность"**, которое содержит начальное впечатление сообщающего о дефекте о том, насколько серьезна проблема. Это может быть описано либо в качественном отношении (т. е. "Это реально, реально плохо. Реально, реально, реально плохо" или "Если честно, это совсем не проблема"), либо при помощи стандартизованной шкалы количественно. Последнее гораздо популярнее, но обычно у каждой организации имеется свой подход. Это может быть нечто совсем простое вроде цифровой шкалы, где "1" означает очень незначительную мелочь, а "10" оказывается непреодолимой проблемой. Однако часто встречаются градации с дескрипторами.

Пример подобной системы оценок приведен ниже.

1. **Блокер.** Дефект настолько серьезный, что система не может быть выпущена без его исправления или разработки обходного пути. Примерами дефектов-блокеров могут стать ситуации, когда система не позволяет пользователю авторизоваться или же прекращает работу, когда кто-то нажимает клавишу <A>.
2. **Критический.** Хотя система может быть выпущена с дефектом подобной величины, он оказывает серьезное влияние на основную функциональность программы или делает ее практически непригодной к использованию. Другими словами, дефект, который мог бы быть помечен как блокер, но для которого существует какое-то решение, может быть классифицирован как критический.
3. **Значительный.** Это дефект, который вызывает относительно серьезные проблемы, хотя и не настолько серьезные, чтобы полностью остановить систему. Весьма вероятно, что пользователь заметит дефект, работая с той частью программы, где этот дефект присутствует.
4. **Обычный.** Это дефект, который доставляет неудобства пользователю, или же более серьезный дефект с простым и понятным решением.
5. **Незначительный.** Этот дефект может быть замечен, а может нет, и при этом не вызывает больших проблем для пользователя, или же для него существует простое и понятное решение.
6. **Тривиальный.** Этот дефект, вероятно, даже не заметят до тех пор, пока его не начнут искать специально. Примером может быть опечатка в большом блоке текста.
7. **Улучшение.** Иногда этот дефект даже и не является дефектом. Это нечто, что хочет пользователь, но при этом не определенное требованиями, или то, что не вписывается в рамки текущего проекта. В этом случае дефект может быть оформлен как улучшение. Другой вариант — пользователь может подумать, что проблема является дефектом, но менеджмент отклонит это предложение и отметит его как улучшение.

Оценка серьезности дефекта зачастую является скорее искусством, чем наукой, хотя существует некоторая эвристика для определения того, насколько серьезна проблема. В зависимости от области работы тестировщика ПО, а также внутренних правил тестирования в его организации, серьезность дефекта может кардинально отличаться. Даже в рамках одной организации могут быть конфликты между инженерами, менеджерами, тестировщиками и прочими заинтересованными лицами по поводу того, насколько серьезен дефект.

Серьезность не должна быть принята за приоритет. Серьезность говорит о том, насколько серьезна проблема, в то время как приоритет определяет порядок, в котором организация хотела бы видеть исправление проблемы. Например, у опечатки на главной странице веб-приложения низкая серьезность, т. к. она не оказывает влияния на функциональность программы. Тем не менее это создает плохое впечатление о создавшей приложение компании, и для исправления потребуется совсем немного времени разработчика. В этом случае приоритет может быть очень высоким. С другой стороны, у дефекта может быть низкий приоритет, но высокая серьезность. В качестве примера можно привести проблему с производительностью чтения информации из базы данных, которая оказывается настолько низкой, что делает систему практически непригодной к использованию после получаса работы. Это очень серьезная проблема. Однако на ее исправление уйдет много времени, а оптимизация базы данных уже запланирована по ходу проекта, и поэтому у проблемы будет низкий приоритет. Но при прочих равных условиях обычно дефекты высокой серьезности имеют более высокий приоритет, чем дефекты более низкой серьезности.

9.3.8. Решение

В поле **"Решение"** описывается, как избежать дефекта или по крайней мере смягчить его. Предположим, что дефектом является ситуация, когда в пароле не получается использовать специальные символы. Тогда решением станет выбор только алфавитно-цифровых. Важно заметить, что решениями не всегда являются *хорошие* решения; в них может не задействоваться определенный функционал. Например, если в текстовом редакторе не работает функция подсчета слов, решением может стать отказ от использования подсчета или применение другой программы (например, `wc -w` в UNIX). Это может вызвать дискомфорт и недовольство у пользователя, но позволит получить доступ к функциональности.

В некоторых случаях решение может не существовать или по крайней мере его нельзя найти. Если система прекращает работу в случайные моменты времени, то невозможно найти решение за исключением тривиального "не используйте программу". Обычно это не принимается в качестве решения. Это не означает, что решения не существует; можно лишь предполагать, что оно неизвестно; причиной проблемы может быть какое-то значение в настройках или во входных данных, но команда тестирования пока не нашла его. Если пользователь не может авторизоваться в веб-приложении, или сервер не запускается, или система постоянно выдает неправильные результаты на каждый запрос, тогда, возможно, не существует решения, т. к. система полностью непригодна к использованию. В таких случаях очевидно, что серьезность дефекта является блокером или его эквивалентом.

9.3.9. Заметки

Мне нравится думать об этом поле как о "прочем". В нем записывается всё, что может быть полезно для отслеживания дефекта, и то, что может или не может быть релевантным проблеме. Также здесь удобно размещать данные, являющиеся слишком длинными или не подходящими другим разделам, которые должны быть сравнительно короткими и простыми для понимания разработчиками, менеджерами, другими тестировщиками и всеми, кто может заниматься этим дефектом и пытаться понять его, но при этом не настолько знаком с этой частью программы, как работавший в ней тестировщик.

Что именно точно здесь будет записано, зависит от вида программы, которую вы тестируете, но обычно это технические детали программы, или ее среда, или же факторы, которые могут быть или могут не быть уместными. Можно привести примеры:

1. Трассировка стека.
2. Копии соответствующих частей лог-файла программы.
3. Переменные системы или окружения.
4. Технические спецификации системы, в которой был обнаружен дефект (т. е. операционная система, процессор, объем памяти и т. д.).
5. Другие приложения, запущенные на компьютере в то же время.
6. Особые настройки, флаги или аргументы, переданные программе.
7. Подозрительное или вызвавшее внимание поведение других программ.
8. Скопированное сообщение об ошибке.

9.4. Исключения в шаблоне

Для некоторых дефектов поля могут быть намеренно оставлены пустыми. Например, если дефект сравнительно простой и для него нет какой-то дополнительной информации, то в поле "Заметки" можно проставить "Нет". Во многих случаях, когда ожидаемое поведение очевидно, в соответствующем поле также можно проставить "Нет"; если наблюдаемое поведение "система вылетает", то для большинства обозревателей очевидно, что ожидаемое поведение будет "система не вылетает". Лучше написать "Нет", "N/A" или какой-то другой знак, благодаря чему у изучающих сообщение о дефекте не возникнет впечатления, что тестировщик просто забыл заполнить какие-то поля.

В других случаях в поле может быть проставлено "Неизвестно" или подобный маркер, показывающий, что автор сообщения о дефекте не знает, что должно быть в этом поле. Например, если система прекращает работу случайным образом, без сообщений об ошибке или каких-то других выходных данных, и тестировщик не способен воспроизвести ситуацию, в поле "Шаги воспроизведения" может быть проставлено "Неизвестно". Если требования неоднозначные или противоречивые, ожидаемое поведение может быть отмечено "Неизвестно". Если тестировщик не

уверен, какое влияние дефект окажет на конечного пользователя, т. е. потому что проблема глубоко в инфраструктуре системы, в поле "Влияние" также можно проставить "Неизвестно".

В зависимости от того, сколько внимания уделяется тестированию в организации, одни поля можно убрать, а другие добавить. Например, в некоторых организациях у тестировщиков могут отсутствовать знания в предметной области, которые помогут определить влияние или серьезность дефекта. Как минимум любое сообщение о дефекте должно включать шаги воспроизведения, ожидаемое поведение и наблюдаемое поведение. Все остальное является глазурью на торте дефекта.

9.5. Примеры дефектов

В этом разделе будет приведено несколько примеров дефектов.

Обратите внимание, что сообщение о дефекте представляет описание дефекта, а не то, как он может быть исправлен. В первом примере существует множество путей, позволяющих решить проблему дефекта — сообщение по умолчанию для экрана ошибки "HTTP 500 Internal Server Error" можно изменить таким образом, чтобы к нему добавить фразу "Нажмите кнопку Назад, чтобы попробовать снова", или в программный код может быть добавлена проверка на нулевой указатель, и поэтому исключение никогда не возникнет, или внутри обработчика исключений может сработать дополнительный обработчик ошибок. С точки зрения того, кто сообщает о дефекте, все это не имеет значения; проблема в том, что дефект существует, и работа заключается в том, чтобы сообщить о нем точно, а не говорить о том, как его исправить. Это подобно написанию требований, которые говорят, что должно быть сделано, а не как именно.

- КРАТКОЕ ОПИСАНИЕ. Сайт возвращает ошибку 500, если имя пользователя пустое.
- ОПИСАНИЕ.

  Когда пользователь пытается авторизоваться и заполняет пароль, но не заполняет имя пользователя, система возвращает страницу с ошибкой 500 (см. раздел "ПРИМЕЧАНИЯ" с текстом страницы).

  Проверялись различные пароли. Ошибка не возникает, если пароль не вводится.
- ШАГИ ВОСПРОИЗВЕДЕНИЯ.

  Предусловия: пользователь не авторизован.

  1. С помощью любого веб-браузера зайдите на главную страницу сайта.
  2. В текстовом поле **Пароль** введите *foo*.
  3. Убедитесь, что текстовое поле **Имя пользователя** пустое.
  4. Нажмите на кнопку **Войти**.
- ОЖИДАЕМОЕ ПОВЕДЕНИЕ.

  Пользователь видит страницу ошибки с сообщением "Пожалуйста, заполните имя пользователя и пароль".

- НАБЛЮДАЕМОЕ ПОВЕДЕНИЕ.

  Пользователь видит ошибку 500.

- ВЛИЯНИЕ.

  Пользователь, который забывает набрать имя пользователя, но набирает пароль, не увидит ожидаемую страницу с ошибкой.

- СЕРЬЕЗНОСТЬ.

  Обычная. Это граничный случай, но пользователь может не знать о том, что можно нажать кнопку **Назад** и попытаться авторизоваться снова.

- РЕШЕНИЕ.

  Гарантировать, что поле с именем пользователя не пустое во время авторизации.

- ЗАМЕТКИ.

  Текст страницы следующий:

```
500 Internal Server Error
Приносим извинения, что-то пошло не так. Пожалуйста, попробуйте позже
Серверный лог содержит следующую ошибку:
Caught NullPointerException in LoginProcedure, Line 38
```

- КРАТКОЕ ОПИСАНИЕ. "Невидимая стена" на уровне 12 "Удивительного болгарского сантехника".

- ОПИСАНИЕ.

  На уровне 12 через три блока справа от первого Завода Анаконды имеются три блока, размещенных друг на друге. Но они одного цвета с задним планом и поэтому не видны пользователю. Согласно требованиям в игре не должно быть невидимых для пользователя препятствий.

- ШАГИ ВОСПРОИЗВЕДЕНИЯ.

  1. Начните игру с 12-го уровня.
  2. Двигайтесь вправо три экрана.
  3. Перепрыгните через Завод Анаконды.
  4. Пройдите по трем блокам вправо.
  5. Попытайтесь перейти вправо на следующий блок.

- ОЖИДАЕМОЕ ПОВЕДЕНИЕ.

  Игрок переходит на следующий блок.

- НАБЛЮДАЕМОЕ ПОВЕДЕНИЕ.

  Игрок не может двигаться, т. к. на пути невидимые блоки.

- ВЛИЯНИЕ.

  Игрок может быть поставлен в тупик невозможностью двигаться.

- СЕРЬЕЗНОСТЬ.

  Значительная. Это отдельно оговаривается в требованиях и является довольно раздражающим для игрока.

- РЕШЕНИЕ.

  Перепрыгнуть через невидимые блоки, когда ты знаешь, что они на пути.

- ЗАМЕТКИ.

  Нет.

- КРАТКОЕ ОПИСАНИЕ. Пользователь не может авторизоваться в системе.
- ОПИСАНИЕ.

  При попытке авторизоваться в системе с использованием всех обычных и административных учетных записей появляется одно и то же сообщение: "Проблема с авторизацией. Возможно, вы неправильно набрали свой пароль?"

- ШАГИ ВОСПРОИЗВЕДЕНИЯ.

  1. Перейти на страницу авторизации.
  2. Набрать *TestUser1* в поле **Имя пользователя**.
  3. Набрать пароль пользователя TestUser1 в поле **Пароль**.
  4. Нажать кнопку **Войти**.

- ОЖИДАЕМОЕ ПОВЕДЕНИЕ.

  Появляется страница с приветствием.

- НАБЛЮДАЕМОЕ ПОВЕДЕНИЕ.

  Отображается сообщение об ошибке "Проблема с авторизацией. Возможно, вы неправильно набрали свой пароль?"

- ВЛИЯНИЕ.

  Пользователи не могут авторизоваться в системе.

- СЕРЬЕЗНОСТЬ.

  Блокер. В таком состоянии система непригодна к использованию всеми пользователями.

- РЕШЕНИЕ.

  Нет.

- ЗАМЕТКИ.

  Если вы не знаете пароль для пользователя TestUser1, обратитесь к руководителю группы тестирования.

  В логах нет каких-либо необычных записей, связанных с попытками авторизации.

Глава 10

Дымовое и приемочное тестирование

10.1. Дымовое тестирование

Мой отец — водопроводчик. Перед подключением труб в новом здании к водопроводу он заполняет трубы дымом и смотрит, не попадает ли дым из труб в помещения. Зачем он это делает? Потому что, если имеются какие-то зазоры или плохо изолированы стыки, гораздо легче очистить здание от дыма (т. к. он просто рассеется), чем от воды. Заметить дым в комнатах также проще, чем проверять, если ли потечки на стенах. Это гораздо быстрее, чем проверять каждую трубу сантиметр за сантиметром, а в итоге получить ту же самую информацию. После завершения дымового теста поступает вода и начинается другое, более специфическое тестирование, например гарантирующее правильность давления на разных этажах.

Аналогично происходит дымовое тестирование программного обеспечения. **Дымовой тест** (smoke test) объединяет минимальное количество тестов, выполняемых для подтверждения того, что тестируемая система готова к проведению дальнейшего тестирования. Его можно назвать стражем дальнейшего тестирования — до тех пор, пока система не способна выполнять какой-то минимальный набор операций, вы не можете приступить к запуску полноценного тестового набора. Обычно дымовой тест представляет собой небольшое количество тестов, проверяющих, может ли быть установлена программа, работают ли ее основные функции и не возникают ли очевидные проблемы.

Например, если вы тестируете интернет-магазин, вам надо проверить, что пользователь может зайти на сайт, найти товар и добавить его в корзину. При полноценном тестировании проверяются многочисленные граничные случаи и пути, а также такая вспомогательная функциональность, как рекомендации и обзоры. В качестве другого примера можно привести дымовое тестирование компилятора, проверяющее, что он может откомпилировать программу "Привет, мир!", а полученный в итоге исполняемый файл действительно печатает на экране "Привет, мир!". В дымовом тестировании не будет проверяться, что произойдет, когда в коде присутствуют сложные булевы выражения, или если отсутствует библиотека, на которую имеется ссылка в коде, и будет ли выдаваться нужное сообщение об ошибке для кода, который нельзя откомпилировать. Дымовое тестирование просто определяет, что система способна выполнять базовые функции. Это очень полезно, когда

система должна либо работать полностью, либо не работать вообще, например при установке части программного обеспечения на новой архитектуре. Если код скомпилировался неправильно, то весьма вероятно, что программа не запустится. Дымовое тестирование позволит нам обойти всю работу по настройке соответствующего тестового окружения, если программа просто не работает.

Зачем заморачиваться с дымовым тестированием перед выполнением полноценного тестового набора? Разве не найдутся и без него в итоге те же самые проблемы? По всей вероятности, да, но ключевыми словами здесь являются *"в итоге"*. Дымовое тестирование позволяет вам определить, готова ли программа к тестированию без запуска полноценного тестового набора. Зачастую настройка системы для тестирования сопряжена с такими расходами, как установка ПО на серверах и компьютерах клиентов, изучение тест-планов и генерация тестовых данных. Если программа настолько неработоспособна, что даже не выполняет базовые функции, то благодаря дымовому тестированию вы не потратите время на работу, связанную с прогоном всех тестов.

Приведу пример из своей жизни. Однажды я работал над проектом, в рамках которого мы еженедельно выпускали новую версию программы. Каждую неделю эту новую версию необходимо было устанавливать в лаборатории примерно на 15 разных компьютерах. Установка программы и обновление тестовых данных для новой версии — даже при помощи скрипта через сеть — обычно занимали несколько часов. Однако иногда у программы было столько дефектов (или несколько, но критических), что она была практически непригодной для тестирования. Такие ошибки, как постоянное повреждение данных или неспособность пользователей авторизоваться, делали тестирование этой версии более сложным, чем обычно. Вместо того чтобы узнать об этом *после* утра, потраченного на установку и обновление, мы запускали быстрый дымовой тест и узнавали об этих ошибках в самом начале рабочего дня. Благодаря этому тестировщики получали дополнительное время для работы над чем-то еще (например, они могли заняться созданием тестовых скриптов или работой с инструментами автоматизации), а разработчики узнавали раньше, что существует проблема с программой. В результате дефекты исправлялись быстрее, и это означало, что тестировщики могли вернуться к работе по тестированию программы.

Дымовое тестирование может быть **сценарным** (scripted) и **импровизационным** (unscripted). В импровизационном тестировании опытный тестировщик или пользователь просто "играет" с программой некоторое время перед началом настоящего тестирования. Обычно этим занимается человек, хорошо знакомый с программой, для того, чтобы гарантировать, что протестирована наиболее важная функциональность, известные дефекты были исправлены, а программа в целом работает как необходимо. Сценарное тестирование означает, что существует конкретный тест-план, который определяет необходимые для выполнения тест-кейсы. И если сценарное тестирование дает нам отрегулированную среду и помогает отслеживать то, что уже работало, то импровизационное тестирование является гораздо более гибким и позволяет тестировщикам проверить различные аспекты программы для различных релизов.

Еще более мягкой версией дымового тестирования является **медиатестирование**. В нем проверяется, что все соответствующие файлы были записаны на носитель информации правильно и могут быть прочитаны. Это просто окончательная проверка, что диск не был поврежден перед отправкой его заказчику или что файлы были помещены на сервере в нужный каталог. Она может быть совсем простой — файлы на диске должны быть по битам эквивалентны файлам исходной локации, или же чуть более сложной, заключающейся в том, чтобы запустить диск на компьютере и убедиться, что начинается установка программы.

10.2. Приемочное тестирование

Дымовое тестирование проверяет, готова ли программа к тестированию; другими словами, может ли команда тестирования принять программу, чтобы начать работать с ней? Это подмножество **приемочного тестирования**. Приемочным тестированием является любой вид тестирования, который проверяет, может ли система перейти в следующую фазу своего жизненного цикла, будь это тестирование, передача заказчику или подготовка его к использованию потребителем.

В зависимости от области работы приемочные тесты могут быть сценарными или импровизационными. Для крупных подрядных компаний или сложных проектов зачастую существует несколько тест-планов, проверяющих правильность работы различных частей системы, а также показателей качества (например, что во время тестового периода будет обнаружено не более трех значительных дефектов), которые должны быть оценены. Для небольших программ приемочное тестирование может заключаться в разрешении заказчику или пользователю поработать несколько минут с программой, чтобы понять, устраивает она его или нет.

Так как разработчики ПО обычно не являются специалистами в той области, для которой они создают программу, существует большая вероятность, что они создадут программу, которая не удовлетворит нужды потребителя. Даже если требования оговорены ясно, часто существуют характерные для данной предметной области неоднозначности и допущения. Например, для программиста понятна начинающаяся с нуля индексация — во многих популярных языках программирования нумерация элементов массива идет с 0, и мало кто из разработчиков Java напишет цикл `for` таким образом:

```
for (int j=1; j <= 10; j++) { ... }
```

Однако большинство людей, не являющихся программистами, тот факт, что индекс 1 будет указывать на второй элемент списка, приведет в замешательство.

Официальные приемочные тесты обычно определяются заранее и согласовываются разработчиком ПО и заказчиком. Если все тесты проходят или обе стороны приходят к приемлемому для них компромиссу, тогда говорят, что ПО принято заказчиком. Другие возможные решения включают частичный платеж, платеж после исправления определенных дефектов или понимание, что существует определенный уровень качества, не предполагающий прохождения всех тестов (т. е. проходят все основные тесты и по крайней мере 95% незначительных тестов).

Эксплуатационное тестирование, также называемое **полевым тестированием**, представляет собой тестирование системы в реальных (эксплуатационных) условиях. Часто во время процесса разработки или пошаговых проверок проблему не то что не могут найти — о ней даже не могут подумать. Эксплуатационное тестирование позволит найти эти упущенные из вида ошибки путем запуска системы в реальной среде. Например, каждая составляющая часть новой системы авионики может быть протестирована с использованием описываемых в этой книге техник, но система не пройдет эксплуатационное тестирование до тех пор, пока самолет, для которого разработано программное обеспечение, не поднимется в воздух. Зачастую эксплуатационное тестирование является последним видом выполняемого в системе тестирования, которое гарантирует, что система действительно будет работать правильно в реальных условиях. Это может быть довольно дорогой метод тестирования как с точки зрения настройки, так и с точки зрения риска. Например, если система авионики выйдет из строя и самолет полетит не так, как надо, цена этого отказа окажется чрезвычайно высокой; и даже если она работает правильно, полет самолета в течение нескольких часов окажется на несколько порядков дороже, чем покупка компьютерного времени для проведения симуляции и интеграционных тестов.

Другим видом приемочного тестирования является **пользовательское приемочное тестирование**. Обычно используемое в Agile-средах, пользовательское приемочное тестирование (ППТ — user acceptance testing, UAT) проверяет, что система соответствует целям пользователей и работает приемлемо для них. Обычно профильному специалисту (ПС — человек, не являющийся разработчиком и понимающий область, для которой написана программа) дается определенное количество задач, которые необходимо решить. ПС не получает инструкции о том, как выполнять задачи от тестировщика; он самостоятельно определяет, как использовать программу (при помощи соответствующей документации или вспомогательных материалов) для выполнения задач. Если, например, тестируется веб-браузер, для пользовательского приемочного тестирования можно пригласить опытного веб-серфера (если этот термин еще используется) как профильного специалиста. У ПС могут быть три задачи — зайти на главную страницу популярной поисковой системы, добавить страницу в закладки и вернуться на эту страницу через закладки. Участники команды могут наблюдать за действиями ПС, но они не должны никак ему помогать или что-то объяснять. Сдерживание участников команды от попыток оказать помощь может быть сложной задачей, т. к. те, кто разрабатывает программу, обладают более высоким уровнем знаний о системе и хотят показать пользователю "как надо лучше". Однако действия ПС по выполнению задач без какой-либо сторонней помощи покажут вам, что может быть сложным для пользователей, которые будут пользоваться программой самостоятельно.

Альфа-тестирование и **бета-тестирование** можно рассматривать как виды приемочного тестирования. Хотя эти термины могут иметь немного разные значения в различных областях, они подразумевают участие независимых пользователей или команд, использующих разрабатываемое программное обеспечение. Во всех случаях альфа-тестирование предшествует бета-тестированию при условии, что альфа-

тестирование выполняется. В нем участвует очень небольшая группа выбранных заказчиков, чаще всего специалистов высокого класса или с высокими техническими навыками, которые используют программу. Иногда это может быть не заказчик, занимающийся тестированием, а группа тестирования, которая является сторонней по отношению к группе, участвовавшей в разработке. Бета-тестирование подразумевает доступ к релизу более широкого круга потребителей. Обычно оно проводится после альфа-тестирования для больших хорошо протестированных проектов. Однако ничто не может помешать команде отказаться от альфа-тестирования и передать программу напрямую большой группе людей — возможно, всей клиентской базе, оговаривая, что программа является бета-версией. Google славится этим; Gmail, например, технически находился в стадии бета-релиза первые 5 лет своего существования, в течение которых миллионы людей использовали этот почтовый сервис. Если вас не очень беспокоит, что в мире узнают о дефектах, бета-тестирование может стать быстрым способом получить информацию об использовании и сообщения о дефектах от большого количества пользователей.

В онлайн-системах бета-тестирование часто может быть "смешано" с обычным использованием системы. У вас есть некий набор пользователей — скажем, интересующиеся новыми возможностями программы, или важные заказчики, или даже случайная выборка, — и у них есть доступ к новому функционалу, который вы хотите подвергнуть бета-тестированию. Обычно необходимо либо обсудить это с ними, либо убедиться, что они знают о том, что используемый функционал пока не является полным.

Похожим по концепции на приемочное тестирование является подход **"кормление собак"** (dogfooding) или **"съешь сам свою собачью еду"** (eat your own dogfood). Несмотря на несколько шокирующее название, это означает, что разрабатывающая программу команда сама использует эту программу. Это полезно в ситуациях, когда разрабатываемая система является знакомой компьютерным пользователям в общем (например, операционная система или текстовый редактор) или программистам в частности (среда разработки или компилятор). Обычно это невозможно до завершающих стадий проекта, когда наличие достаточного функционала уже позволяет относительно приемлемо пользоваться программой — ведь не имеет смысла "есть свою собачью еду", которая еще не готова. "Собачья еда" позволяет чрезвычайно быстро находить дефекты, т. к. программу использует вся команда, а не только тестировщики. Даже если другие участники команды не заняты целенаправленным поиском дефектов в системе, тот факт, что ее использует большое количество людей — и делает это так, как могли бы другие пользователи, — означает, что, скорее всего, будет обнаружено больше дефектов. Они, как правило, образуют кластеры в областях, в которых работают пользователи, позволяя собрать дополнительные метрики (по использованию и плотности дефектов в различных подсистемах). И как бонус у разработчиков, использующих свою программу, появляются дополнительные стимулы по обеспечению качества своего кода!

Глава 11

Исследовательское тестирование

Мы разобрались, как понимать требования, разрабатывать тест-план и находить дефекты самыми разными способами. Для всего, что мы делали до этого момента, было основополагающее предположение, что мы знаем, каким должно быть поведение программы, ну, или, по крайней мере, кто-то знает. Иногда, особенно в начале разработки и перед тем, как выкристаллизованы все идеи и сама программа, может оказаться, что никто не знает, что должно произойти при определенных обстоятельствах. Это могут быть проблемы со входными значениями, о которых не подумал никто из системных инженеров, или вопросы юзабилити, не предусмотренные при проектировании системы. Нежданные ситуации также могут возникнуть, когда в расписании нет времени для формального тест-плана или для разрабатываемого ПО он кажется ненужным. В конце концов, тестировщики могут захотеть проверить составляющие системы для их понимания и найти дефекты совершенно случайно.

При подобных обстоятельствах вы можете заняться **исследовательским тестированием**. Оно определяется как тестирование без заданного тест-плана, в котором основной целью является изучение разработки системы и влияние на этот процесс в противоположность конкретному определению, соответствует ли наблюдаемое поведение ожидаемому. Исследовательское тестирование позволяет тестировщику действовать самостоятельно, устанавливая граничные случаи по ходу работы или следуя указаниям, если что-то вызывает проблемы. Оно также позволяет тестировщику обучаться на практическом опыте вместо чтения проектной документации или кода, и зачастую это гораздо более эффективный способ понять систему.

11.1. Преимущества и недостатки исследовательского тестирования

На исследовательское тестирование часто ссылаются как на тестирование *ad hoc* (свободное или интуитивное тестирование), но этот термин может означать неаккуратность и небрежную работу. Однако исследовательское тестирование не является небрежным, оно просто менее жесткое. Очень часто много внимания уделяется тому, является ли исследовательское тестирование подходящим, и еще больше внимания на определение того, что необходимо тестировать в дальнейшем.

Исследовательское тестирование зачастую становится первым опытом тестировщика при знакомстве с системой или ее новой функцией. Пошаговые инструкции — добавляемые как шаги выполнения в тест-план — могут быть не определены путем простого чтения проектной документации. Часто проводится некоторое предварительное тестирование для понимания того, как все реализовано. По мере того как тестировщик получает опыт по работе с системой, может потребоваться дополнительное исследовательское тестирование, т. к. теперь тестировщик лучше понимает те закоулки системы, которые не были покрыты предварительным тест-планом. А иногда тестировщику просто нечем заняться, и он начинает охоту на те дефекты, которые остались незамеченными после прохождения тестов, созданных исключительно на основе тест-планов.

Потратив некоторое время на блуждание в сорняках — где еще не побывали косилки тест-планов, — тестировщик может найти странности системы, которые, возможно, никогда не будут обнаружены более формализованным тестированием. Кроме того, исследовательское тестирование часто обнаруживает больше дефектов, и при этом гораздо быстрее. Большее время, потраченное на тестирование, как противоположность мелким заботам, связанным с изучением тест-планов, записью статусов тестов и т. п., означает больше времени, потраченного на поиски дефектов и меньше времени на разработку связанной с этими дефектами документации.

У исследовательского тестирования есть побочное преимущество, заключающееся в том, что тестировщик узнает о системе быстрее, чем при формальном тестировании. Бездумное следование тест-плану тестирования не допускает того обучения, которое характерно для людей. И если вам кажется сложным обучение игре на пианино при помощи чтения книги, то понимание тестируемой системы при помощи чтения и выполнения тест-планов покажется еще более сложным.

Требуется совсем немного усилий для установки ПО или доступа к системе и последующих поисков дефектов, в отличие от шагов, связанных с разработкой и написанием тест-плана и его выполнением. Если у вас есть полчаса на поиск всевозможных дефектов в программе, вам не захочется тратить первые 25 минут на написание предусловий, постусловий, шагов выполнения — и все ради того, чтобы в итоге запустить несколько тестов. Это также означает, что очень легко выполнить обновленный тест, даже если программа существенно изменилась; просто займитесь еще исследовательским тестированием! Не нужно обновлять никакие тест-планы, за исключением тех, что в голове у тестировщика, в то время как для тест-плана (или еще хуже, для автоматизированного теста) может потребоваться внесение достаточно серьезных правок в случае изменения исходной программы.

Но у исследовательского тестирования имеются недостатки. В конце концов, существует причина, по которой тест-планы создаются в первую очередь. Первым недостатком является то, что оно нерегулируемое, а это означает отсутствие заранее подготовленного реального плана, определяющего, что именно следует тестировать. Попытки объяснить не тестировщикам, что именно нужно тестировать и почему это может быть сложно, и даже попытки объяснять задним числом могут оказаться непростыми. При погружении в тестирование некой программы у вас может быть прекрасная воображаемая модель того, что вы делаете и по каким причинам, но зачастую это все теряется после завершения работы.

Большая часть ответственности по принятию решения о том, что именно нужно тестировать, лежит на тестировщике во время исследовательского тестирования. Добросовестный и хорошо понимающий систему тестировщик выполнит работу по обнаружению дефектов гораздо лучше, чем не понимающий систему или ленивый тестировщик. Если оба этих тестировщика будут следовать одному тест-плану, тогда при условии того, что второй из них достаточно добросовестный для выполнения тестов и записи правильных результатов, объем покрытия будет известным и примерно одинаковым для них двоих. Таким образом, формализованный тест-план часто мешает опытным тестировщикам, но помогает неопытным, которым нужна помощь в понимании того, что нужно тестировать.

Другой проблемой, которая может возникнуть во время исследовательского тестирования, является возможная невоспроизводимость обнаруженных во время его проведения дефектов. Может оказаться, что причина падения чего-либо через 10 минут после начала исследовательского теста в том, что тестировщик сделал что-то, немного отличающееся от предыдущего раза в первую минуту тестирования. Попытка повторить все шаги без знания этого отличия может привести к полному разочарованию и, скорее всего, окажется невозможной. С формализованным тестированием, при условии, что тестировщик выполнил все шаги, это перестает быть такой проблемой.

Сложно оценить, какое покрытие получила тестируемая система при помощи исследовательского тестирования. Если тестировщик не записывает в точности, что он делает, либо его действия не фиксируются специальными программами по логированию действий или захвату изображения на экране, могут оставаться такие части системы, в качестве тестирования которых вы не можете быть уверены. Сложно что-то анализировать, когда в наличии так мало тестовых артефактов. Если вы работаете в связанной с безопасностью области или в такой, где важно документировать все, что было протестировано, полагаться исключительно на исследовательское тестирование окажется не лучшей идеей.

Тестирование с течением времени все больше и больше полагается на автоматизацию, и нет признаков, что этот процесс замедлится. Однако поскольку исследовательское тестирование во многом базируется на знаниях тестировщика и его знакомстве с тестируемой системой, на данный момент практически невозможно автоматизировать этот вид тестирования. Если не брать в расчет непредвиденные достижения в области искусственного интеллекта, вам всегда будет нужен человек для выполнения исследовательского тестирования.

11.2. Руководство по исследовательскому тестированию

Два простых шага для исследовательского тестирования:

1. Используйте свое лучшее суждение.
2. Если вы не знаете, что делать, см. пункт 1.

Исследовательское тестирование предполагает, что вы, как тестировщик, понимаете систему или по крайней мере понимаете ее лучше с течением времени. В конце концов, если вы пишете тест-план перед тестированием какого-то свойства системы, то по определению это тот момент, когда вы меньше всего знаете о системе. Вы будете узнавать ее все больше и больше по ходу времени и вашего взаимодействия с ней. Исследовательское тестирование также предполагает, что вы будете способны распознать дефект, даже если не станете проводить строгое сравнение наблюдаемого поведения с ожидаемым. Это означает, что должно быть активное взаимодействие с инженерами, системными проектировщиками и прочими заинтереованными лицами с целью определения, было ли заданное поведение на самом деле ожидаемым.

Как можно провести исследовательское тестирование? Это будет зависеть от конкретной тестируемой системы, но все-таки можно наметить некоторые общие руководящие принципы. Первый из них заключается в проверке того, что часто используется, и тех важных задач, которые должны быть выполнены и работать правильно. Не важно, как система обрабатывает странные угловые случаи, если не работает ни один из простых базовых случаев! Проверьте счастливые пути и убедитесь, что они работают правильно, прежде чем сойти с них в заросли сорняков.

После того как вы определили, что ключевой базовый функционал действует без ошибок, можно задуматься и о граничных случаях. Если имеется запрос на ввод числа, что произойдет, если число отрицательное, или вы ничего не вводите, или вводите последовательность управляющих символов? Если нужно ввести имя файла, то что произойдет, если файл не существует или неправильно форматирован? Если вы не знаете, где что-то пошло не так, обратитесь к *главе 7* и посмотрите, возможно ли воспроизвести какие-то из ошибок в вашей системе.

Посмотрите, что произойдет, когда пересекутся области действия различных функционалов. Если тестируемая система способна прочитать файл, выполните с ним некие действия, запишите результат на диск — и что произойдет, если вы этим файлом перезапишите исходный файл? Что произойдет, если одна часть системы сортирует по возрастанию, а другая по убыванию, — не возникнет ли где-то конфликт сортировок? Если система выполняет сложный рендеринг и одновременно пытается скачать данные из сети, может ли возникнуть "состояние гонки"?[1]

Попробуйте почувствовать себя разработчиком. Даже если вы не программист, можете представить сценарии, о которых не думал тот, кто концентрировался на какой-то конкретной функции программы или пытался заставить правильно работать какой-то метод, в то время как менеджер проекта постоянно запрашивал у него отчеты о состоянии дел. В такой ситуации несложно что-то упустить из виду. Разработчик мог не проверить все нулевые указатели, целочисленные переполнения или не проконтролировать, что произойдет, если запущенные потоки никогда не закончатся. Используя эти знания, вы можете попытаться воспроизвести такие

[1] Ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода. — *Прим. пер.*

ситуации с точки зрения пользователя путем ввода пустых строк, чрезвычайно огромных чисел, превышающих максимально возможное в 32-битном диапазоне, или чтения огромных массивов информации из базы данных.

Не бойтесь следовать своей интуиции. Если вы заметили что-то, что кажется странным, забавным или неожиданным, попробуйте добиться этого другими способами и посмотрите, есть ли что-то, что эти способы связывает. Также у вас могут появиться идеи, которые не пришли бы вам в голову, если бы вы занимались только разработкой тест-плана. Теперь самое время попробовать.

В итоге, хотя исследовательское тестирование является гораздо менее формализованным процессом, чем создание и выполнение тест-планов, вам понадобится описать обнаруженные дефекты как можно быстрее и как можно более подробно. Чем дольше вы ждете, тем больше забудете. Более того, чем дольше вы используете систему, тем вероятнее, что окажетесь в ситуации, что не сможете воспроизвести ошибку. Определение вызвавших дефект шагов и их запись должны превалировать над поиском других дефектов. В конце концов, если вы не отметите дефекты и не предоставите разработчику достаточно информации для их исправления, то нет никакого смысла заниматься тестированием. Вы также можете позже использовать шаги воспроизведения, которые вы нашли, для создания более формализованного теста.

Глава 12

Ручное тестирование против автоматизированного тестирования

До сего момента мы рассматривали преимущественно ручное тестирование, т. е. мы разрабатывали и проходили тест-кейсы самостоятельно. Но если вы занимались ручным тестированием, то знаете, что это отнимающее много времени и часто крайне скучное занятие, в котором тестировщик должен следовать скрипту (гораздо интереснее выполнять какие-то действия по заданиям скрипта, чем работать с запущенной программой). Вот если бы была какая-то машина, которую можно было использовать для выполнения повторяющихся заданий!

Если вы тестируете программное обеспечение, тогда на помощь с тестами вам придет компьютер (ведь у вас же есть компьютер для запуска программы, которую вы тестируете). Существует множество инструментов, которые помогут вам протестировать программу эффективно и продуктивно.

В автоматизированном тестировании вы пишете тесты при помощи языка программирования, скриптового языка или приложения, которые затем выполняются компьютером. Эти тесты могут проверять самые разнообразные аспекты программы, начиная от возвращаемых методами значений до того, как должен выглядеть графический интерфейс на экране.

12.1. Преимущества и недостатки ручного тестирования

12.1.1. Преимущества ручного тестирования

1. *Оно простое и прямое.* Существует причина, по которой в этой книге сперва обсуждается ручное тестирование — с ним проще понять концепции тестирования, не беспокоясь о синтаксисе и причудах используемого языка программирования или инструмента тестирования. Вы используете те же настройки программного обеспечения или по крайней мере очень похожие, что и пользователь. Большинство людей, даже не обладающих обширными техническими познаниями, смогут следовать хорошо написанному тест-плану. Зато у них возникнут большие сложности, например, с пониманием тестов JUnit.
2. *Оно дешевое.* По крайней мере, с наивной точки зрения, нет никаких дополнительных первоначальных затрат. Довольно легко выполнять неформальное тес-

тирование даже без написания тест-плана — просто запустите программу и по ходу думайте о базовых случаях и граничных случаях.

3. *Оно легкое в настройке.* Если вы в состоянии скомпилировать программу и запустить — что, надеюсь, вы можете сделать, т. к. программы, которые нельзя запустить, пользователи обычно не любят, — значит, у вас есть все необходимое для базового ручного тестирования. Созданием автоматизированных тестовых фреймворков, настройкой тестовых драйверов и всей сопутствующей работой для запуска автоматизированных тестов заниматься не надо.

4. *Нет необходимости изучать, покупать и/или писать дополнительное ПО.* Хотя существуют бесплатные приложения для тестирования, необходимо помнить, что цена покупки является не единственной ценой, которую вы заплатите за использование приложения при разработке. Необходимо потратить время на изучение приложения, синтаксис используемого им скриптового языка и разобраться с неочевидными багами, которые в нем присутствуют. Поиск наиболее подходящего приложения для тестирования, обучение пользованию им инженеров по тестированию и написание (а также отладка) скриптов для него могут сократить время на запуск тестов и продумывание дизайна тестов.

5. *Оно чрезвычайно гибкое.* Если изменяется пользовательский интерфейс, сравнительно просто изменить тест-план. Если вы занимаетесь исследовательским тестированием или другим тестированием вне тест-плана, то если тестировщик знает об изменениях, он может изменить выполненные шаги. Даже если тестировщик не знает об изменениях интерфейса, человек может догадаться о различиях просто во время использования программы. Это невозможно в случае с автоматизированным тестированием; если интерфейс изменяется, тогда любой работающий с этим интерфейсом автотест завершится неуспешно, если только в код тестирования не были внесены правки для работы с новым интерфейсом. Впрочем, это может быть и преимуществом; автотесты намного более строгие, что означает, что они лучше отлавливают непредвиденные модификации. Человек может даже не заметить определенные изменения и просто "поплывет по течению".

6. *Более вероятно, что вы будете тестировать именно то, что важно пользователям.* Ручное тестирование запущенной программы выполняется так, как это делал бы пользователь; это своего рода тестирование черного ящика. Выполняя эту работу, тестировщик видит программу так, как ее видит пользователь, и использует ее подобным образом. Это означает не только то, что тестировщик поймет, как используется программа, и получит знания в предметной области, но и то, что его внимание будет сосредоточено на тех аспектах программы, которые важны пользователю. Пользователей не волнуют определенные значения, возвращаемые конкретными спрятанными в глубинах кода функциями; их беспокоит, чтобы конечные результаты были правильными. В ручном тестировании ваше внимание будет автоматически направлено на тот функционал программы, который важен пользователям больше всего, потому что тестировщики также будут пользователями.

7. *Люди могут обнаружить проблемы, которые не увидят автотесты.* Автотесты обнаружат проблемы в том, что они должны проверять. Однако если при их создании не ставилась задача — проверять какую-то часть программы, то они не обнаружат проблему, даже если ее легко заметит человеческий глаз. В качестве примера представим простую программу, подсчитывающую количество слов. Автотесты проверят, что она выдает правильные ответы для различных входных данных. Однако программа также отправляет в терминал дополнительную последовательность управляющих символов, которые изменяют цвет экрана с черного на фиолетовый. Это будет немедленно обнаружено тестировщиком-человеком, который зарегистрирует дефект, но до тех пор, пока автотесты не начнут специально проверять цвет фона экрана, проблема останется незамеченной.

12.1.2. Недостатки ручного тестирования

1. *Оно скучное.* Вы когда-нибудь проводили целый день, нажимая кнопку, затем записывая результат, затем нажимая кнопку, затем записывая результат, *ad nauseam*? Это отличный способ выгореть и оттолкнуть сотрудников. Если это не ваша цель, вы захотите свести количество скучных заданий до минимума.
2. *Оно часто невоспроизводимо.* Если тестировщик вдруг делает что-то не совсем так и обнаруживает дефект, он или она может не воспроизвести в точности шаги, которые вызвали дефект. Такие ситуации несколько смягчаются четко определенными и хорошо проработанными тест-планами, но даже если у вас чрезвычайно проработанные инструкции, может оказаться сложным воспроизвести *те самые* шаги, которые привели к некоему состоянию. Например, представим тест-кейс из тест-плана по тестированию браузера. Один из шагов воспроизведения гласит "откройте новую вкладку". Но существует множество путей сделать это; выберет ли тестировщик опцию **Новая вкладка** из меню, или использует комбинацию клавиш, или щелнет правой кнопкой мыши по ссылке, а затем откроет ее в новой вкладке? Возможно, дефект проявляется только при одной из этих ситуаций. С помощью автотестов вы сможете перезапускать тесты очень точно, выполняя одни и те же шаги каждый раз.
3. *Некоторые задания сложно или невозможно протестировать вручную.* Предположим, что вы тестируете производительность системы, одно из требований к которой гласит, что задержка между запросом и ответом не должна превышать 50 миллисекунд. Так как время человеческой реакции превышает заданную максимальную задержку более чем в четыре раза, то даже если на дисплее информация отобразится мгновенно, проверку вручную все равно осуществить невозможно. Могут быть и другие составляющие программы (например, количество запущенных потоков или значение переменной), которые не видны пользователю, но которые можно протестировать напрямую только при помощи специальных программ или инструментов.
4. *Возможность человеческой ошибки.* Каждый раз, когда для вас кто-то что-то делает, существует вероятность, что этот кто-то допустит ошибку. В случае выполнения тест-кейса человек может прочитать инструкцию неправильно, не так

ее понять, записать статус теста не там, где надо, или же пройти тест неправильно. Чем дольше кто-то выполняет ручное тестирование, тем более вероятно, что возникнет одна из таких ошибок. В некоторых организациях поняли это довольно давно — например, у водителей-дальнобойщиков и пилотов самолетов есть ограничения по времени работы без перерыва. Хотя теоретически автотест может выполнять не те шаги, на которые он запрограммирован, вероятность этого чрезвычайно мала. Компьютеры очень хорошо делают именно то, что вы говорите им делать, и делают это снова и снова.

5. *Оно чрезвычайно требовательно ко времени и ресурсам.* Тестирующий персонал, вручную нажимающий кнопки или делающий что-то с программой, выполняет это с такой же скоростью, с какой и обычный пользователь. Это даже в лучшем случае, потому что тестировщики должны еще прочитать тест-план и следовать ему, внимательно проверять правильность выполнения шагов, отмечать статусы, а также выполнять побочную работу во время ручного тестирования. Прохождение ручных тестов может быть на несколько порядков медленнее, чем хорошо написанных автоматизированных, даже если они выполняют одни и те же действия. Команде тестировщиков потребуется неделя для ручного прогона 1000 тест-кейсов, а на десктопном компьютере эта же работа займет меньше 10 минут. Это означает, что или большая часть рабочего времени тестировщиков будет потрачена на монотонные задания, или что придется нанять больше сотрудников для выполнения тестов.
6. *Оно ограничивает вас тестированиями черного и серого ящиков.* По существу невозможно выполнить тестирование белого ящика вручную для программ, написанных на многих популярных языках программирования. Вы сможете тестировать не отдельные методы или функции, а только их взаимодействие с прочими частями системы. В зависимости от сложности программы это может значительно затруднить поиск дефектов, т. к. вы не сможете работать на самом низшем уровне.

12.2. Преимущества и недостатки автоматизированного тестирования

12.2.1. Преимущества автоматизированного тестирования

1. *Отсутствие человеческого фактора во время выполнения тестов.* Если нужно назвать то, в чем хороши компьютеры, то первым в этом списке будет их способность выполнять одни и те же операции снова и снова одинаково. Напишите рисующую круг программу, и круг, нарисованный после первого запуска программы, будет выглядеть так же, как и круг после тысячного запуска. Люди не способны на такое — линии некоторых кругов будут дрожащими, какие-то круги будут отличаться по размеру, одни будут близки к идеалу, а некоторые окажутся совсем неудачными. Подобным образом и автотест будет каждый раз в точности проходить один и тот же набор шагов выполнения — и не начнет не-

ожиданно выполнять тесты не в том порядке, не будет щелкать не по тем кнопкам, не проставит ошибочные статусы.

2. *Чрезвычайно быстрое выполнение тестов.* Компьютеры могут выполнять задачи и проверять результаты гораздо быстрее, чем люди. Благодаря этому не только выше скорость прохождения тестов, но и можно выполнить больше тестовых прогонов и, как следствие, тестов.
3. *Легкость выполнения после установки системы.* Настройка системы для запуска ручного теста может быть трудоемким процессом. Необходимо установить программное обеспечение, настроить программу для отслеживания тестов, загрузить соответствующие данные в базу данных. В большинстве реализующих автоматизированное тестирование систем запуск набора тестов осуществляется простым щелчком по кнопке в IDE или запуском из командной строки. Это помогает не только снизить объем муторной работы, выполняемой тестировщиками, но и ускорить время итерации между разработкой и тестированием кода.
4. *Легкость повторения.* Как упоминалось ранее, использование автотестов минимизирует вероятность человеческой ошибки. Но даже без таких "ошибок" наличие почти идеально повторяемых тестов снижает вероятность недетерминированных ошибок (т. е. ошибок, которые только иногда возникают при выполнении теста, а не при каждом его запуске). Выполнение в точности тех же шагов в том же порядке уменьшает возможность того, что тест упадет "случайно", хотя и не устраняет ее полностью. К слову, эти "случайные" падения почти всегда имеют некую первопричину, просто вы как тестировщик еще не нашли ее.
5. *Проще анализировать процесс.* Поиск шаблонов в падениях тестов становится гораздо проще, когда тест-кейсы можно перезапускать множество раз и сохранять их статусы. Если сохраняется информация о прошедших тестовых прогонах, то даже возможно определить, какие тест-кейсы и виды тест-кейсов обнаруживают дефекты, а какие всегда проходят, потому что особенности тестируемой ими системы изменились незначительно или содержат совсем немного дефектов.
6. *Меньшая ресурсоемкость.* Запуск тысячи тестов на вашем локальном компьютере может сделать его на 10 минут недоступным для других задач, но это определенно лучше, чем если бы ваша команда тестировщиков работала над этими тестами неделю. Вычислительные мощности становятся год от года все дешевле, чего нельзя сказать о времени инженеров.
7. *Идеально для тестирования некоторых составляющих системы, которые сложно тестировать вручную.* Те части системы, с которыми пользователь не работает напрямую (например, внутренние переменные или записываемые в базу данные), потребуют дополнительного труда для проверки вручную. Иногда это может быть даже невозможно.
8. *Хорошая масштабируемость.* Добавление автотестов часто приводит к сублинейному увеличению времени на их выполнение, т. к. большая часть времени будет потрачена на их разработку. То есть добавление пяти тест-кейсов к существующему тест-плану из пяти тест-кейсов увеличит время выполнения значи-

тельно меньше, чем в два раза. Добавление дополнительных ручных тестов приведет к сверхлинейному увеличению времени выполнения, т. к. выполняющие вручную тестирование люди не могут поддерживать постоянную скорость и обычно замедляют работу по мере выполнения тестов. Все больше времени будет требоваться на перепроверку результатов, т. к. исчерпываются внутренние силы.

12.2.2. Недостатки автоматизированного тестирования

1. *Требуется дополнительное время на установку.* После того как вы поймете основные принципы написания тест-плана (которые, я надеюсь, читатели уже изучили в предыдущих главах), будет довольно просто написать тест-план и определить тест-кейсы для компонентов программы. Однако если вы пишете автотесты, вам необходимо убедиться не только в том, что написанный код можно протестировать, вам также придется установить библиотеки для тестирования и тестовый фреймворк, а также, возможно, изучить новый синтаксис, терминологию и парадигмы. Даже если это фреймворк, который вы использовали ранее, нужно потратить какое-то время для настройки его работы под конкретную тестируемую систему.
2. *Возможно, не удастся выявить некоторые дефекты, с которыми может столкнуться пользователь.* Даже если вы собираетесь искать их, используемые вами тестовые фреймворки и библиотеки могут не обнаруживать дефекты или определенные виды дефектов. Например, предположим, что у вас есть собственный фреймворк для пользовательского графического интерфейса вашего приложения. Фреймворк может вам рассказать о содержимом окна приложения и о его видимости, но не его расположении. Если система не отображает окно в необходимом месте, вы не узнаете об этом, используя автоматизированную систему. Ситуацию может облегчить использование распространенных фреймворков и систем отображения, но такая возможность имеется не всегда.
3. *Требуется обучение написанию автоматизированных тестов, а также работе с дополнительными инструментами и фреймворками.* Написание автотестов является знанием, которое сильно отличается от знания ручных тестов. Хотя логика и теория являются общими, написание автотестов требует еще большего внимания к точности, а также тому, что взаимодействие с системой было описано предельно подробно.
4. *Требуются более квалифицированные сотрудники.* Хотя ручное тестирование может быть выполнено без обширных технических знаний, для автотестов требуется базовое понимание программирования или написания скриптов. Тестировщики должны быть либо уже знакомы с фреймворком тестирования, либо необходимо потратить время и ресурсы для его изучения. В этот период у них будет невысокая производительность по созданию новых тестов и поиску с их помощью дефектов.
5. *Автотесты тестируют только то, что они ищут.* Если люди, выполняющие тесты, легко заметят проблему с какой-то частью программы, даже если не ищут

ее, то компьютеры так не смогут. Если вы не написали тест по проверке, что 5 + 5 в калькуляторе даст 10, а программа вычислит 11, то сообщение об ошибке вы не получите.

6. *Могут появиться тавтологические и прочие плохие тесты.* В больших тестовых наборах часто накапливается хлам, т. к. люди могут не проверять их довольно долгое время. Если тесты проходят, он остается незамеченным, а между тем небольшие изменения и правки будут просачиваться в код. Со временем это может привести к появлению **тавтологических тест-кейсов** (т. е. тестов, которые всегда проходят, потому что они проверяют всегда истинные условия, например, что `1 == 1`). Или же, некоторые тесты могут стать ненужным (например, проверяющими те части программы, которые больше не используются или недоступны пользователям). Без явной работы по обеспечению актуальности тестов вы не сможете обнаружить, что ваш компьютер тратит на тестирование больше времени, чем необходимо.

12.3. Реальный мир

Если выбирать между ручным и автоматизированным тестированием, то какое лучше? "Лучше", конечно же, является слишком общим термином; при принятии решения следует учитывать множество факторов. Один из интересных фактов тестирования программного обеспечения заключается в том, что автоматизированные и ручные тесты не сильно пересекаются; то, что хорошо работает в ручном тестировании, не подходит для автоматизированного, и наоборот. При этом бывает, что что-то сложно обнаружить и традиционным автоматизированным тестированием, и ручным, например при проведении тестирования безопасности или исследовании множества граничных случаев. Мы обсудим некоторые вариации в тестировании в последующих главах, которые помогут невелировать слабые стороны традиционных видов тестирования, рассматриваемых сейчас.

На решение об использовании или неиспользовании ручного или автоматизированного тестирования для какой-то конкретной системы, подсистемы или требования может повлиять множество факторов:

1. Насколько важны тесты?
2. Какое влияние оказывает пользовательский интерфейс или другие "нематериальные" аспекты на завершенный продукт?
3. Как часто будут запускаться тесты?
4. Насколько опытны тестировщики в команде?
5. Какое расписание тестирования?
6. Насколько сложно автоматизировать тесты?

В общем, большая часть тестирования с точки зрения количества запускаемых тестов будет так или иначе автоматизирована. Запуск хорошо спроектированного юнит-теста может занять всего лишь несколько микросекунд компьютерного времени; даже простейший ручной тест потребует гораздо больше времени. Автотесты

легче запускать часто и быстро. При прочих равных условиях лучше иметь больше быстрых тестов и запускать их чаще, чем меньше медленных тестов, запускать которые получается реже.

Когда вам, как тестировщику, нужно автоматизировать тесты? Если тест может быть автоматизирован за приемлемое время, вам почти всегда необходимо его автоматизировать. Это упрощает ваше тестирование и процесс разработки, защищает от человеческих ошибок в будущих тестовых прогонах и делает вашу жизнь менее скучной. Однако иногда возникают проблемы, которые помешают вам писать автотесты; для части системы может отсутствовать тестовый фреймворк, дефицит времени не даст вам заняться этой работой, или может просто не получиться заставить работать нужные вам инструменты автоматизации при определенных условиях. В этих случаях имеет смысл выполнять ручной тест и позже вернуться к нему для автоматизации. Небольшое предупреждение — в большинстве случаев "позже" никогда не наступает, и вы застреваете в рутинной и повторяющейся работе.

В "реальном мире" разработки ПО практически все организации, с которыми я взаимодействовал, использовали смесь ручных тестов и автотестов для своих приложений, при этом больше внимания уделяя автоматизированному тестированию. В целом автоматизированное тестирование оказывается очень выгодным для современной разработки, особенно позволяя быстро выполнить тесты после внесения изменений для того, чтобы убедиться, что не возникли дефекты регрессии. Хотя, конечно же, существуют затраты на внедрение автоматизированного тестового фреймворка, но для нетривиальных проектов преимущества от выполнения компьютером работы по тестированию быстро перевесят эти недостатки. С другой стороны, релизы программы, протестированной исключительно автотестами — без какой-либо проверки, как она работает, с точки зрения пользователя, — осуществляется только теми организациями, в программном обеспечении которых не важна работа пользователя, которые имеют чрезвычайно высокий уровень доверия к своим автотестам и/или которые очень безрассудны.

Глава 13

Введение в юнит-тестирование

В предыдущей главе мы обсуждали преимущества и недостатки автоматизированного тестирования по сравнению с ручным тестированием. Теперь мы начнем погружаться в отдельный вид автоматизированного тестирования — юнит-тестирование[1], которое позволит вам тестировать код напрямую. Юнит-тестирование поможет вам убедиться, что отдельные методы и функции работают так, как должны.

13.1. Юнит-тестирование: сама идея

Что такое юнит-тесты? В двух словах, они тестируют наименьшие модули (юниты) функциональности, которые в традиционном объектно-ориентированном программировании обычно означают методы, функции и объекты. Это суть тестирования белого ящика; для того чтобы писать правильные юнит-тесты, вам необходимо понимать реализацию системы на самом глубоком уровне. Они позволяют вам проверить, что написанные вами методы делают с учетом конкретных входных данных то, что вы ждете от них.

Ниже приведены некоторые примеры того, что тестируют юнит-тесты.

1. Метод `.sort()` возвращает [1, 2, 3], когда вы подаете на вход [3, 2, 1].
2. Передача ссылки на нулевой указатель в качестве аргумента функции приводит к выбрасыванию исключения.
3. Метод `.formatNumber()` возвращает форматированное соответствующим образом число.
4. Передача строки в качестве входного аргумента функции, которая принимает целочисленные значения, не приводит к неработоспособности программы.
5. У объекта имеются методы `.send()` и `.receive()`.
6. Создание объекта с параметрами по умолчанию устанавливает для атрибута `default` значение `true`.

---

[1] Или модульное тестирование. — *Прим. пер.*

Как вы можете увидеть, эти функции тестируют те составляющие программы, которые напрямую не видны конечному пользователю. Ему никогда не потребуется работать с определенным методом или объектом. И его не волнует, что имеется у метода или объекта, какие значения установлены для переменных или что произойдет, если в качестве аргумента передана ссылка на нулевой указатель. Конечно же, они могут увидеть *результаты* всего этого, но они не увидят, какая именно часть кода стала причиной.

Юнит-тестирование обычно выполняется разработчиком, который пишет код. Разработчик хорошо знаком с программным кодом и должен знать, какие данные должны быть переданы, какие существуют граничные и угловые случаи, чему равны граничные значения и т. п. Лишь в отсутствие разработчика тестировщик белого ящика должен заниматься разработкой юнит-тестов.

Несмотря на это, очень важно понимать юнит-тестирование, даже если вы не разработчик. Во многих организациях тестировщики программ помогают с процедурами юнит-тестирования. Мы не говорим про их написание; это может быть разработка инструментов, которые помогут сделать написание юнит-тестов легче, или проверка, что юнит-тесты тестируют то, что от них ожидает разработчик, или обеспечение того, что тесты подходящие и покрывают все существенные случаи.

Ранее мы отметили, что пользователь никогда напрямую не увидит те составляющие программы, которые тестируются юнит-тестами. Тогда какой в нем смысл? Не будет ли более разумным тестировать все с точки зрения пользователя? Таким путем мы отловим лишь те дефекты, которые оказывают влияние на пользователя. Нам не придется беспокоиться о том, что там происходит на низком уровне. И, возможно, мы сэкономим время, если не будем писать юнит-тесты, так? К сожалению, не в этом случае. Существует множество причин для написания юнит-тестов в дополнение к тестам системного уровня. Обратите внимание, впрочем, что существует много видов дефектов, которые плохо отлавливают юнит-тесты, и поэтому юнит-тестирование должно быть только частью вашей стратегии тестирования.

1. *Проблемы обнаруживаются раньше.* Обычно юнит-тесты пишутся совместно с кодом. Нет необходимости ждать запуска системных тестов (как правило, более продолжительных), разработки ручных тестов, или когда очередная сборка программы попадет в руки тестировщиков. Разработчик находит проблему еще во время создания ПО. Как и в большинстве случаев, чем быстрее в процессе найдена проблема, тем дешевле, быстрее и проще ее исправить.
2. *Более быстрая оборачиваемость.* Если проблема найдена, не надо ждать, когда произойдет сборка или же когда она попадет в руки тестировщиков. Разработчик находит проблему во время выполнения набора юнит-тестов, а затем может немедленно вернуться к коду и заняться исправлением. Если разработчик должен ждать, когда тестировщик изучит программу, оформит дефект и назначит ответственного за исправление, может пройти немало времени до того, как разработчик наконец-то исправит этот дефект. За это время он или она могли забыть о каких-то деталях реализации, или же, по крайней мере, что-то могло по-

меняться и на восстановление контекста потребуется время. Чем раньше дефект найден, тем быстрее он может быть исправлен.

3. *Более быстрое выполнение, чем у тестов системного уровня.* У хорошо разработанного юнит-теста нет зависимостей от других библиотек, классов, файлов и т. п. Это означает, что он может быть чрезвычайно быстрым и выполнение большинства тестов займет несколько миллисекунд или меньше. Сосредоточив внимание на конкретных частях кода, эти части можно тестировать много раз без дополнительных настроек и увеличения времени выполнения, вызванного другими аспектами системы.
4. *Разработчики понимают свой код.* Написание тестов позволяет разработчику понимать ожидаемое поведение функции. Оно также дает возможность разработчику добавлять тесты для тех составляющих функции, которые могут оказаться проблемными. Например, разработчик может знать, что функция сортировки некорректно сортирует цифровые строки, потому что необходимо установить определенные флаги для того, чтобы она обрабатывала их как числа, а не как строки. Если мы сортируем по цифровому значению, то 1000 больше 999. Но если функция рассматривает входные данные как строки, то 999 окажется "больше", чем 1000. Если вам непонятно, почему так происходит, представьте числа как странно выглядящие буквы, где A = 0, B = 1, C = 2 и т. д. В этом случае 1000 будет эквивалентна BAAA, а 999 будет эквивалентно JJJ. В словаре BAAA (1000) окажется впереди JJJ (999), но если мы рассматриваем их как числа, то 1000 (BAAA) окажется после JJJ (999). Тестировщик черного ящика может не понять, что происходит "под капотом", и не задумается о том, что надо протестировать этот конкретный граничный случай.
5. *Живая документация.* Тесты становятся своеобразной "живой документацией" для кода. Они позволяют понять, почему кодовая база такая, какая есть, иным образом, чем сам код и любые комментарии или документация к программе. Неуспешные тесты обновляются; они либо удаляются, либо изменяются вместе с программой. В отличие от традиционной документации, если юнит-тесты выполняются на регулярной основе (например, до слияния с основной веткой), то такие тесты не могут стать устаревшими. Устаревшие тесты обычно не проходят, и это яркий индикатор для их исправления.
6. *Альтернативная реализация.* Можно представить тесты как другую реализацию программы. В некотором смысле программа — это просто список того, что должен делать компьютер при определенных входных данных, ЕСЛИ `foo` (IF foo) меньше 5, ТОГДА (THEN) напечатать `"small foo"`. КОГДА (WHEN) создается новый объект `bar`, УСТАНОВИТЕ (SET) для переменной `baz` значение `false`. Всеобъемлющий набор тестов предоставляет подобную услугу, говоря немного по-другому, что программа должна делать. Всегда есть место для ошибки, но если вы реализуете задуманное один раз в качестве теста и один раз в виде кода, то вероятность того, что оба будут написаны неправильно, окажется меньше.
7. *Возможность определить, вызвали ли изменения кода проблемы в другом месте.* Современные программы могут быть довольно большими и сложными, и не

всегда легко — а зачастую человеку даже невозможно — узнать, не стали ли внесенные вами изменения причиной каких-то неожиданных последствий. Автор сбился со счета, сколько раз он, скажем, собирался изменить цвета экрана, и это становилось причиной переполнения стека в какой-то рекурсивной функции в коде, для которой по каким-то причинам требовалось, чтобы цвет экрана был зеленый. Однако обладая сравнительно полным тестовым набором, разработчик может легко проверить, не нарушает ли он что-нибудь очевидное в коде. Такой подход не защищен от ошибок, но он определенно позволяет избежать множества проблем.

13.2. Пример на естественном языке

Перед тем как погрузиться в код, давайте рассмотрим юнит-тест на естественном языке. Скажем, мы реализуем класс `LinkedList` и хотим проверить равенство. Если мы создаем два списка с одинаковыми данными, они будут показаны как равные, хотя не являются одним объектом `LinkedList`. Как мы можем определить тест-кейс для такого случая?

"Создай два связанных списка, *a* и *b*, каждый с данными 1 → 2 → 3. Когда списки сравниваются оператором равенства, они должны быть показаны как равные".

Можно увидеть, что здесь имеются три шага, которые соответствуют шагам в ручном тесте. Это неудивительно. В конце концов, базовые концепции тестирования не изменяются, несмотря на уровень абстракции конкретного теста. Эти шаги:

1. *Предусловия.* Сперва нам необходимо создать предусловия для теста. Если вы помните из предыдущих глав, предусловия являются условиями, которые должны быть выполнены перед началом выполнения теста. В нашем случае предусловиями является то, что у нас есть два различных связанных списка, названные *a* и *b*, и они содержат одинаковые данные, 1 → 2 → 3, принадлежащие одному типу данных.

2. *Шаги выполнения.* Они составляют поведение, которое мы намереваемся протестировать в юнит-тесте. В нашем примере оператор равенства применяется к двум связанным спискам *a* и *b* и должен вернуть булево значение.

3. *Ожидаемое поведение.* Помните, что ключевой принцип тестирования программного обеспечения заключается в том, что ожидаемое поведение должно равняться наблюдаемому поведению. Юнит-тест должен определять ожидаемое поведение и затем проверять, равняется ли оно наблюдаемому поведению (т. е. фактическим результатам теста). В мире юнит-тестирования это называется **утверждением** (assertion) — тест утверждает, что ожидаемое поведение равняется наблюдаемому поведению. В нашем случае наш тест будет утверждать, что возвращаемое значение сравнения двух связанных списков является истиной.

13.3. Превратим наш пример в юнит-тест

Вполне возможно провести юнит-тестирование в "ручном" стиле. Просто создайте программу, которая создает связанный список *a*, создает связанный список *b*, применяет к ним двоим оператор равенства и в итоге проверяет, является ли результирующее значение истинным. Если да, напечатаем "тест пройден!"; в противном случае напечатаем "тест неуспешный!". Но обычно мы используем тестовый фреймворк для автоматизации большей части работы и гарантирования, что мы проводим тестирование согласованным и последовательным образом.

В этой книге мы будем использовать JUnit (**http://junit.org**) в качестве фреймворка юнит-тестирования. JUnit является экземпляром тестового фреймворка xUnit, который, в свою очередь, произошел от SUnit, тестового фреймворка для Smalltalk. Хотелось бы отметить, что Smalltalk является одним из тех языков, которые опередили свое время. Если вы хотите увидеть будущее программного инжиниринга, то лучше всего посмотреть, какие крутые возможности были у этих языков 20 лет назад и считались слишком медленными, слишком незрелыми или слишком "академическими". Обычно эти крутые возможности входят в моду годы спустя, когда сообщество очередного языка, в котором они оказались, громко трубит об их новизне (см. сборка мусора, макросы, метапрограммирование).

Но оставим занудствование. Тестовый фреймворк JUnit позволяет нам создавать юнит-тесты, в которых большая часть "закулисной" работы уже проведена. Разработчик может сфокусироваться на создании тестовой логики и понимании, что именно тестируется, вместо того чтобы тратить время на написание вывода сообщений "вау, тест прошел" или "тьфу, тест неуспешный".

Хотя мы будем рассматривать JUint, т. к. это популярный и легкий для понимания тестовый фреймворк, он является далеко не единственным фреймворком юнит-тестирования. Среди других тестовых фреймворков для Java можно назвать более богатый возможностями TestNG; JTest, который может автоматически генерировать юнит-тесты; и cucumber-jvm, позволяющий писать тесты на понятном любому человеку языке. У каждого из них есть свои достоинства и недостатки. Если вы ищете новые возможности для юнит-тестирования, просто наберите в поиске "фреймворки юнит-тестирования *ваш язык программирования*".

Впрочем, необходимо помнить, что конкретная реализация используемого тестового фреймворка совсем не так важна, как концепции, которые вы изучаете и можете применять. При чтении этой главы меньше беспокойтесь о синтаксисе и больше о понимании концепций юнит-тестирования. Думайте о том, как аспекты юнит-тестирования одновременно схожи и различны с концепциями, которые вы уже изучили в ручном тестировании.

Далее приведена реализация юнит-теста, проверяющего связанные списки на равенство, как указано выше. Не беспокойтесь, если вы не понимаете весь код, в последующих разделах он будет рассмотрен подробно.

```
import static org.junit.Assert.*;
public class LinkedListTest {
```

```
    @Test
    public void testEquals123() {
        // Предусловия - для a и b устанавливаются значения 1 -> 2 -> 3
        LinkedList<Integer> a = new LinkedList<Integer>( [1,2,3] );
        LinkedList<Integer> b = new LinkedList<Integer>( [1,2,3] );
        // Шаги выполнения - выполнить оператор равенства
        boolean result = a.equals(b);
        // Постусловия/ожидаемое поведение - утверждение, что
        // результат является истинным
        assertEquals(true, result);
    }
}
```

Глядя на этот тест, легко найти параллели с ручным тестом. Имеются некие предусловия перед началом теста, определенные шаги выполнения и постусловия, позволяющие сравнить наблюдаемое поведение с ожидаемым. Как упомянуто в одной из предыдущих глав, ключевой концепцией является понимание, что при тестировании программы некое поведение ожидается при определенных обстоятельствах, и когда эти обстоятельства наступают, тест должен проверить, что наблюдаемое поведение фактически является ожидаемым поведением. Хотя юнит-тесты проверяют меньшие модули функциональности, чем большинство тестов черного ящика, основная концепция остается неизменной.

Когда тест завершается неуспешно, конкретная причина этого будет отображаться на экране. Мы ожидаем, что тесты должны проходить, поэтому обычно они отображаются просто символом ".". Если вы используете среду разработку или иное графическое представление теста, неуспешные тесты часто помечаются красным цветом, а успешные — зеленым.

13.3.1. Предусловия

Перед тем как запустить тест, вам надо определить необходимые предусловия для теста. Это похоже на предусловия в ручном тесте, только вместо того, чтобы сосредоточиваться на системе в целом, вы сосредоточиваетесь на настройке параметров для конкретного вызываемого метода. В приведенном выше примере юнит-тест должен проверить, что в двух связанных списках одинаковые значения (а именно, 1 → 2 → 3) будут рассматриваться как равные при помощи метода `.equals` объекта связанного списка. Для того чтобы протестировать на равенство два связанных списка, сперва мы должны создать два связанных списка и установить для их узлов одинаковый набор значений. Этот код просто создает списки и помещает их в переменные `a` и `b`. Мы будем использовать два этих связанных списка в следующей фазе юнит-тестирования.

13.3.2. Шаги выполнения

Здесь осуществляется проверка равенства. Мы определяем, является ли `a.equals(b)` истинным или нет, и помещаем булево значение результата в переменную `result`.

13.3.3. Утверждения

Вспомним, что утверждения проверяют соответствие ожидаемого поведения тому, которое наблюдается при шагах выполнения. А именно, мы используем утверждение `assertEquals`, которое проверяет, что результирующее значение является истинным (`true`). Если это так, тест проходит, и мы можем сказать, что при данных условиях метод `.equals()` работает правильно. Если нет, то тест завершается неуспешно.

Существуют разнообразные утверждения, которые можно использовать. Некоторые из них являются взаимозаменяемыми; например, вместо утверждения, что `result` должен равняться `true`, мы бы могли напрямую утверждать, что результат является истинным. В Java это выглядит так:

```
assertTrue(result);
```

Приведем список наиболее часто используемых утверждений вместе с простыми примерами использования:

1. `assertEquals`: утверждает, что два значения равняются друг другу, например `assertEquals(4, (2 * 2))`.
2. `assertTrue`: утверждает, что выражение истинное, например `assertTrue(7 == 7)`.
3. `assertFalse`: утверждает, что выражение ложное, например `assertFalse(2 < 1)`.
4. `assertNull`: утверждает, что значение переменной является `Null`, например для неинициализированной переменной `assertNull(uninitializedVariable)`.
5. `assertSame`: утверждает, что переменные не только равны, но и указываются на один и тот же объект. Например:

```
Integer a = Integer(7);
Integer b = Integer(7);
Integer c = a;
assertSame(a, b); // Ложное утверждение; сравниваемые значения являются
                  // одинаковыми, но ссылаются на разные объекты
assertSame(a, a); // Истинное утверждение; одинаковые ссылки на один объект
assertSame(a, c); // Истинное утверждение; разные ссылки на один объект
```

Кроме того, существует несколько инвертированных версий этих утверждений, таких как `assertNotEquals`, которые проверяют, что исходное утверждение не является истинным. Например, `assertNotEquals(17, (1 + 1))`. По моему опыту, такие утверждения используются гораздо реже. Обычно при возможности хочется проверить конкретное *ожидаемое* поведение, а не то, что это *не неожидаемое* поведение. Проверка того, что что-то не существует, может быть индикатором того, что тест хрупкий или непродуманный. Представьте, что вы написали метод, который будет генерировать романтические стихи XIX века. Вы знаете, что эти стихи не должны

начинаться со слова "гомоиконичность", поэтому вы пишите тест, чтобы проверить это:

```
@Test
public void testNoLispStuff() {
    String poem = PoemGenerator.generate("19th_Century_Romantic");
    String firstWord = poem.split(" ");
    assertNotEquals("homoiconicity", firstWord);
}
```

Когда ваши стихи начинаются со слов "Узрите", "Дорогая" или "Ясный", тест пройдет. Однако он также пройдет, если стихи начинаются с `%&*()_` или `java.lang.StackOverflowError`. В общем, тест должен искать *позитивное* поведение, а не отсутствие *негативного*. Представьте тестирование, проверяющее, что на веб-странице *не появляется* сообщение с приветствием. И если на странице отображается сообщение об ошибке 500 Internal Server Error, то тест все равно пройдет. Тщательно продумывайте случаи сбоев при тестировании на отсутствие определенного поведения.

13.3.4. Обеспечение проверки тестами того, что вы ожидаете

Один из простейших способов добиться этого — сперва обеспечить, чтобы ваши тесты проваливались! Хотя мы подробно рассмотрим стратегию разработки, которая всегда требует, чтобы тесты сперва проваливались, в *главе 15*, посвященной разработке через тестирование, небольшая правка теста часто может доказать, что он не проходит успешно все время, потому что, например, вы ошибочно утверждаете, что `true == true`.

Еще хуже, если вы тестируете что-то совершенно другое, а не то, что, как вам кажется, вы тестируете. Например, предположим, что вы тестируете возможность использования отрицательных чисел в вашем новом методе `absoluteValue()`. Вы пишете тест:

```
@Test
public void testAbsoluteValueNegatives() {
    int result = absoluteValue(7);
    assertEquals(result, 7);
}
```

Это совершенно валидный и проходящий тест. Однако из-за опечатки он в действительности не тестирует ничего, связанного с отрицательными числами. В тесте не проверяется, что вы ввели то, что вам хотелось ввести; компьютер всегда делает то, что ему сказали, а не то, что вы *имели в виду*.

Вы также должны удостовериться, что понимаете, что должен делать метод, либо может оказаться, что вы тестируете поведение, которое ожидаете вы (как разработчик/тестировщик), но не ожидает пользователь (или требования к программе). Например, рассмотрим тест, который является переписанной версией одного из предыдущих тестов.

```
@Test
public void testAbsoluteValueNegatives() {
    int result = absoluteValue(-7);
    assertEquals(result, -7);
}
```

Если вы не понимаете, что должно произойти после получения методом отрицательного значения (в данном случае метод должен вернуть значение без отрицательного знака), тогда вы напишете совершенно неправильный метод, который возвращает совершенно неправильное значение, но ваши тесты будут проходить, потому что они будут проверять, будет ли возвращено некорректное значение. Помните, что тесты — это не магия; им нужно говорить, какое должно быть ожидаемое значение. Если вы ожидаете ошибочное значение и наблюдаемое значение соответствует ошибочному, тест пройдет. Тесты не безошибочны. Вам по-прежнему необходимо использовать свой ум, чтобы убедиться, что они всё проверяют правильно.

В некоторых случаях вы можете совсем ничего не тестировать! Тест JUnit падает, только если ошибочно утверждение. Если вы забыли добавить утверждение, тогда тесты все равно пройдут, вне зависимости от того, какие шаги выполнения вы в него добавите.

В рассмотренном ранее тесте сравнения связанных списков что бы вы могли изменить, дабы удостовериться, что ваши тесты тестируют именно то, что вам нужно?

Что произойдет, если вы измените данные первого связанного списка так, что он будет содержать 1 → 2?

```
@Test
public void testEquals123() {
    LinkedList<Integer> a = new LinkedList<Integer>( [1, 2] );
    LinkedList<Integer> b = new LinkedList<Integer>( [1, 2, 3] );
    boolean result = a.equals(b);
    assertEquals(true, result);
}
```

Или 7→ 8 → 9?

```
@Test
public void testEquals123() {
    LinkedList<Integer> a = new LinkedList<Integer>( [7, 8, 9] );
    LinkedList<Integer> b = new LinkedList<Integer>( [1, 2, 3] );
    boolean result = a.equals(b);
    assertEquals(true, result);
}
```

Или вы хотите заменить проверку равенства проверкой *неравенства*?

```
@Test
public void testEquals123() {
    LinkedList<Integer> a = new LinkedList<Integer>( [1, 2, 3] );
    LinkedList<Integer> b = new LinkedList<Integer>( [1, 2, 3] );
```

```
    boolean result = !(a.equals(b));
    assertEquals(true, result);
}
```

Во всех этих примерах тест должен провалиться. Здесь вы можете немного отдохнуть, зная, что ваш тест не является тавтологическим (всегда завершающимся успешно, вне зависимости от того, что делает код) или тестирующим что-то отличное от того, что, вы думаете, он тестирует.

Вы можете захотеть, чтобы ваш тест сперва провалился, и, таким образом, при его прохождении вы будете знать, что возникшая при тестировании проблема исправлена. И хотя по-прежнему нет гарантии, что ваши изменения исправили проблему — в конце концов, ваше утверждение может быть неверным! — но это все же весьма вероятно.

13.4. Проблемы с юнит-тестированием

Юнит-тестирование и техники, которые мы на данный момент изучили, помогут нам продвинуться довольно далеко, но не пройдут за нас всю дорогу. Просто используя утверждения и код, который мы рассматривали, нельзя проверять, например, что будет выведена конкретная строка, или появится окно, или будет вызван другой метод..., и много всего важного. В конце концов, если бы все методы возвращали разные значения для различных входных данных, никогда не отображая их пользователю и не взаимодействуя с окружением, у нас не было бы возможности узнать, как работают наши программы. Нашими единственными выходными данными стали бы общее увеличение шума вентилятора и нагрев процессора.

Любое поведение помимо возврата значения называется **побочным эффектом**. Отображение окна, печать текста, связь с другим компьютером в сети — все это с терминологической точки зрения побочные эффекты вычислений. Даже присвоение переменной или запись данных на диск являются побочными эффектами. Функции и методы без побочных эффектов, которые только получают данные из параметров, называются **чистыми**. Чистые функции всегда будут возвращать одинаковый результат для одинаковых входных значений и могут вызываться бесконечное количество раз без изменения каких-либо других аспектов системы. Функции и методы, у которых *имеются* побочные эффекты или которые могут выдавать различные результаты, зависящие еще от чего-либо помимо переданных в качестве параметров значений, являются **нечистыми**. Некоторые языки, такие как Haskell, проводят четкое разделение между чистыми и нечистыми функциями, но в Java такого нет.

Примером чистой функции может быть математическая функция, например, вычисляющая квадратный корень, и в Java она будет выглядеть так:

```
public double getSquareRoot(double val) {
    return Math.sqrt(val);
}
```

Если предположить, что отсутствуют связанные с плавающей запятой ошибки или подобные, квадратный корень из 25 всегда будет равняться 5, вне зависимости от того, какие глобальные переменные установлены, вне зависимости, сколько сейчас времени и какая сегодня дата, вне зависимости от всего. Нет и побочных эффектов от вызова функции квадратного корня; здесь нет выскакивающего окошка каждый раз после того, как ваша система сосчитает квадратный корень.

Примером нечистой функции может быть вывод данных о глобальных переменных, или любой метод, выводящий что-то в консоль или на экран, или зависящий от переменных, которые специально не передаются. В целом, если вы видите метод, который ничего не возвращает (void), то он, вероятно, нечистый — чистая функция, возвращающая тип `void`, будет абсолютно бесполезной, т. к. возвращаемое значение является единственным способом взаимодействия с остальной частью программы. Ниже приведен пример нечистой функции, которая позволяет пользователям отправиться в котокафе (это место, где вы можете попить кофе и погладить котов):

```
public class World {
    public void goToCatCafe(CatCafe catCafe) {
        System.out.println("Petting cats at a Cat Café!");
        catCafe.arrive();
        catCafe.haveFun();
    }
}
```

Чистые функции обычно проще тестировать, т. к. при передаче одинаковых входных значений будут получены одинаковые результаты, а это позволяет легко протестировать вход и выход при помощи стандартных процедур юнит-тестирования. Нечистые функции являются более сложными, т. к. у вас может не оказаться выходного значения, к которому можно применить утверждения. Кроме того, они могут зависеть от частей кода вне этого конкретного метода или модифицировать их. Приведем пример нечистого метода, который сложно протестировать, т. к. его зависимости и выходные данные не локализованы. В следующем коде все переменные с префиксом `_global` определены и установлены как внешние для метода:

```
public void printAndSave(CatCafe catCafe) {
    String valuesToPrint = DatabaseConnector.getValues(_globalUserId);
    valuesToSave = ValuesModifier.modify(valuesToPrint);
    writeToFile(_globalLogFileName, valuesToSave);
    printToScreen(_globalScreenSettings, valuesToPrint);
}
```

Сравните это с функцией квадратного корня, где мы точно знали, что именно мы передаем. Так как переданное значение является единственным значением, к которому у функции есть доступ, и единственное значение, которое нас интересует, — это возвращаемое значение, проверить результаты конкретного расчета довольно легко. Но в методе `printAndSave()` присутствуют зависимости от нескольких внешних переменных. При первичном запуске кода все может работать замечательно; при изменении значений переменных и последующем запуске все может завер-

шиться неуспешно. Сложно понять, где нужно осуществлять проверку ожидаемого поведения. Похоже, код что-то записывает в файл, но нам нужно разобраться с его именем и расположением, которые зависят от значения переменной в определенный момент времени. Нам также надо понять, что за переменная `_globalUserId`, для того, чтобы определить, какие значения будут правильными и как будет все отображаться на экране в зависимости от значения переменной `_globalScreenSetting`. В любом из этих случаев на значения переменных способно повлиять множество внешних факторов, и выходные значения зависят о того, над чем у нас может не оказаться прямого контроля. В результате все это делает тестирование нечистых методов более чем сложной задачей.

Это не означает, что нечистые функции плохие! Как мы увидели, они крайне необходимы, если вы хотите сделать что-то помимо нагрева процессора. В конце концов, вывод чего-либо на экран с технической точки зрения является побочным эффектом. Но поддерживая как можно большее количество функций чистыми и ограничивая нечистые функции определенными областями, вы делаете тестирование системы более легким. Вы можете представить этот процесс как "карантин" нечистых функций для того, чтобы понимать, где могут возникнуть трудности в тестировании.

13.5. Создание тест-раннера

Предполагая, что у нас загружены правильные jar-файлы, можно вручную скомпилировать отдельные классы юнит-тестов, как показано в приведенной ниже команде. Обратите внимание, что версии и расположение jar-файлов в вашей системе могут отличаться.

Компиляция:

```
javac -cp .:./hamcrest-core-1.3.jar:./junit-4.12.jar FooTest.java
```

Отметим, что эта команда работает в операционных системах OS X и UNIX. Для работы в Windows нужно заменить `:` на `;` (`java -cp .;./junit-4.12.jar;./hamcrest-core-1.3.jar TestRunner`). Если вы используете Windows 7, вам также потребуется поместить аргумент `classpath` в кавычки (`java -cp ".;./junit-4.12.jar;./hamcrest-core-1.3.jar" TestRunner`). Не используйте `~` или другие сокращения при ссылке на путь, который указывает на расположение файлов junit и hamcrest. Ваши Java-файлы могу скомпилироваться, но не запуститься, т. к. в `javac` опция `-cp` обрабатывает пути иначе, чем в `java`.

Затем вы можете добавить свой метод `public static void main` в каждый отдельный класс для запуска каждого отдельного теста и определить тестовые методы, которые надо запускать.

Но вы можете представить, что это станет очень утомительным по мере того, как мы будем добавлять новые тесты и новые методы `public static void main` в каждый из этих файлов для запуска всех отдельных юнит-тестов. Оказывается, лучшим решением станет создание **тест-раннера**. Тест-раннер настроит окружение и выпол-

нит набор юнит-тестов. Если вы используете систему сборок или среду разработки, этот тест-раннер обычно создается и обновляется автоматически. Тем не менее я считаю полезным показать, как создать собственный тест-раннер, т. к. от вас может потребоваться создание какого-то особого, или вы можете работать над системой, при разработке которой нет среды разработки или задействуется инструмент сборок, не поддерживающий автоматический запуск тестов.

Приведем пример простой программы тест-раннера, которая будет выполнять все методы с аннотацией `@Test` в любом из заданных классов. Если произойдет сбой, то на экране будет отображена информация о проблемном тесте; в ином случае программа даст пользователю знать, что все тесты прошли успешно.

```
import java.util.ArrayList;

import org.junit.runner.*;
import org.junit.runner.notification.*;

public class TestRunner {
    public static void main(String[] args) {
        ArrayList<Class> classesToTest = new ArrayList<Class>();
        boolean anyFailures = false;

        // Добавьте любые классы, которые вы хотите протестировать
        classesToTest.add(FooTest. class);

        // Провести все добавленные классы через цикл
        // и использовать JUnit для их запуска
        for (Class c: classesToTest) {
            Result r = JUnitCore.runClasses(c);

            // Вывести информацию в случае проблем с этим классом
            for (Failure f : r.getFailures()) {
                System.out.println(f.toString());
            }

            // Если r неуспешная, то был по крайней мере один отказ.
            // Таким образом установим anyFailures в true - и ее нельзя
            // изменить обратно на false (никакое количество успешных
            // тестов не перекроет тот факт, что один из тестов оказался
            // неуспешным)
            if (!r.wasSuccessful()) {
                anyFailures = true;
            }
        }
        // После завершения проинформировать пользователя о том, все
        // ли тесты прошли или же какие-то из них оказались неуспешными
        if (anyFailures) {
            System.out.println("\n!!! - По крайней мере одна неудача, см. выше");
```

```
        } else {
            System.out.println("\nВСЕ ТЕСТЫ ПРОШЛИ");
        }
    }
}
```

Этот простой тест-раннер выполнит все тесты в любых классах, добавленных в список `classesToTest`. Если вы хотите добавить дополнительные тестовые классы, просто следуйте шаблону выше и добавьте их в список классов для тестирования. Затем вы можете скомпилировать и выполнить ваш тестовый набор с использованием следующих команд:

```
javac -cp .:./hamcrest-core-1.3.jar:./junit-4.12.jar *.java
java -cp .:./hamcrest-core-1.3.jar:./junit-4.12.jar TestRunner
```

Глава 14

Продвинутое юнит-тестирование

Хотя вы можете добиться многого, используя полученные в предыдущей главе знания, существуют аспекты юнит-тестирования, которые невозможно реализовать с помощью этих техник. В этой главе мы продвинемся дальше и изучим, как создавать более сложные тест-кейсы.

14.1. Тестовые двойники

Юнит-тест должен быть локализованным тестом, т. е. он должен проверять конкретный тестируемый метод или функцию и не тревожиться о других аспектах системы. Если тест завершается неуспешно, мы хотим быть уверены, что проблема находится в коде этого метода, а не в чем-то, от чего зависит этот метод. Программное обеспечение часто взаимосвязано, и конкретный метод, который опирается на другие методы или классы, может работать некорректно, если эти части кода не работают корректно.

В следующем методе мы позабавимся с утиным прудом. Вызов `.haveFunAtDuckPond()` с `Duck d` будет кормить утку столько раз, сколько указано в переменной `numFeedings`. Затем метод вернет объем удовольствия, прямо пропорциональный тому, сколько раз покормили утку. Утка будет крякать каждый раз при кормлении. Обратите внимание, что мы кормим нашу утку натуральным кормом для уток. Не кормите уток хлебом, это вредно для них! Если в качестве параметра передана утка-null или же количество кормлений равняется нулю или меньше, то возвращается 0 как объем удовольствия (отсутствие уток и отрицательное кормление не являются удовольствием). Давайте также предположим, что реализация Duck дефектная и вызов метода `.quack()` приведет к исключению `QuackingException`:

```
public int haveFunAtDuckPond(Duck d, int numFeedings) {
    if (d == null || numFeedings <= 0) { return 0; }
    int amountOfFun = 0;
    for (int j=0; j < numFeedings; j++) {
        amountOfFun++;
        d.feed();
        d.quack();
    }
```

```
    return amountOfFun;
}
```

Хотя код этого метода отлично работает для всех входных данных, он требует наличия существующей утки. В противном случае любое неотрицательное число кормлений приведет к выбросу исключения. Если мы видим неуспешное завершение юнит-теста для какого-то конкретного метода, мы, естественно, думаем, что проблема в этом методе. Только после его изучения мы поймем, что проблема находится где-то в другом месте. Как мы можем протестировать код, который зависит от неработающего кода?

Я бы не задавал этот вопрос, если бы у меня не было ответа — **тестовые двойники** (test doubles). Тестовые двойники являются фейковыми объектами, которые вы можете использовать в ваших тестах, чтобы "заменить" другие объекты в кодовой базе. У этого подхода имеется множество преимуществ помимо сокрытия частей кода, которые не работают. Тестовые двойники также позволят вам локализовать причину ошибок. Если наши тесты для `haveFunAtDuckPond()` завершатся неуспешно, тогда проблема лежит только в этом методе, а не в классах или методах, от которых зависит этот тест.

JUnit напрямую не поддерживает тестовых двойников, но вам помогут библиотеки, которые вы можете установить совместно с JUnit. Для следующих нескольких разделов мы будем применять библиотеку под названием Mockito, которая позволяет использовать тестовые двойники, моки, верификации и заглушки. Я знаю, что мы еще не рассматривали эти термины, но разве не здорово знать, что нас ждет впереди?!

Для того чтобы использовать Mockito, вам надо убедиться, что jar-файлы mockito и objensis указаны в переменной `classpath` вместе с jar-файлами JUnit. Если вы используете Makefile (набор инструкций по компиляции) из предыдущей главы и скачали соответствующие jar-файлы, все должно заработать автоматически. Если вы используете среду разработки или инструмент для сборки (такой как Gradle или Maven), то, возможно, всё уже настроено. В случае если вы не используете тест-раннер из предыдущей главы, вам следует рассмотреть документацию вашего инструмента сборки, чтобы убедиться, что вы все настроили правильно.

Далее вам необходимо добавить строчку `import static org.mockito.*` в ваши файлы тестовых классов для получения доступа ко всем инструментам библиотек. Напомню, что использование `import static` вместо `import` позволит нам обращаться к статическим членам классов mockito без явной ссылки на них.

Ниже приведен пример тестового двойника в JUnit и Mockito для тестирования метода, который зависит от объекта — тестового двойника. Обратите внимание, что Mockito называет всех тестовых двойников "моками", даже если они не используют возможности мок-объекта (рассматриваемого ниже в этой главе):

```
// Тестируемый класс
public class Horse {
    public int leadTo(Water w) {
        w.drink();
```

```
        return 1;
    }
}
// Юнит-тест для класса
import static org.junit.Assert.*;
import static org.mockito.*;
import org.junit.*;
public class HorseTest {
    // Тест, что отправка лошади на водопой вернет 1
    @Test
    public void testWaterDrinkReturnVal() {
        Horse horse = new Horse();
        // Мы создаем тестового двойника для воды
        Water mockWater = mock(Water.class);
        int returnVal = horse.leadTo(mockWater);
        assertEquals(1, returnVal);
    }
}
```

Мы создали "фейковый" объект вместо передачи реального экземпляра класса `Water`. `Water` находится в "карантине" и не может стать причиной ошибки. Если тест завершится неуспешно, то причиной этого будет тестируемый метод. Следовательно, когда происходит сбой, мы будем точно знать, где в коде искать проблему. Однако всякий раз, когда вы используете двойников, вы также зависите от предположений о том, как код, от которого вы зависите, должен работать в реальной программе. Ничто не остановит вас, скажем, от создания тестовых двойников, имеющих метод, которого на самом деле в классе нет. В этом случае ваши тесты будут прекрасно работать, а программа — нет.

Двойники также могут использоваться для ускорения выполнения тестов. Представьте объект `Database`, который записывает информацию в базу данных и возвращает код состояния. При обычных условиях программе требуется запись на диск и, возможно, даже доступ к сети. Допустим, это занимает одну секунду. Может показаться, что это немного, но умножьте это время на количество всех тестов, имеющих доступ к базе данных. Даже у сравнительно небольшой программы могут быть сотни и даже тысячи юнит-тестов, добавляющих минуты или часы ко времени каждого тестового прогона.

Вы никогда, запомните — никогда, не должны использовать двойника для текущего тестируемого класса! В противном случае окажется, что вы больше ничего не тестируете, т. к. создаете фейковый объект для того, что вы тестируете.

Тестовые двойники необходимо использовать как можно чаще, когда в коде помимо тестируемого класса используется иной класс. Иногда это сложно реализовать. Для того чтобы минимизировать проблемы, вам следует при любой возможности передавать объект в качестве параметра вместо того, чтобы полагаться на переменные члена или глобальные переменные. Еще более сложными и более распространенными являются методы, которые генерируют и используют объекты внутри

себя. Зачастую невозможно использовать тестовых двойников для таких методов. Рассмотрим пример:

```
public class Dog {
    DogFood _df = null;
    DogDish _dd = null;
    DogWater _dw = null;

    public void setUpDogStuff() {
        _dd = new DogDish();
        _df = new DogFood();
        _dw = new DogWater();
    }
    public int eatDinner() {
        _df.eat();
        return 1;
    }
}
```

Если бы мы писали тест для этого класса, у нас не нашлось бы способа создания двойников объектов, являющихся внутренними для класса! Даже если бы мы переработали метод `setUpDogStuff()` для приема параметров `DogDish`, `DogFood` и `DogWater`, нам бы пришлось работать с дополнительными параметрами, в то время как нас интересует только `DogFood`.

Давайте немного модифицируем этот метод, чтобы сделать его более гибким для создания тестовых двойников:

```
public class Dog {
    DogFood _df = null;
    DogDish _dd = null;
    DogWater _dw = null;

    public void setDogFood(DogFood df) {
        _df = df;
    }
    public void setDogDish(DogDish dd) {
        _dd = dd;
    }
    public void setDogWater(DogWater dw) {
        _dw = dw;
    }
    public int eatDinner() {
        _df.eat();
        return 1;
    }
}
```

Если бы мы хотели протестировать этот код, нам бы потребовалось создать объект и установить значение отдельно от реального исполнения теста:

```
public class DogTest {
    @Test
    public void eatDinnerTest() {
    Dog d = new Dog();
        d.setDogFood(mock(DogFood.class));
        int returnVal = d.eatDinner();
        assertEquals(1, returnVal);
    }
}
```

Уже лучше, но все еще не идеально, т. к. для настройки теста все еще требуются дополнительные операторы. Но если мы просто передадим `DogFood` методу в качестве параметра, как здесь:

```
public class Dog {
    public int eatDinner(DogFood df) {
        df.eat();
        return 1;
    }
}
```

тест будет выглядеть так:

```
public class DogTest {
    @Test
    public void testEatDinner() {
        Dog d = new Dog();
        int returnVal = d.eatDinner(mock(DogFood.class));
        assertEquals(1, returnVal);
    }
}
```

Это позволит нам значительно упростить и сфокусировать тестирование. Вы также можете заметить, что помимо улучшения тестируемости код обладает рядом других преимуществ. Его легче читать и понимать, он короче, а появление ошибок менее вероятно. Работая с исходной версией, программист легко может в какой-то момент забыть установить переменную `_df` для объекта `DogFood`, что приведет к выбросу исключения нулевого указателя при попытке собаки поесть. Это по-прежнему возможно, но уже менее вероятно, если вы будете передавать объект непосредственно методу. Мы обсудим преимущества написания тестируемого кода — и почему тестируемый код является хорошим кодом, и наоборот, — в следующей главе.

14.2. Заглушки

Если тестовые двойники являются фейковыми объектами, то заглушки (stubs) являются фейковыми методами. В приведенных выше примерах мы не беспокоились о том, что делает метод `.eat()` объекта `DogFood`; мы просто не хотели его вызывать с реальным объектом `DogFood`. Впрочем, во многих случаях мы ожидаем определен-

ного возвращаемого значения при вызове метода. Давайте модифицируем метод `.eat()` объекта `DogFood` так, чтобы он возвращал целое значение, показывающее, насколько вкусная собачья еда:

```
public class Dog {
    public int eatDinner(DogFood df) {
        int tastiness = df.eat();
        return tastiness;
    }
}
```

Если бы мы просто использовали `df` как обычный тестовый двойник, то не могли бы сказать, что вернет `df.eat()`. А именно, ответ зависит от тестового фреймворка — в некоторых случаях вернется значение по умолчанию, в некоторых будет вызван реальный объект, а в некоторых будет выброшено исключение. А это должно быть просто — вы не должны вызывать методы объекта-двойника, пока не создали заглушки для них. Вся суть создания тестового двойника в том, что у вас имеется определенный вами объект, и вам не нужно полагаться на внешние определения. Допустим, у двойника нашего объекта `DogFood` имеется (научно определенный) уровень вкусности, равный 13. Мы может определить, что когда мы вызываем метод `.eat()`, то необходимо просто вернуть значение 13. Мы создали **заглушку** нашего метода:

```
public class DogTest {
    @Test
    public void testEatDinner() {
        Dog d = new Dog();
        DogFood mockedDogFood = mock(DogFood.class);
        when(mockedDogFood.eat()).thenReturn(13);
        int returnVal = d.eatDinner(mockedDogFood);
        assertEquals(13, returnVal);
    }
}
```

Теперь, когда у объекта `mockedDogFood` вызывается его метод `.eat()`, будет возвращено значение 13. И снова мы подвергли карантину другие методы путем создания их фейков, которые действуют так, как необходимо. В некоторых языках, но не в Java, мы можем даже создать заглушки методов, которые не существуют. Это позволяет нам тестировать не только классы, в которых имеются ошибки, но даже и те, для которых все методы еще не написаны.

14.3. Моки и верификация

"Да, да, это все прекрасно, — могу я услышать ваши слова, — но ты не ответил на главный вопрос! Мы все еще зависим от утверждения, что метод возвращает значение, и поэтому не можем протестировать не возвращающие значение методы!" Вспомним нечистый метод `goToCatCafe()`, который мы хотели протестировать в прошлой главе:

```
public class World {
    public void goToCatCafe(CatCafe catCafe) {
        System.out.println("Petting cats at a Cat Café!");
        catCafe.arrive();
        catCafe.haveFun();
    }
}
```

Здесь нет возвращаемого значения и поэтому нет ничего, что мы бы могли использовать в качестве утверждения. Единственный способ протестировать этот метод — убедиться, что методы `.arrive()` и `.haveFun()` вызывались для объекта `catCafe`. Мы можем сделать это с использованием специального тестового двойника, который называется **моком** (от англ. *mock* — имитация). Мок-объект позволит вам утверждать, что для него вызывался конкретный метод. В примере выше вместо того, чтобы утверждать, что возвращается конкретное значение (а оно никогда не возвращается), вы делаете "метаутверждения", что вызываются методы `.haveFun()` и `.arrive()`. Это называется **верификацией**, т. к. вы *верифицируете*, что вызывался метод. Обратите внимание, что этот вид верификации не имеет никакого отношения к той верификации, которая "проверяет правильность программного обеспечения". Это две различные концепции, которые существуют в одном мире.

Вот как мы можем протестировать этот метод с использованием верификации:

```
public void testGoToCatCafe(CatCafe catCafe) {
    CatCafe cc = mock(CatCaft.class);
    World w = new World();
    w.goToCatCafe(cc);
    verify(cc.arrive());
    verify(cc.haveFun());
}
```

Обратите внимание, что здесь нет традиционных утверждений (например, `assertEquals`). Утверждения "спрятаны" внутри вызовов `verify()`. Если верификация завершается неуспешно, тест-кейс также завершится неуспешно, как если бы использовалось простое утверждение.

Вам может понадобиться проверить, что вызов осуществляется определенное количество раз. Давайте предположим, что у нас есть метод `.multiPet()`, который позволяет вам погладить одного кота несколько раз в зависимости от переданного целочисленного значения.

```
public void multiPet(Cat c, int numTimes) {
    for (int j = 0; j < numTimes; j++) {
        c.pet();
    }
}
```

Используя метод `times()` в верификации, вы можете проверить, что если вызван метод `multiPet(c, 5)`, то кот `C` будет поглажен 5 раз, а если аргумент `numTimes` равняется 900, то кот будет поглажен 900 раз (счастливый кот!).

```
/**
 * Проверка, что если мы вызвали multiPet с валидным котом и
 * аргументом numTimes, равным 5, то кот будет поглажен 5 раз
 * /
@Test
public void testMultiPet5() {
    World w = new World();
    Cat c = mock(Cat.class);
    w.multiPet(c, 5);
    verify(c.pet(), times(5));
}
```

И наоборот, вы также можете проверить, что метод никогда не вызывается. Давайте представим в нашем классе `World` еще один связанный с `CatCafe` метод `.petCat()`.

```
public void petCat(Cat c, boolean gentle) {
    if (gentle) {
        c.pet();
    } else {
        // Ничего не делаем, т. к. не гладим котов,
        // не будучи нежными
    }
}
```

В этом случае мы хотим протестировать, что если переменная `gentle` равняется `true`, то будет вызван метод `.petCat()`, в противном случае вызов не произойдет. Мы можем создать два юнит-теста для каждого из этих классов эквивалентности.

```
/**
 * Проверка, что попытка погладить кота, будучи нежным, приведет
 * к поглаживанию кота один раз. Это осуществляется верификацией того,
 * что c.pet() вызван один раз.
 */

@Test
public void testPetCatGently() {
    World w = new World();
    Cat c = mock(Cat.class);
    w.petCat(c, true);
    verify(c.pet(), times(1));
}

/**
 * Проверка, что попытка погладить кота, не будучи нежным,
 * не приведет к поглаживанию кота. Это осуществляется
 * верификацией того, что c.pet() ни разу не вызывается.
 */
```

```
@Test
public void testPetCatNotGently() {
    World w = new World();
    Cat c = mock(Cat.class);
    w.petCat(c, false);
    verify(c.pet(), never());
}
```

Также можно заменить метод `never()` вызовом `times(0)`, поскольку они являются эквивалентными.

В итоге метод `verify` используется для создания утверждения в исполняемом коде. То есть, в отличие от традиционных утверждений, которые проверяют, что значение является правильным, вызов `verify` утверждает, что метод был вызван (или нет).

Обратите внимание, что мы не тестировали, работал ли вызов `System.out.println()`. Мы проделаем это в *разд. 14.6* далее в этой главе.

14.4. Фейки

Иногда нужен тест, который зависит от объекта и требует сложного поведения или невыполнения. В этом случае вы можете использовать **фейк** (fake). Фейк является особым видом тестового двойника, который действует подобно обычному объекту. Однако он создается как часть теста, и это означает, что он выполняется быстрее и проще. Например, вы можете удалить любую часть кода, которая записывает данные на диск (что всегда замедляет написание тестов). Вы можете выполнять простые расчеты вместо длительных сложных. Вы можете снизить зависимость объекта от других объектов.

Фейки требуют больше работы для создания тестового двойника, т. к. они являются "облегченной" версией объекта, а не просто определяют его внешнее поведение. Однако это позволяет выполнять с тестовыми двойниками больше сложных тестов, чем относительно простых. Предположим, что ваш класс `DuckPond` выглядит следующим образом:

```
public class DuckPond extends Pond {
    private int _funLevel = 0;
    public void haveFun() {
        _funLevel++;
    }
    public void haveUltraFun() {
        int funMultiplier = super.retrieveUltraLevelFromDatabase();
        _funLevel += funMultiplier * _funLevel;
    }
    public int getFunLevel() {
        return _funLevel;
    }
}
```

Давайте оставим в стороне любые проблемы, связанные с качеством этого кода и его исправлением с целью сделать его более тестируемым. В нашем случае вызов `haveUltraFun()` требует запроса к базе данных, который выполняется из метода суперкласса `DuckPond`. Однако этот метод также изменяет переменную `_funLevel` в зависимости от значения, полученного после обращения к `retrieveUltraLevelFromDatabase()`. Значение переменной `_funLevel` будет зависеть и от обращения к базе данных, и от того, сколько раз вызывались `haveFun()` и `haveUltraFun()`. И хотя можно просто создать тестовый двойник, который будет возвращать определенные значения, добавление такого поведения для теста, вызывающего несколько методов объекта `DuckPond`, может потребовать много дополнительной работы. Хуже того, что, вероятно, эту работу придется повторять во множестве тестов.

Использование `DuckPond` "как есть" также означает, что каждое обращение к `haveUltraFun()` приведет к значительному замедлению тестов. Помните, что обращения к диску или сети не приветствуются в юнит-тестировании, т. к. они требуют на порядок или больше времени, чем сам тест.

Для того чтобы решить проблему с производительностью, давайте создадим фейковую версию объекта, которую мы позже сможем использовать в наших тестах. Эта фейковая версия будет "обрезанной" версией `DuckPond`, но будет повторять ее общее поведение.

```
public class FakeDuckPond extends Pond {
    private int _funLevel = 0;
    public void haveFun() {
        _funLevel++;
    }
    public void haveUltraFun() {
        // ЗДЕСЬ БЫЛО ОБРАЩЕНИЕ К БАЗЕ ДАННЫХ
        _funLevel += 5 * _funLevel;
    }
    public int getFunLevel() {
        return _funLevel;
    }
}
```

Теперь мы предполагаем, что значение возвращаемой из базы данных переменной `funMultiplier` всегда равняется 5. Это одновременно ускорит тесты (теперь нет чтения базы данных) и упростит расчеты (и наше понимание того, каким должно быть ожидаемое поведение). Однако, в отличие от традиционного тестового двойника или мока, мы не должны определять, каким должно быть внешнее поведение. Класс сам по себе определит (упрощенное) поведение.

14.5. *setup() и tearDown()*

Зачастую существует что-то, что должно запускаться в начале теста определенного класса, и что-то, что должно запускаться в его завершение. Например, вы можете настроить мокированное соединение с базой данных и использовать это как преду-

словие для всех тестов вместо того, чтобы повторять один и тот же код во всех тестах. Для таких случаев JUnit предоставляет аннотации для создания методов `setUp()` и `tearDown()`. А именно, вы можете добавить аннотацию `@Before` для любых методов, которые должны запускаться перед каждым конкретным тест-кейсом, и аннотацию `@After` для методов, которые должны запускаться после. Хотя вы можете назвать эти методы по-другому, а не `setUp()` и `tearDown()`, именно эти названия являются общими, и встречать их вы будете довольно часто.

Ниже приведен пример использования методов `setUp()` и `tearDown()`:

```
public class BirdTest {
    DatabaseConnection _dbc = null;

    // Настраиваем мокированное соединение с базой данных
    @Before
    public void setUp() throws Exception {
        mockedDbc = mock(DatabaseConnection.class);
        _dbc = setupFakeDbConnection(mockedDbc);
    }

    // Закрываем мокированное соединение с базой данных
    @After
    public void tearDown() throws Exception {
        _dbc = null;
    }

    // Тесты, проверяющие, что у только что созданной птицы
    // уровень пушистости по умолчанию равняется 1
    @Test
    public void testFluffyBird() {
        Bird b = new Bird(_dbc);
        assertEquals(1, b.fluffinessLevel);
    }

    // Тесты, проверяющие, что только что созданная птица красивая
    @Test
    public void testPrettyBird() {
        Bird b = new Bird(_dbc);
        assertTrue(b.isPretty());
    }
}
```

Обратите внимание, что методы `@Before` и `@After` вызываются перед *каждым* тест-кейсом, а не единожды перед всеми тест-кейсами и единожды после всех их. В примере выше `setUp()` будет вызван дважды и `tearDown()` также будет вызван дважды.

Обратите также внимание, что хотя ничто не останавливает вас от создания множества методов с аннотациями `@Before` и `@After`, обычно в этом нет необходимости, а

код оказывается более сложным. Если вы разместите несколько методов с этими аннотациями, они будут запускаться в детерминированном порядке (т. е. если вы осуществите новый запуск, они будут запускаться в том же порядке). Однако поскольку тесты не должны зависеть друг от друга, этот порядок не определен и может изменяться всякий раз, когда вы обновляете свой код и/или версию JUnit. Таким образом, лучше всего иметь максимум по одному методу `@Before` и `@After` на тест.

Вернемся к процедурам настройки (setup) и демонтажа (teardown), если они у вас довольно сложные, то использования множества аннотаций обычно не требуется. Вместо аннотирования множества методов вы можете использовать один метод, но обращаться из него к другим методам-помощникам.

14.6. Тестирование системного вывода

Конкретный вариант использования, в котором вам помогут аннотации `@After` и `@Before`, — проверка системного вывода. Консольный вывод требуется проверять довольно часто, но его тестирование в Java является неинтуитивным. Хотя возможно передавать объект `System` с каждым методом, которые затем мокировать и стабировать, это не идиоматический код Java, и подобное решение добавит много дополнительного кода (и, скорее всего, сложности) в вашу кодовую базу. Лучшим решением, которое я видел, оказалось использование метода `setOut()` класса `System`, чтобы поместить вывод `System.out` и `System.err` в `BytyArrayOutputStream`[1].

Ниже приведен пример того, как с использованием этой техники проверить определенные выходные данные из программы Java:

```
public class Kangaroo {
    public void hop() {
        System.out.println("Hoppity hop!");
    }
}
public class KangarooTest {
    private ByteArrayOutputStream out = new ByteArrayOutputStream();

    @Before
    public void setUp() {
        System.setOut(new PrintStream(out));
    }

    @After
    public void tearDown() {
        System.setOut(null);
    }
```

[1] Полную информацию можно найти здесь: **http://stackoverflow.com/q/1119385** (да, даже авторы книг иногда ищут в онлайне ответы на вопросы).

```
    @Test
    public void testHop() {
        Kangaroo k = new Kangaroo();
        k.hop();
        assertEquals("Hoppity hop!", out.toString());
    }
}
```

14.7. Тестирование *private*-методов

Ведутся серьезные споры по поводу того, имеет ли смысл тестировать закрытые `private`-методы или нет. Задать такой вопрос — это отличный способ начать жаркие споры между разработчиками и тестировщиками в вашей любимой социальной сети. Я приведу вам аргументы каждой стороны, так что вы сможете принять решение самостоятельно, а затем я скажу вам свое мнение.

Те, кто утверждает, что закрытые `private`-методы *никогда* не должны тестироваться, объясняют это тем, что любые вызовы из остальной части программы (т. е. из остального мира с точки зрения класса) должны поступать через открытые `public`-методы класса. Эти `public`-методы сами по себе имеют доступ к `private`-методам; а если нет, то какой смысл в этих `private`-методах? Тестируя только открытые (public) интерфейсы класса, вы минимизируете число тестов, которые есть у вас, и концентрируетесь на тестах и коде, которые имеют значение.

Другой причиной для отказа от тестирования `private`-методов является то, что оно препятствует последующему рефакторингу кода. Если вы уже написали тесты для закрытых методов, придется поработать, чтобы внести изменения во что-либо, находящееся "за сценой". В некотором смысле тестирование закрытых методов означает идти вразрез с одним из ключевых принципов объектно-ориентированного программирования, а именно с сокрытием данных. Система окажется менее гибкой и более сложной для будущих изменений.

Те, кто утверждает, что `private`-методы *всегда* должны тестироваться, указывают на то, что эти `private`-методы все равно являются кодом, даже если их не вызывают извне класса. Предполагается, что юнит-тестирование тестирует функциональность на самом нижнем из возможных уровней, и обычно это вызов метода или функции. Если вы собираетесь тестировать на более высоком уровне абстракции, то почему бы не провести тестирование на системном уровне?

Мое мнение таково, что, как и в большинстве технических вопросов, правильный ответ зависит от того, что вы пытаетесь сделать и что собой представляет кодовая база. Отмечу, что в большинстве технических вопросов сказать, что "это зависит от ситуации", является отличным способом оказаться правым, несмотря ни на что. Давайте рассмотрим несколько примеров.

Представим, что мы добавляем службу в Rent-A-Cat, чтобы автоматизировать фотографирование котов:

```
public class Picture {
    private void setupCamera() {
        // ...
    }
    private void turnOffCamera() {
       // ...
    }
    private Image captureImage() {
        // ...
    }
    public Image takePicture() {
        setupCamera();
        Image i = captureImage();
        turnOffCamera();
        return i;
    }
}
```

Это относительно простой код, и легко увидеть, что все ваши `private`-методы будут вызваны и протестированы `public`-методами. С точки зрения пользователя этого класса, легко проверить, что все работает правильно — если возвращается валидное изображение, то метод работает.

Теперь давайте представим, что в другой части этой службы имеется библиотека изменения изображения, которую мы также хотим протестировать:

```
public class ImageLibrary {
    public Image transform(Image image, int inSize, int outSize,
      String format, boolean color, boolean reduce, boolean dither) {
        if (inSize < outSize && format.equals("jpg") || dither == false) {
            return privateMethod1(image, inSize, outSize);
        } else if (inSize < outSize && format.equals("png") || dither == true) {
            return privateMethod2(image, inSize, outSize);
        } else if (outSize > inSize || (color == false && reduce == false) {
            return privateMethod3(image, inSize, outSize);
        } else {
            // Здесь может быть еще больше условий
            // if...then...else if
        }
    }
    // Здесь располагается множество private-методов
}
```

Оцените множество сложных тестов, которые понадобятся всего для одного этого метода! Но даже в этом случае вы не фокусируетесь на реальных изменениях изображения. Многие из ваших тестов будут предназначены для того, чтобы убедиться, что был вызван нужный метод!

Кто-то может возразить, что это недостаточно продуманный код и его надо переработать, в идеале с размещением `private`-методов в классе, скажем, `ImageTransformer`,

где методы могут быть `public` и легко проверены юнит-тестированием. Не могу не согласиться с этим. Однако в том-то и дело, что в реальном мире часто встречается подобный код и тестировщик не всегда может сказать руководству, что компании надо потратить несколько месяцев на решение технических проблем вместо того, чтобы добавлять новые возможности. Если вашей целью является тестирование программного обеспечения, и тестирование качественное, вам, возможно, придется иногда тестировать `private`-методы. Для более подробной информации о том, как это делается в Java, обратитесь к *главе 24*.

14.8. Структура юнит-теста

14.8.1. Основной план

Юнит-тесты в Java обычно группируются по классам и далее по методам; они отображают структуру программы. Так как юнит-тесты являются тестами белого ящика, которые тесно работают с кодом, нахождение ошибок в кодовой базе на основе конкретного неуспешного теста оказывается гораздо более простым, чем в интеграционных или ручных тестах.

14.8.2. Что тестировать?

Что именно тестировать, будет зависеть от области использования тестируемого ПО и количества времени для тестирования, а также от стандартов организации и прочих внешних факторов. Конечно, это утверждение не укажет вам направление движения, и подобные предостережения, наверное, могут быть помещены перед каждым абзацем этой книги. Существуют некоторые эвристические методы, которым можно следовать и которые напрямую отражают какие-то из вопросов, обсуждаемых при разработке тест-плана.

В идеале вам нужно посмотреть на метод и подумать о различных успешных и неуспешных случаях, определить классы эквивалентности, продумать, какие хорошие граничные и внутренние значения можно протестировать при помощи этих классов эквивалентности. Вы можете сосредоточиться на тестировании наиболее распространенных случаев, а за редко используемые взяться позже, по крайней мере, на первых порах. Если вы создаете критически важное для безопасности программное обеспечение, часто имеет смысл взяться за тестирование отказов, прежде чем проверять "счастливые пути". Лично я часто сперва работаю с базовым сценарием, а затем думаю о возможных случаях отказа. Зачастую я возвращаюсь обратно, иногда с профайлером, чтобы увидеть, какой код выполняется наиболее часто, и добавить для него дополнительные тест-кейсы. Я могу попробовать создать мысленную модель того, что вызывается часто вместо использования профайлера. Я определенно подумаю, откуда поступают входные данные в методы. Если они из системы, над которой у меня нет контроля (например, пользователи — лучший пример систем, над которыми у меня нет контроля), и значения оказываются неожиданными, я определенно потрачу больше времени на размышления о возможных случаях отказа и проверке граничных значений.

Не надо создавать набор тестов, который выполняется так долго, что люди не будут запускать его часто, но хорошо спроектированный набор юнит-тестов с соответствующими двойниками, моками, заглушками и т. п. должен работать очень быстро, даже если в нем много тестов. Сначала я бы ошибочно решил, что лучше создавать как можно больше тестов. Но после определения, сколько тестов необходимо для конкретной части программы, вы можете начать находить компромисс между количеством времени на разработку и количеством времени на тестирование.

14.8.3. Утверждайте меньше, называйте прямо

Когда юнит-тест завершается неуспешно, он должен прямо показать, что пошло не так и где. Тест не должен быть всеохватывающим "один за всех". Правильно написанный юнит-тест выполняется быстро (редко больше нескольких десятков миллисекунд), так что дополнительное время выполнения и работа по написанию большего количества тестов окажутся незначительными по сравнению с тем временем, которое сэкономят для вас юнит-тесты, сообщив, что именно пошло не так. Юнит-тест со множеством утверждений показывает отсутствие мысли о том, что именно этот юнит-тест должен был проверить. Рассмотрим следующий пример:

```
public class CatTest {
    @Test
    public void testCatStuff() {
        Cat c = new Cat();
        c.setDefaults();
        assertTrue(c.isAGoodKitty());
        assertEquals(0, c.numKittens());
        assertFalse(c.isUgly());
        assertNull(c.owner());
    }
}
```

В чем окажется причина, если тест завершится неуспешно? Возможно, что недавно созданная кошка не является хорошим котенком, как следовало бы. Возможно, произошла ошибка с количеством котят у недавно созданной кошки. Возможно, что владелец кошки был назначен с ошибкой. Вам придется проверять утверждения в коде теста, потому что факт провала теста "cat stuff" не скажет вам ни о чем. Если имя завершившегося неуспешно теста не говорит вам, в чем причина отказа, и не указывает, где находится проблема, возможно, вы тестируете слишком много в каждом отдельном тест-кейсе. У меня неоднократно бывало, что я правил тест программы, основываясь только на названии неуспешного теста, даже не глядя на его код. Это является следствием хорошо определенных и проработанных тестов.

14.8.4. Юнит-тесты должны быть независимыми

Юнит-тесты не должны зависеть от порядка запуска. То есть тест 2 не должен зависеть от побочных эффектов или результата теста 1. JUnit и другие фреймворки тестирования не запускают отдельные тест-кейсы в предопределенном порядке. Избе-

гание зависимостей друг от друга и разрешение каждому тесту выполняться независимо локализует сбои конкретными тестами. Теперь представьте написание следующего кода:

```
public class Cat {
    private int _length;

    public Cat(int length) {
        _length = length;
    }
    public int getWhiskersLength() {
        return _length;
    }
    public int growWhiskers() {
        _length++;
        return _length;
    }
}

public class CatTest {
    int _whiskersLength = 5;

    @Test
    public void testWhiskersLength() {
        Cat c = new Cat(5);
        assertEquals(_whiskersLength, c.getWhiskersLength());
    }

    @Test
    public void testGrowWhiskers() {
        Cat c = new Cat(5);
        _whiskersLength = c.growWhiskers();
        assertEquals(6, _whiskersLength);
    }
}
```

Если мы запустим эти тесты с JUnit, которые выполняются в случайном порядке, иногда второй тест пройдет, а иногда завершится неуспешно. Почему?

Давайте предположим, что тесты выполняются в том порядке, в котором записаны, т. е. `testWhickersLength()` выполняется первым. Переменная `_whiskersLength`, которая по умолчанию равняется 5, будет соответствовать начальному значению, устанавливаемому при создании объекта `Cat`. То есть наше утверждение, что длина усов (whiskers length) равняется 5, будет правильным. Когда вызывается метод `testGrowWhiskers()`, длина усов увеличивается на единицу, и возвращаемое значение (т. е. новое значение усов) помещается в переменную `_whiskersLength`, изменяя ее значение на 6. Так как `_whiskersLength` равняется 6, утверждение также правильное. Поздравляем, все тесты прошли!

Теперь рассмотрим, что произойдет, когда тесты пойдут в обратном порядке и `testGrowWhiskers()` будет выполняться первым, а затем `testWhickersLength()`. В конце теста `testGrowWhiskers()` переменная `_whiskersLength` (которая является переменной уровня класса и поэтому используется всеми методами) равняется 6. Теперь выполняется `testWhickersLength()`, но длина усов нового объекта `Cat` равняется 5, и это значение не совпадает с `_whiskersLength`, равной 6. Теперь мы имеем падающий тест, но такой, который завершается неуспешно время от времени в зависимости от того, в каком порядке выполняется тест.

Мы можем исправить это, гарантируя, что тесты не зависят друг от друга. Самый легкий и лучший способ сделать это — устранить любые общие данные между тестами. В нашем случае это означает независимость от переменной уровня класса `_whiskersLength`. Давайте перепишем тесты так, чтобы они могли работать независимо.

```
public class CatTest {
    @Test
    public void testWhiskersLength() {
        Cat c = new Cat(5);
        assertEquals(5, c.getWhiskersLength());
    }

    @Test
    public void testGrowWhiskers() {
        Cat c = new Cat(5);
        int whiskersLength = c.growWhiskers();
        assertEquals(6, whiskersLength);
    }
}
```

Хотя для этого примера нашлось относительно простое решение, в других случаях может оказаться труднее найти зависимости или внести исправления. Каждый раз, когда вы находите тест, который проходит время от времени, может потребоваться поиск "спрятанных" зависимостей между тестами. Этими спрятанными зависимостями могут быть общие для методов объекты или переменные, статические переменные класса или другие внешние данные, на которые опираются ваши тесты.

Обратите внимание, что в JUnit *возможно* определить порядок запуска тестов путем использования аннотации `@FixMethodOrder`. Однако постарайтесь избегать ее использования. Тесты легко могут оказаться в ловушке зависимости друг от друга, если разрешить им запускаться в определенном порядке.

Существует еще одно преимущество создания тестов без зависимостей от других тестов. Независимые тесты могут запускаться параллельно с использованием различных ядер процессора или даже полностью на разных машинах. Если выполнение теста зависит от последовательности запуска, тогда вы не сможете запускать их параллельно. На современной многоядерной машине вы можете выполнять тесты во много раз быстрее, если они могут запускаться независимо. И хотя точное уско-

рение будет зависеть от конкретных запускаемых тестов, аппаратных свойств вашего компьютера и прочих факторов, вы легко можете увидеть снижение времени выполнения тестов на 50% и больше.

14.8.5. Старайтесь сделать тесты лучше каждый раз, когда вы их касаетесь

Зачастую приходится работать с устаревшим кодом, в том числе с кодом, который плохо написан или не имеет хорошего тестового покрытия. Может показаться заманчивым продолжить работать в том же ключе, в каком создавался код, но вы должны попытаться сделать всё возможное, чтобы улучшить тестирование кода. Поддерживайте ваш код легко тестируемым и при необходимости создавайте обертки вокруг кода, который сложно протестировать.

Дискуссия о написании тестируемого кода продолжится в *главе 16 "Написание тестируемого кода"* (а что еще можно ожидать в главе с таким названием?)

14.9. Покрытие кода

Покрытие кода скажет вам, какая часть кодовой базы реально выполняется при запуске тестового набора. Так как точное определение того, что означает "какая часть кодовой базы", может быть сложным, существуют различные виды покрытия кода. Простейшей формой покрытия кода является покрытие методов; в данном случае измеряется процент методов, которые вызываются тестами. Например, представим, что в классе `Turtle` есть два метода, `crawl()` и `eat()`:

```
public class Turtle {
    public void crawl() { ... }
    public void eat() { ... }
}
```

Если у нас есть тест, вызывающий `crawl()`, но нет вызывающих `eat()`, то значит, что у нас покрытие кода составляет 50% (был протестирован один из двух методов). Даже если бы у нас была сотня тестов, вызывающих `crawl()` и всевозможными путями проверяющих, как ползает черепаха, но ни один не тестировал бы `eat()`, то тестовое покрытие у нас все равно составило бы 50%. 100%-ное покрытие кода мы получим, только когда добавим хотя бы один тест, проверяющий `eat()`.

Более детальной формой покрытия кода является покрытие операторов. Оно измеряет процент программных операторов, которые были выполнены по крайней мере одним тестом. Программный оператор является наименьшим "кусочком" кода, который можно рассматривать как отдельную часть кода; в Java такие части обычно разделяются точкой с запятой. Можно привести примеры операторов:

1. `int j = 7;`
2. `System.out.println("Bark!");`
3. `foo--;`

Обратите внимание, что "оператор" не означает "строку кода"! Например, в Java у вас может быть такая строка:

```
doSomething(k); k++;
```

В данном случае мы имеем дело с несколькими операторами, которые расположены на одной строке.

Когда разработчики обсуждают "покрытие кода", они обычно имеют в виду именно "покрытие операторов". По сравнению с покрытием методов использование покрытия операторов предоставляет гораздо более проработанную информацию о том, какие части кодовой базы фактически были протестированы. Например, давайте добавим некоторые детали в наш класс `Turtle`, чтобы мы смогли увидеть, как выглядит код внутри различных методов:

```
public class Turtle {
    CurrentLocation _loc = World.getCurrentLocation();
    GroundType _g = World.getGroundType(_loc);

    public void crawl() {
        if (_g == DIRT) {
            move(SLOWLY);
        } else if (_g == GRASS) {
            eat();
            move(MORE_SLOWLY);
        } else {
            move(EVEN_MORE_SLOWLY);
        }
    }

    public void eat() {
        System.out.println("Yum yum");
    }
}
```

С использованием покрытия методов всего один тест для `eat()` или всего один тест для `crawl()` дадут 50% кодового покрытия. За этим остается незамеченным то, что `crawl()` намного сложнее, чем `eat()`, и любой одиночный тест не сможет проверить различные варианты выхода, в то время как `eat()` может быть хорошо протестирован всего одним тестом. Также мы не узнаем, какие конкретно строки не были протестированы, — нам придется исследовать код теста, чтобы определить, тестировалось ли то, что черепаха ползет по земле, траве или по чему-то еще. Результат покрытия операторов может точно сказать нам, какие строки никогда не выполнялись по время тестового прогона, и мы будем точно знать, какие виды тестов необходимо добавить, чтобы убедиться, что каждая строка была протестирована по крайней мере единожды.

Существуют другие варианты покрытия кода, среди которых покрытие ветвей, измеряющее, какой был протестирован процент условных выражений (условий `if`,

операторов `case` и т. п.). Однако эти виды покрытий кода обычно используются для более специализированного тестирования. Гораздо более вероятно, что вы будете часто сталкиваться с покрытиями операторов и методов.

Если у вас имеется оператор или метод, покрытый тестами, это не означает, что все дефекты этой части кода были обнаружены этими тестами! Легко представить дефекты, проскальзывающие через покрытие методов. В нашем примере с `Turtle` если бы возникли проблемы с черепахой на траве, а наш тест проверял случай, когда черепаха находится на земле, покрытие методов показало бы, что `crawl()` проверен, в то время как в нем по-прежнему могли бы прятаться дефекты. На более низком уровне абстракции, покрытие операторов не проверяет все варианты выполнения конкретного оператора. Давайте рассмотрим следующий класс с единственным методом и связанный с ним тест-кейс:

```
public class Cow {
    public int moo(int mooLevel) {
        int timesToMoo = Math.ceil(100 / mooLevel);
        for (int j=0; j < timesToMoo; j++) {
            System.out.println("moo!");
        }
        return timesToMoo;
    }
}
public class CowTest {
    @Test
    public void mooTest() {
        Cow c = new Cow();
        int mooTimes = c.moo(20);
        assertEquals(5, mooTimes);
    }
}
```

С точки зрения покрытия кода, у нас 100%-ное покрытие кода и 100%-ное покрытие методов — единственный метод класса вызывается из теста, и выполняется каждый оператор метода. Однако вызов метода `moo()` с переменной `mooLevel`, равной 0, вызовет выброс исключения `DivideByZeroException`. Этот дефект не будет обнаружен тест-кейсом, несмотря на то, что выполняются все операторы. Проверка всех классов эквивалентности не является "бронебойной защитой", но она поможет исправить подобные ситуации.

Конечно, метрики покрытия кода могут быть еще более обманчивыми, чем в этом примере. Как только оператор *выполняется* тестом, этот код считается "покрытым". Ничто не проверяет, что юнит-тесты действительно что-то проверяют. Рассмотрим следующий пример:

```
public class CowTest {
    @Test
    public void mooTest() {
        Cow c = new Cow();
```

```
        int mooTimes = c.moo(1);
        assertTrue(true);
    }
}
```

Этот тест приводит к 100%-ному покрытию кода, но при этом почти ничего не говорит вам о коде. Единственная информация, которую вы можете получить от прохождения теста, — это то, что вызов `c.moo(1)` не приводит к завершению работы программы.

Покрытие кода — мощный инструмент, но, как и все в разработке программного обеспечения, не является универсальным спасательным кругом. Это отличный способ увидеть, какие области кодовой базы нуждаются в дополнительном тестировании, но он не гарантирует, что любой покрытый код непременно свободен от дефектов. Он даже не гарантирует, что определенная часть кода действительно была протестирована.

Подобно вашим любимым питательным хлопьям, юнит-тестирование не должно становиться ни вашим полноценным завтраком, ни всем тест-планом. Юнит-тестирование прекрасно подходит для проверки отдельных методов и низкоуровневой функциональности, но оно не сильно поможет с пониманием того, как все сочетается. Более того, все усложняется при попытке определить, как должен выглядеть конечный продукт; все отдельные методы могут работать, но совместно они образуют нечто, что совсем не удовлетворяет требованиям.

При тестировании необходимо помнить о выполнении ручного тестирования, интеграционного тестирования и, в зависимости от ваших нужд, прочих видов тестирования, таких как тестирование безопасности или тестирование производительности. Полагаться на один конкретный вид тестирования — это лучший способ пропустить важные дефекты.

Глава 15

Разработка через тестирование

Хотя мы в общем рассмотрели написание юнит-тестов, остался вопрос: как мы интегрируем написание тестов в процесс разработки программного обеспечения? Раньше тестирование программ было полностью отдельным процессом от написания кода, но сегодня обеспечение качества программы также является работой программистов. И наоборот, тестирование ПО приобрело множество особенностей разработки; редкий тестировщик не написал ни строчки кода. Даже тестировщики, занимающиеся ручным тестированием, часто пишут интеграционные скрипты или нечто подобное.

Разработка через тестирование (test-driven development, TDD) является методологией для написания качественного программного обеспечения с хорошо продуманным набором тестов. Следуя принципам TDD, разработчики будут знать, что тестировать, в каком порядке и как сбалансировать юнит-тестирование и написание кода для тестируемого приложения. Конечно, это не панацея. В мире программ, как нам напоминает Фредерик Брукс (Frederick Brooks), не существует "серебряной пули", которая решит все ваши проблемы. Однако с правильным использованием TDD можно немного приручить оборотня разработки программного обеспечения.

15.1. Что такое разработка через тестирование?

Разработка через тестирование является методологией, которая состоит из нескольких ключевых составляющих.

1. *Сперва пишутся тесты, затем код.* Перед тем как вы начнете размышлять о том, *как* сделать что-то, вы подумаете о том, *что* нужно сделать. Поскольку ваш код еще не написан, вы можете проверить, что тест изначально завершается неуспешно (чтобы избежать тавтологических тестов, которые всегда проходят), и поставить перед собой конкретную цель, к которой вы будете двигаться (успешное завершение только что написанного теста).
2. *Написание только того кода, благодаря которому тест будет завершаться успешно.* Это гарантирует, что вы фокусируетесь на написании нужного кода вместо того, чтобы тратить время на разработку, возможно, излишнего фрейм-

ворка или другого кода. Одним из ключевых преимуществ разработки через тестирование является то, что она позволяет вам сосредоточиваться, обучая вас следить за целью текущего цикла, а не думать о всевозможных фрагментах кода, которые вы могли бы написать.

3. *Написание только тех тестов, которые тестируют код.* Заманчиво писать тесты только ради их написания; но это правило поможет вам заниматься тестами, которые написаны лишь для разработанного функционала.
4. *Короткий цикл оборачиваемости.* TDD выделяет быстрые циклы, благодаря которым разработчик придерживается правильного пути и сосредоточивается на короткой конкретной цели.
5. *Рефакторинг ранний и частый.* **Рефакторинг** — процесс изменения кода без изменений его внешней функциональности. Он может включать в себя как простые действия, вроде изменения имени переменной или добавления комментария, так и сложные, вплоть до изменения ключевой архитектуры или алгоритмов. Рефакторинг отличается от простого написания кода улучшением внутреннего качества кодовой базы без прямого воздействия на внешнее качество. И хотя рефакторинг не оказывает немедленного влияния на внешнее качество тестируемой системы, тем не менее зачастую он косвенно воздействует на нее. Это выражается в том, что код становится более легким для чтения, понимания, поддержки и модификации. По мере хода процесса разработки рефакторинг будет упрощать работу программистов.

Вот пример плохо написанной программы. Мы подвергнем ее рефакторингу для того, чтобы сделать ее более легкой для чтения без модификации ее существующего функционала.

```
public class Hours {
    public static void main(String[] args) {
        double chikChirik = 792.34;
        try {
            chikChirik = Double.parseDouble(args[40 / 3 - 13]);
        } catch (Exception ex) {
            System.exit(1 * 1 * 1);
        }
        int kukurigu = 160 % 100;
        int gruhGruh = (2 * 2 * 2 * 2 * 2 * 2) - 4;
        System.out.println((chikChirik * kukurigu * gruhGruh) + " seconds");
    }
}
```

Первое, на что мы обратим внимание, — здесь совсем нет комментариев. Это делает код сложным для чтения. Также использование болгарской ономатопеи (т. е. звуков животных) в качестве имен переменных, хотя и интересно с лингвистической точки зрения, не дает вам никакой информации о том, что эти переменные должны представлять (*chik-chirik* — это звук, который издают болгарские птицы, *kukurigu* — так кричат болгарские петухи, а болгарские свиньи хрюкают *gruh-gruh*). Также присутствуют ненужная настройка переменных и усложнение. Например,

переменным `kukurigu` и `gruhGruh` присвоены значения 60 после проведения расчетов. Значение по умолчанию переменной `chikChirik`, равное `792.34`, никогда не используется; зачем тогда установлено именно это значение? Все исключения перехватываются в той части кода, где устанавливается переменная `chikChirik`; мы же должны проверять все отдельные случаи сбоев. Некоторые из наших переменных ни разу не изменяются — они должны быть объявлены константами (в Java это переменные `final`). И в итоге имеется возможность вынести часть вычислений в отдельные методы вместо того, чтобы выполнять все в методе `main`.

Все эти проблемы могут быть исправлены без изменения поведения программы. Это и есть *рефакторинг*, а не исправление дефектов. Сейчас код работает сам по себе, как и задумано, просто дальнейший процесс разработки с ним не будет легким. Давайте исправим некоторые проблемы и посмотрим, как выглядит код после рефакторинга. Разработка программного обеспечения — очень сложный процесс, который часто напрягает умы даже лучших программистов, тестировщиков и менеджеров. За все, что мы можем сделать для уменьшения мыслительной нагрузки себя и будущих членов команды, нам, без сомнения, скажут спасибо!

```
public class Hours {
    final static int MINUTES_PER_HOUR = 60;
    final static int SECONDS_PER_MINUTE = 60;
    /**
      * Получив количество часов, необходимо вернуть количество
      * секунд, которые соответствуют этому количеству часов
    */
    public static double calculateSeconds(double hours) {
        return hours * MINUTES_PER_HOUR * SECONDS_PER_MINUTE;
    }
    /**
      * Если в качестве первого аргумента командной строки указать
      * количество часов, то будет выведено количество секунд в этом
      * количестве часов. Например, 1 час = 3600 секунд.
      * Дополнительные аргументы командной строки игнорируются
    */
    public static void main(String[] args) {
        double numHours = -1;
        try {
            numHours = Double.parseDouble(args[0]);
        } catch (NumberFormatException nfex) {
            // Переданный аргумент не может быть обработан
            System.exit(1);
        } catch (ArrayIndexOutOfBoundsException oobex) {
            // Аргумент не был передан
            System.exit(1);
        }
        System.out.println(calculateSeconds(numHours) + " seconds");
    }
}
```

Такой код гораздо легче читать и понимать, хотя его поведение осталось тем же, что и у исходного кода до рефакторинга. И в такой код гораздо легче вносить изменения. Предположим, мы хотим отображать сообщение об ошибке, если аргумент не может быть прочитан или проанализирован, а не просто прекращать работу программы. В коде после рефакторинга мы видим различные режимы сбоев и можем легко добавить любое подходящее сообщение об ошибке в зависимости от проблемы (т. е. "Аргумент не может быть обработан как double" или "Должен быть передан по крайней мере один аргумент"). Возможно, мы захотим модифицировать нашу программу для расчета французского республиканского календаря, в котором минута состояла из 100 секунд, а час — из 100 минут (всего в сутках было 10 часов). Довольно просто увидеть константы `SECONDS_PER_MINUTE` и `MINUTES_PER_HOUR` и установить соответствующие значения для них. Реализовать подобное в исходном коде было бы гораздо сложнее.

К сожалению, зачастую к рефакторингу прибегают, когда процесс разработки уже идет вовсю, и применить его оказывается труднее. Так бывает, если рефакторингом раньше не занимались или у команды разработки мало опыта в нем. Трудности, возникающие при рефакторинге, становятся поводом избегать его в дальнейшем. Это становится самоисполняющимся пророчеством; избегание рефакторинга кода делает в дальнейшем рефакторинг еще более сложным! Частый рефакторинг становится составляющей процесса разработки и привычкой, а не тем, чем можно заняться, "когда будет достаточно времени" (заметьте, что времени никогда не бывает достаточно).

15.2. Цикл "красный — зеленый — рефакторинг"

Мы работаем в рамках этих ограничений, которые накладывает цикл "красный — зеленый — рефакторинг". Одиночный цикл в TDD включает в себя три следующих шага.

1. *Красный.* TDD является формой **разработки "сперва тесты"** (test-first development, TFD), поэтому сначала необходимо написать тест. Разработчик пишет сбойный тест для нового функционала или для граничного случая, который необходимо проверить. Только что написанный тест — и только этот тест — должен завершиться неуспешно. Если этот только что написанный тест не завершается неуспешно, это означает, для данного функционала код уже написан. Если другие тесты завершаются сбоем, это означает, что существует проблема с тестовым набором — возможно, связанная с периодическим или недетерминированным сбоем теста — которую нужно исправить перед тем, как двигаться дальше. Эта фаза называется "красной", потому что многие фреймворки юнит-тестирования отображают неуспешные тесты красным. Так как красно-зеленый дальтонизм охватывает немалую часть человеческой популяции, а люди относятся к тем живым существам, которые, наиболее вероятно, займутся программированием, такой выбор цветов кажется не лучшим. Тем не менее мы будем придерживаться этого принятого соглашения по цветам.

2. *Зеленый.* Теперь разработчик пишет код для прохождения теста. Эта работа связана только с тем, чтобы тест завершался успешно, и не должна стать причиной сбоев других тестов. В данный момент возможно появление "уродливого" кода; задачей является заставить его работать, а не сделать красивым. Если другие тесты начинают сбоить, это значит, что разработчик ненароком вызвал регрессию и должен исправить это. В конце этой фазы все тесты должны проходить (быть "зелеными").
3. *Рефакторинг.* После того как все тесты завершились успешно, разработчик должен изучить только что написанный код и заняться его рефакторингом. Здесь возможны как небольшие проблемы, например с "магическими числами" (т. е. "голыми" значениям в программе, которые не описаны или не представлены в виде констант, например `if (mph > 65) { ... }` вместо `if (mph > SPEED_LIMIT) { ... }`), так и такие значительные, как плохо проработанный алгоритм. Все это может быть исправлено в данной фазе, т. к. в предыдущей фазе фокус был на получении правильного результата. Легко запомнить это поможет мнемоническая английская фраза "first make it green, then make it clean" (сперва зеленый, потом чистый). Так как всегда существует запасной вариант в виде правильно функционирующего (но плохо написанного) кода, разработчик может попробовать различные подходы, не беспокоясь о том, что код может совсем перестать работать. В самом худшем случае код можно вернуть к состоянию конца "зеленой" фазы и попробовать другой путь его модификации.

После каждого цикла "красный — зеленый — рефакторинг" разработчик может подумать о добавлении нового функционала и затем начать новый цикл. Такая цикличность будет повторяться до завершения работ над программой. Это путь тестирования, программирования и рефакторинга, который в конечном итоге приведет к готовому продукту. Его побочным эффектом станет основательный набор тестов, который имеет непосредственное отношение ко всем функциям программы.

Все это можно переписать в качестве очень простого алгоритма. Таким образом, мы увидим, как это помогает сосредоточить внимание работающего над программой человека; всегда существует четко определенный следующий шаг:

1. Написать тест для функциональности, которая еще не была создана.
2. Запустить тестовый набор — завершиться сбоем должны только новые тесты. Если это не так, сперва необходимо понять, почему завершились неуспешно другие тесты, и исправить эту проблему.
3. Написать достаточно кода, чтобы этот тест проходил, а другие тесты не начинали сбоить.
4. Запустить тестовый набор — если какой-либо тест завершится неуспешно, вернуться к шагу 3; если все тесты прошли, продолжать дальше.
5. Провести рефакторинг написанного кода и/или любого связанного кода.
6. Запустить тестовый набор — если какой-либо тест завершится сбоем, вернуться к шагу 5; если все тесты прошли, продолжать дальше.

7. Если необходимо добавить функциональность, перейти к шагу 1. Если функциональность добавлять не надо, приложение готово!

15.3. Принципы разработки через тестирование

При написании кода необходимо помнить о нескольких принципах разработки через тестирование.

- *YAGNI (You Ain't Gonna Need It — тебе это не понадобится).* Не пишите код, который вам не нужен для прохождения тестов! Всегда заманчиво создать красивую абстрактную систему, которая сможет в будущем обрабатывать все модификации того, чем вы занимаетесь сейчас, но тем самым вы сделаете код более сложным. Что еще хуже, вы усложните его таким образом, что в дальнейшем это не поможет развитию системы. Избегайте сложности, пока она на самом деле не окажется нужна. Если у вас есть граничные случаи или классы эквивалентности, с которыми надо разобраться, сперва добавьте больше тестов.
- *KISS (Keep It Simple, Stupid — не усложняй, тупица).* Одной из целей TDD является гарантирование, что кодовая база остается гибкой и расширяемой, и одним из главных врагов этих двух целей является сложность. Сложные системы трудно понять и поэтому модифицировать; сложные системы, как правило, подходят конкретным системам, для которых они были разработаны, и в них трудно добавлять новые возможности или функционал. Сохраняйте ваш код и дизайн простыми и сознательно избегайте добавления дополнительной сложности.
- *Fake It 'Til You Make It (Притворяйся, пока не сделаешь).* Вполне нормально задействовать фейковые методы и объекты в ваших тестах или использовать `return 0` в качестве заменителя тела метода. Вы можете вернуться к этим вопросам позже по мере необходимости с дополнительными тестами и кодом.
- *Avoid Slow Running Tests (Избегайте медленных тестов).* Если вы работаете с TDD, вы запускаете по крайней мере три полных тестовых прогона в рамках итераций цикла "красный — зеленый — рефакторинг". Это минимум, если предполагать, что ваш код не вызывает проблем с другими тестами и не имеет собственных дефектов. Если прогон вашего тестового набора занимает две или три секунды, это минимальная цена за высокое качество, которое предоставляет TDD; если же на прогон уходит несколько часов, как долго продержатся разработчики, прежде чем бросить все и заняться непосредственно программированием?
- *Remember That These Are Principles, Not Laws (Помните, что это принципы, а не законы).* Было бы контрпродуктивным полностью игнорировать то, что еще должно выполнить программное обеспечение в рамках следующих итераций цикла "красный — зеленый — рефакторинг". Иногда тест может быть медленным, но необходимым, или же создание полноценного метода окажется таким же простым, как и добавление его фейковой версии. Хотя необходимо стремиться следовать принципам TDD, я не знаю никого, кто никогда не нарушал бы ни

один из этих принципов (хотя, возможно, это больше относится к тем людям, с которыми я провожу время).

15.4. Пример: создание программы FizzBuzz с использованием разработки через тестирование

Для того чтобы понять, как работает TDD, давайте напишем простую программу FizzBuzz с его использованием. Напомним, что FizzBuzz работает следующим образом:

1. Печатает все числа от 1 до 100 с определенными исключениями.
2. Если число делится без остатка и на 3, и на 5, вместо числа должно быть напечатано слово "FizzBuzz".
3. Если число делится без остатка на 3, вместо числа должно быть напечатано слово "Fizz".
4. Если число делится без остатка на 5, вместо числа должно быть напечатано слово "Buzz".

Сперва давайте создадим "ходячий скелет" приложения. Предположим, что у нас уже установлен JUnit или подобный фреймворк тестирования, и у нас нет необходимости развертывать его; всё, что нам надо сделать, — это сгенерировать наш начальный класс `FizzBuzz`. Так как мы перебираем диапазон значений и принимаем решение по каждому из них, давайте создадим класс, у которого есть метод `main` и метод `fizzbuzzify`, который возвращает правильную строку для заданного значения.

Забегая вперед, мы знаем, что хотим пройтись по числам от 1 до 100. Следовательно, существуют четыре случая, которые нам хотелось бы протестировать:

1. Должно быть возвращено само число, если оно не делится без остатка на 3 и на 5.
2. Должна возвращаться строка "Fizz", если число делится без остатка на 3, но не делится без остатка на 5.
3. Должна возвращаться строка "Buzz", если число делится без остатка на 5, но не делится без остатка на 3.
4. Должна возвращаться строка "FizzBuzz", если число делится без остатка на 3 и на 5.

Нашему тестированию помогает то, что функция является чистой — ее возвращаемое значение полностью определяется входным параметром. У нее нет зависимостей от глобальных переменных, нет побочных эффектов в виде вывода и нет внешних зависимостей. Подача на вход "2" всегда вернет "2" и ничего больше, подача на вход "3" всегда вернет "Buzz" и ничего больше, и т. д.

```
public class FizzBuzz {
    private static String fizzbuzzify(int num) {
```

```
        return "";
    }
    public static void main(String args[]) {
    }
}
```

Теперь давайте добавим наш первый тест для первого случая. Первое число, которое не делится без остатка на 3 или на 5, — это 1, поэтому давайте использовать 1 как первое значение для тестирования нашего метода `fizzbuzzify()`:

```
public class FizzBuzzTest {
    @Test
    public void test1Returns1() {
        String returnedVal = FizzBuzz.fizzbuzzify(1);
        assertEquals("1", returnedVal);
    }
}
```

Если мы запустим его, он завершится сбоем — `fizzbuzzify` возвращает пустую строку, которая не равняется 1, и таким образом утверждение оказывается неверным. Отлично, это можно легко поправить!

```
public class FizzBuzz {
    private static String fizzbuzzify(int num) {
        return "1";
    }
    public static void main(String args[]) {
    }
}
```

Теперь, когда мы запустим тест, он завершится успешно! Давайте перейдем к следующей фазе и поищем возможности для рефакторинга. В данном случае я не думаю, что они есть; конечно, здесь есть "магическое число" (хорошо, технически — магическая строка, представляющая число), но что можно сделать? Заменить его константой `NUMBER_ONE`? Понятнее от этого не станет.

Давайте добавим второй тест, для 2, который должен вернуть не `Fuzz` и `Buzz`, а "2":

```
public class FizzBuzzTest {
    @Test
    public void test1Returns1() {
        String returnedVal = FizzBuzz.fizzbuzzify(1);
        assertEquals("1", returnedVal);
    }
    public void test2Returns2() {
        String returnedVal = FizzBuzz.fizzbuzzify(2);
        assertEquals("2", returnedVal);
    }
}
```

Когда мы запустим тест, он ожидаемо завершится сбоем, т. к. наш метод `fizzbuzzify()` всегда возвращает 1 и никогда не возвращает 2. Также обратим вни-

мание, что наш первый тест не стал падать после добавления второго теста — единственным падающим тестом оказался новый тест `test2Returns2()`. Исправить код будет довольно просто, так?

```
public class FizzBuzz {
    private static String fizzbuzzify(int num) {
        if (num == 1) {
            return "1";
        } else {
            return "2";
        }
    }
    public static void main(String args[]) {
    }
}
```

Теперь все тесты проходят! Мы гении программирования! Похлопаем сами себя по плечу!

Конечно, такой шаблон нельзя использовать в дальнейшем. Подвергнем код незначительному рефакторингу, чтобы он работал со всеми целыми значениями, а не требовал нового `else if` для каждого значения:

```
public class FizzBuzz {
    private static String fizzbuzzify(int num) {
        return String.valueOf(num);
    }
    public static void main(String args[]) {
    }
}
```

Гораздо лучше! Тесты по-прежнему проходят, поэтому мы можем двинуться к новому циклу петли. Давайте добавим тест для `fizzbuzzify(3)`, возвращающего "Fizz":

```
@Test
public void test3ReturnsFizz() {
    String returnedVal = FizzBuzz.fizzbuzzify(3);
    assertEquals("Fizz", returnedVal);
}
```

`fizzbuzzify(3)` действительно вернет "3", что, конечно, вызовет сбой нашего теста. Однако это можно быстро исправить!

```
private static String fizzbuzzify(int num) {
    if (num == 3) {
        return "Fizz";
    } else {
        return String.valueOf(num);
    }
}
```

Ура, наши тесты проходят! Впрочем, это не идеальное решение — оно будет работать только с 3, а мы знаем, что должно быть любое число, которое без остатка делится на 3. Немного рефакторинга, и мы сможем обрабатывать любое число, делящееся на 3:

```
private static String fizzbuzzify(int num) {
    if (num % 3 == 0) {
        return "Fizz";
    } else {
        return String.valueOf(num);
    }
}
```

Теперь мы можем найти новые идеи для последующих юнит-тестов — возможно, мы захотим проверить 6, 9 или 3000. Но сейчас давайте двигаться дальше и добавим "Buzz":

```
@Test
public void test5ReturnsBuzz() {
    String returnedVal = FizzBuzz.fizzbuzzify(5);
    assertEquals("Buzz", returnedVal);
}
```

И снова наш тест упал, поэтому добавим дополнительное `else` в наши условные выражения:

```
private static String fizzbuzzify(int num) {
    if (num % 3 == 0) {
        return "Fizz";
    } else if (num % 5 == 0) {
        return "Buzz";
    } else {
        return String.valueOf(num);
    }
}
```

Теперь тесты проходят, и, похоже, рефакторинг уже не требуется. Давайте создадим последний тест для проверки возврата значения "FizzBuzz":

```
@Test
public void test15ReturnsFizzBuzz() {
    String returnedVal = FizzBuzz.fizzbuzzify(15);
    assertEquals("FizzBuzz", returnedVal);
}
```

Этот тест завершится сбоем, т. к. текущий метод вернет "Fizz":

```
private static String fizzbuzzify(int num) {
    if ((num % 3 == 0) && (num % 5 == 0)) {
        return "FizzBuzz";
    } else if (num % 3 == 0) {
        return "Fizz";
```

```
    } else if (num % 5 == 0) {
        return "Buzz";
    } else {
        return String.valueOf(num);
    }
}
```

Теперь тесты проходят! Мы можем двигаться дальше и заняться рефакторингом — например, вынести `num % 3 == 0` и `num % 5 == 0` в отдельные методы, но мы хотели показать простую схему процесса TDD. Зачастую в реальной разработке шаги оказываются больше, но ключевой принцип, который нужно держать в голове, заключается в поддержании тестов сравнительно конкретными и ориентированными на определенные выходные значения. Группирование входных и выходных значений в классы эквивалентности, как обсуждалось ранее, поможет вам определить, что именно необходимо протестировать и в каком порядке.

15.5. Преимущества TDD

Одним из основных преимуществ TDD является автоматическое создание тестового набора во время разработки. Не нужно беспокоиться о том, чтобы уложиться в сроки разработки тестового фреймворка и написания теста, потому что сам процесс разработки создаст их для вас. К сожалению, тестирование часто откладывается до конца проекта (что означает, что ему будет уделено мало внимания или его даже постараются избежать). Использование TDD гарантирует, что у вас будет хотя бы какой-то тестовый набор, даже если остальное тестирование свернуто.

Когда тесты являются частью рабочего процесса, люди помнят об их создании. Вероятно, вы не думаете каждое утро после пробуждения "о, мне надо запомнить почистить зубы сегодня"; это просто привычка, которая является частью вашего утреннего распорядка. Поэтому вы, скорее всего, будете создавать тесты, если вы занимаетесь этим постоянно и их разработка вошла у вас в привычку. Это определенно здорово, потому что количество тестов коррелирует с качеством кода! Когда чем-то занимаешься часто, делать это легко. Любые проблемы с тестовым набором будут найдены быстро, а заняться проблемным частями или сложным для тестирования кодом можно будет раньше.

Еще одним преимуществом является релевантность тестов, которые вы создаете в рамках TDD, т. к. новые тесты сперва должны завершаться сбоем (подтверждая, что тест не является избыточным), и должны пройти, прежде чем цикл написания части кода будет считаться "завершенным". Менее вероятно, что в этом случае вы напишете тесты, которые являются ненужными или тавтологическими, подвергнете избыточному тестированию одну область или не протестируете достаточно другую.

Если вы пишете тесты по факту, легко попасть в ловушку предположения, что программа делает все правильно. Однако помните, что эти тесты должны сравнивать ожидаемое поведение с наблюдаемым. Если вы "ожидаете" что-то, что уже "наблюдается", есть риск, что ваши тесты окажутся тавтологическими!

TDD заставляет вас двигаться маленькими шагами. Это помогает гарантировать, что вы не уходите далеко в работе над чем-то. Если вы написали четыре строчки кода и создали дефект, то в этом случае гораздо проще понять, где вы ошиблись, чем если бы вы нашли дефект после написания тысячи строк кода. Мне нравится сравнивать написание кода с пересечением населенной злыми пингвинами Антарктиды, и тесты являются нашими крепостями против пингвинов. Можно легко пересечь целый континент, если через каждый километр или два вы строите очередную крепость, и всегда есть место для отступления, если у пингвинов появится безумный блеск в глазах. Вы можете выбрать другой путь, возможно, тот, где меньше пингвинов, и поставить форт там. Переход Антарктиды без строительства противопингвинных фортов окажется безрассудным маневром, потому что любая атака пингвинов может отбросить вас к кораблю, с которого вы высадились. Написание больших кусков кода без тестов точно так же не позволяет добиться значительного прогресса, поскольку любой дефект может привести к тому, что вам придется отказаться от большей части уже написанного вами кода.

Когда вы тестируете код с самого начала, более вероятно, что этот код будет тестируемым. Вы не только научитесь тестировать код в этом конкретном приложении, т. к. вы занимаетесь этим все время, но вы вряд ли напишете код, который не сможете протестировать. Почему? Ваш код должен проходить уже написанные вами тесты, поэтому вы будете стараться писать его таким, чтобы он был тестируемым. Поскольку вы также постоянно добавляете его в кодовую базу, а не рассматриваете как одну большую "версию", являющуюся одним гигантским блоком, то ваш код также будет расширяемым. Вы расширяете его с каждым циклом "красный — зеленый — рефакторинг"!

Использование TDD обеспечивает 100%-ное тестовое покрытие или близкое к нему. Хотя покрытие кода не является идеальной метрикой — в коде, полностью покрытом тестами, может прятаться множество дефектов, — оно подтверждает, что вы по крайней мере единожды проверяете каждую строчку кода. Это намного лучше, чем делать это в массе проектов.

Разработка через тестирование предоставляет структурированный фреймворк для написания программного обеспечения. Хотя, конечно же, существует немало недостатков (некоторые из них будут перечислены далее) и ситуаций, для которых данная методика является неоптимальной, TDD открывает вам путь движения вперед. Неукоснительные, но гибкие шаги цикла "красный — зеленый — рефакторинг" дают разработчикам список того, что делать дальше. Вы постоянно добавляете тесты, пишете код, который должен проходить эти тесты, и осуществляете рефакторинг. Когда у вас нет фреймворка, которому нужно следовать, вы можете потратить много времени на рефакторинг существующего кода, или недостаточно времени на написание тестов, или слишком много энергии на написание тестов, но не кода. Как минимум вы должны учитывать, сколько времени и ресурсов вы хотели бы потратить на каждую составляющую. С TDD у вас уже есть готовые ответы, вам не нужно искать их, и поэтому вы можете спокойно работать над другими вещами. Есть замечательная книга Атула Гаванде "Манифест чек-листа" (Atul Gawande "The Checklist Manofesto"), в которой объясняется, как наличие чек-листа

(даже такого простого, как "красный — зеленый — рефакторинг") может помочь вам в разнообразных начинаниях. Программный инжиниринг не является исключением.

Наконец, и самое главное — разработка программного обеспечения с использованием TDD может дать вам уверенность в вашей кодовой базе. Вы идете по канату с натянутой снизу страховочной сеткой; вы знаете, что программа, которую вы пишете, может по крайней мере делать то, о чем вы ее попросите. У вас есть защитные ограждения, которые предупредят вас, когда вы вызовете проблемы в других частях приложения. У вас есть опытная команда разработчиков, которая сообщит вам о следующих шагах. Вы не одиноки.

15.6. Недостатки TDD

Как было упомянуто выше, не существует универсального лекарства в разработке программного обеспечения. У использования TDD много преимуществ, но не во всех случаях она оказывается подходящей методикой. Знайте о следующих недостатках перед тем, как решиться использовать TDD.

Разработка с TDD означает написание множества юнит-тестов. Если вашу команду нужно принудить к первостепенному тестированию программного обеспечения, фокусирование на юнит-тестах может вытеснить другие виды тестирования, такие как тестирование производительности и системное тестирование. Нужно помнить, что даже если вы написали много юнит-тестов для метода, реализующего некий функционал, это не означает, что вы полноценно протестировали данный функционал.

Нет сомнений, что в краткосрочной перспективе написание тестов означает, что на разработку того же количества функций уйдет больше времени. Конечно, имеются преимущества в виде улучшенного качества кода. Тем не менее если кто-то ждал до ночи, прежде чем проект станет доступен для программного класса (хм), то, вероятно, что TDD является не лучшим решением. В этом случае бронебойная программа, которая не соответствует половине требований, оказывается значительно хуже программы, которая делает все, что должна, до тех пор, пока вы не передадите ей неправильный параметр, не нажмете <Ctrl>+<C> или вдруг не начнете слишком тяжело дышать рядом с ней. Но для больших проектов или для проектов без подобных малых дедлайнов использование TDD или похожей методики часто оказывается более быстрым в долгосрочной перспективе. Можно привести хорошую аналогию — написание программы без тестов подобно езде на картинге; кажется быстро, но на самом деле медленно. Написание программы с тестами подобно управлению реактивным самолетом; кажется медленно, но на самом деле очень быстро.

Традиционно TDD предоставляет меньше времени для принятия архитектурных решений. Из-за малого времени цикла "красный — зеленый — рефакторинг" меньше времени может быть потрачено на дизайн и архитектуру в противоположность разработке кода, реализующего пользовательские истории. Во многих случа-

ях, таких как разработка простого веб-приложения, это совершенно нормально. Архитектура по умолчанию может быть нормальной, и трата лишнего времени на размышления о ней может оказаться контрпродуктивной. В других случаях, особенно когда дело касается новых видов программного обеспечения или областей его работы, выбор архитектуры может быть сложным и логически вытекающим, и имеет смысл потратить больше времени в начале проекта на размышления о ней.

Более того, может оказаться трудным внести изменения в архитектурный дизайн в процессе разработки. Хотя методика предполагает гибкость, некоторые решения по дизайну могут потребовать множества модификаций после написания кода. Будет проще потратить время в начальном цикле разработки ПО для обдумывания, что необходимо сделать, и не предполагать, что вы сможете внести изменения позже.

Для некоторых областей приложений разработка через тестирование является определенно ошибочным подходом. Если вы создаете прототип чего-либо и не уверены, каким должно быть ожидаемое поведение, но знаете, что оно может быстро поменяться и не попасть к потребителю, тогда TDD будет излишним. Постоянно растущий набор тестов, который обычно служит вам страховочной сеткой, окажется камнем на вашей шее. Если вы пытаетесь разобраться с ожидаемым поведением по ходу работы, не имеет смысла использовать методику, которая предполагает, что вы знаете, каким будет ожидаемое поведением. С другой стороны, предельно критичные для безопасности приложения, такие как системы электропитания или управления авиационным радиоэлектронным оборудованием, потребуют гораздо большего объема проработки дизайна и продуманности, чем может обеспечить TDD. В этом случае TDD может оказаться *слишком* гибким.

Помните, когда вы пишете автоматизированные тесты, вы в действительности пишете код. Конечно, этот код выглядит немного по-другому, но вы снова и снова добавляете накладные расходы к вашей кодовой базе. Это еще что-то, что может пойти не так, что-то, что потребует рефакторинга, что-то, что потребуется обновлять каждый раз после изменений в требованиях или направлении проекта. Хотя эти накладные расходы часто компенсируются увеличением качества кода приложения, в некоторых случаях может оказаться не так. В особенно маленьких проектах и скриптах может быть быстрее просто выполнить ручное тестирование, чтобы убедиться, что приложение делает то, что должно, и не тратить время на создание полноценного фреймворка для этого приложения.

Наконец вы, как инженер, часто начинаете новый проект не с самого начала (т. е. не с нуля). Часто вы модифицируете или добавляете функции в уже существующее программное обеспечение, бóльшая часть которого была написана с использованием других методик или парадигм. Если вы начинаете работать над проектом, использующим очень жесткую методику "Водопад" или код которого с трудом поддается тестированию, использование TDD принесет больше проблем, чем преимуществ. Это также может отдалить вас от участников вашей команды или заставить потратить слишком много времени на разработку ваших функций.

Глава 16

Написание тестируемого кода

Хотя мы рассматривали тестирование с точки зрения тестировщика, зачастую бывает полезным рассмотреть его с точки зрения кого-то, кто занимается написанием кода. Как тестировщик, вы можете сделать жизнь разработчика проще, определяя дефекты и уровень качества программы, программист может сделать жизнь тестировщика проще, обеспечивая легкое тестирование написанного им кода. В современных реалиях разработчик и программист очень часто — один человек, особенно если речь идет о юнит-тестах. Таким образом, имеет смысл (с точки зрения теории игр) сделать код максимально тестируемым.

Создавая код тестируемым, вы не только упростите создание тестов, но и получите ряд других преимуществ. Вы будет стремиться писать хороший модульный код. Вносить изменения в будущем окажется проще, т. к. у вас будет больше тестов и вы сможете писать лучшие тесты. Код будет более легким для понимания, и проблем будет меньше, т. к. вы сможете протестировать все граничные случаи. Хотя вашей целью может быть написание тестируемого кода, в конечном итоге вы будете писать код, который лучше во всех отношениях.

16.1. Что мы понимаем под тестируемым кодом?

В каком-то смысле, практически весь код тестируемый. Пока вы контролируете вход и наблюдаете выход, вы можете протестировать любую часть программного обеспечения. Под **тестируемым кодом** мы понимаем код, который легко тестировать в автоматическом режиме на множестве уровней абстракции. Это означает, что написание тестов для систем черного ящика будет простым, что написание юнит-тестов (для классов, методов и других модулей кода) не окажется неоправданно сложным, а более продвинутое тестирование (такое, как тестирование производительности и безопасности) возможно. Не всякий тестируемый код является хорошим кодом — можно написать ужасную мешанину из непроизводительного спагетти-кода, который тем не менее легко тестировать. Однако любой хороший код является тестируемым кодом. Возможность написать тесты для определенной части кода — неотъемлемая часть хорошо написанного кода.

Давайте представим, что мы тестируем следующий фрагмент кода. Это часть видеоигры, в которой симулируется движение птицы. Нажатие кнопки заставляет птицу лететь и изменяет высоту полета и расположение на экране.

```
public class Bird {
    public int _height = 0;
    public int _location = 0;

    public void fly() {
        Random r = new Random();
        _height += r.nextInt(10) + 1;
        _location += r.nextInt(10) + 1;
        Screen.updateWithNewValues(_height, _location);
    }
}
```

Хотя это простой метод, его будет довольно сложно тестировать при помощи юнит-тестов. В конце концов, здесь нет возвращаемых значений, к которым можно применить утверждения. Результаты будут проверяться с учетом переменных уровня класса. Невозможно дать "правильный" ответ, потому что имеется зависимость от генератора случайных чисел, которую невозможно переопределить. Невозможно проверить, что `Screen` был обновлен без наличия реального действующего объекта `Screen`. Всё это означает, что протестировать код в изоляции будет трудно, и поэтому очень трудно реализовать юнит-тестирование. На протяжении оставшейся части этой главы мы рассмотрим стратегии, которые гарантируют, что наш код не окажется таким.

16.2. Основные стратегии тестируемого кода

При написании тестируемого кода на уровне юнит-тестирования следует помнить о двух ключевых концепциях:

1. Гарантирование, что код сегментирован.
2. Гарантирование, что события повторяемы.

Если какой-либо конкретный метод при работе взаимодействует только с несколькими другими частями системы (классами, методами, внешними программами и т. п.) — в идеале, если на его результаты влияют только значения, передаваемые в качестве параметров, — тогда его будет сравнительно легко протестировать. Однако если он взаимодействует со множеством других частей, то может оказаться очень сложно быстро протестировать его. Вы должны убедиться, что работает не только нужная часть системы, но и те, от которых она зависит. Вы можете использовать тестовых двойников, но если код писался без их учета, это может оказаться невозможным. Давайте рассмотрим некий код, который не сегментирован, и поэтому его крайне сложно протестировать:

```
public int getNumGiraffesInZoo() {
    String animalToGet = "Giraffe";
```

```
    DatabaseWorker dbw = DatabaseConnectionPool.getWorker();
    NetworkConnection nwc = NetworkConnectionFactory.getConnection();
    dbw.setNetworkConnection(nwc);
    String sql = SqlGenerator.generate("numberQuery", animalToGet);
    int numGiraffes = 0;
    try {
        numGiraffes = (int) dbw.runSql(sql);
    } catch (DatabaseException dbex) {
        numGiraffes = -1;
    }
    return numGiraffes;
}
```

Сперва может показаться, что все можно легко протестировать — в конце концов, нужно всего лишь утверждать, что число жирафов равняется ожидаемому числу, — но данный код не является хорошо сегментированным. Правильность его работы зависит не только от правильности работы классов `DatabaseWorker`, `NetworkConnection`, `DatabaseConnectionPool`, `NetworkConnectionFactory`, `SqlGenerator` и их соответствующих методов, но также нет возможности использовать двойников вместо них, потому что все они встроены в метод. Проблема с любым из сторонних элементов станет причиной падения теста, и понять, почему это произошло, может оказаться довольно трудно. Что же вы тестировали — нужный вам метод или одну его многочисленных зависимостей?

Давайте структурируем этот метод так, чтобы он стал хорошо сегментированным:

```
public int getNumGiraffesInZoo(DatabaseWorker dbw, SqlGenerator sqlg) {
    String animalToGet = "Giraffe";
    int numGiraffes = 0;
    String sql = sqlg.generate("numberQuery", animalToGet);
    try {
        numGiraffes = (int) dbw.runSql(sql);
    } catch (DatabaseException dbex) {
        numGiraffes = -1;
    }
    return numGiraffes;
}
```

Хотя этот код еще не оптимальный, уже, по крайне мере, возможно заместить все эти зависимости двойниками. В методе меньше внимания уделяется элементам, не связанным с самим методом (например, установление сетевых подключений к воркеру базы данных, который, вероятно, относится к классу `DatabaseConnectionPool` или к самому классу `DatabaseWorker`, но определенно не к методу `getNumGiraffesInZoo`). Мы можем пойти немного дальше и переместить всю работу с базами данных в свой класс, обернув ее так, что только важные части будут видны нашему методу:

```
public int getNumGiraffesInZoo(AnimalDatabaseWorker adbw) {
    String animalToGet = "Giraffe";
    int numGiraffes = 0;
```

```
    try {
        numGiraffes = adbw.getNumAnimals(animalToGet);
    } catch (DatabaseException dbex) {
        numGiraffes = -1;
    }
    return numGiraffes;
}
```

Мы понизили число зависимостей до одного класса `AnimalDatabaseWorker` и вызываем только один его метод.

Вторая концепция заключается в гарантировании, что всё, что вы делаете, является повторяемым. Вам определенно не захочется иметь тест, который работает правильно время от времени. Если есть проблема, вы должны знать о ней немедленно. Если проблемы нет, ложных срабатываний быть не должно.

Вы можете сделать тест повторяемым, если все значения, от которых он зависит, могут быть реплицированы. Это одна из (многих, многих, многих) причин, по которой глобальные переменные — в общем, Плохая Идея. Давайте рассмотрим тестирование следующего метода:

```
public int[] sortList() {
    if (_sortingAlgorithm == "MergeSort") {
        return mergeSort(_list);
    } else if (_sortingAlgorithm == "QuickSort") {
        return quickSort(_list);
    } else {
        if (getRandomNumber() % 2 == 0) {
            return bubbleSort(_list);
        } else {
            return bogoSort(_list);
        }
    }
}
```

Что произойдет, если мы запустим его? Поток выполнения изначально зависит от двух не передававшихся переменных, поэтому если мы хотим добиться повторяемости нашего теста, нам перед тестированием необходимо выполнить дополнительные проверки, чтобы убедиться, что это правильные значения. Кто знает, какая часть кода изменила переменные `_list` и `_sortingAlgorithm`?! Даже если мы знаем заранее, что все переменные установлены правильно, как можно тестировать то, что произойдет, если `_sortingAlgorithm` будет установлена в значение, отличное от `"MergeSort"` или `"QuickSort"`? Метод вызовет либо `bubbleSort()`, либо `bogoSort()`, и нет (разумного) способа заранее определить, какой именно из них. Если ошибка в `bubbleSort()`, но в не в `bogoSort()`, тест может случайным образом время от времени завершаться неуспешно.

Для эффективного тестирования код должен быть сегментирован и написан таким образом, чтобы гарантировать постоянство прохождения определенных частей кода с определенными значениями. Если это так, тогда каждый раз будут получены оди-

наковые выходные значения и тесты не будут падать случайным образом. Пишите свой код таким образом, чтобы не допускать случайных сбоев тестирования, а не провоцировать их.

16.3. Предусмотрите сценарный интерфейс

Хотя в предыдущем разделе рассматривалось, как гарантировать тестируемость кода с точки зрения "белого ящика", создание программы, которая легко тестируема с точки зрения "черного ящика" и системного уровня, может быть еще более сложным. В общем предполагается, что методы должны вызываться — для этого существует встроенный интерфейс и определены параметры для каждого метода. Однако полноценные системы могут работать сами по себе с минимальным влиянием извне.

Некоторые интерфейсы "автоматически" являются сценарными. Например, при создании веб-приложения вы можете использовать фреймворк для веб-тестирования (такой как Selenium или Capybara) для доступа к различным страницам, нажатию кнопок, ввода текста и выполнения всего того, что обычно можно делать на веб-странице. Текстовые интерфейсы также сравнительно легко заставить работать со сценариями, т. к. они просто принимают текст и выводят текст. Вывод можно перенаправить в файл или иным образом проверить на правильность.

Программы, которые не предоставляют метод написания сценариев в силу своего интерфейса, такие как GUI-приложения, в идеале должны иметь встроенные сценарии. Это можно реализовать через **тест-хуки** (test hooks) — "спрятанные" методы, которые предоставляют возможность ввести данные или получить информацию из программы. Подобные методы обычно не являются общедоступными, и воспользоваться ими можно при помощи специальных ключей доступа или прочих мер безопасности.

Существуют недостатки добавления хуков в программу или в любой сценарный интерфейс. С одной стороны, это угроза безопасности — если кто-то получит доступ к тестовым хукам, он сможет определить скрытую информацию, перезаписать данные или выполнить другие вредоносные действия (или все это может произойти в результате любопытства и ошибки). Добавление сценарного интерфейса приведет к дополнительной сложности программы, а также к увеличению ее размера. Интерфейс может стать тормозом производительности.

Еще хуже то, что сценарии могут создать у вас ложное чувство безопасности. Добавление отдельного способа доступа к пользовательским функциям означает, что вам потребуется (как минимум) два параллельных способа использования системы. В некоторых случаях может оказаться так, что функционал, нормально работающий через сценарный интерфейс, не будет работать при доступе к нему "обычным" способом.

Время, потраченное на создание сценарного интерфейса, также означает, что будет потрачено меньше времени на разработку другого функционала программы или на улучшение ее качества. Этот компромисс допустим, но его следует оценивать в контексте проекта.

Вы можете тестировать графические и неграфические интерфейсы без тестовых хуков, но обычно это сложно. Написание кода для непосредственного взаимодействия с чем-либо часто является самым легким и прямым путем. Существуют программы, позволяющие вам напрямую манипулировать курсором, делать скриншоты конечных результатов и выполнять прочее взаимодействие, которое не является сценарием. Однако эти инструменты часто бывают привередливыми и требуют определенной ручной проверки.

16.4. Написание тестов заранее

В идеале вам необходимо использовать парадигму TDD или что-то подобное. Даже если вы не используете TDD, вы должны писать множество тестов примерно в то же самое время, когда вы пишете код. Вы быстро поймете это, когда окажется, что написанный вами код не может быть протестирован и вы не можете дальше создавать код, с тестированием которого возникнут проблемы, когда вы займетесь этим "позже". Обратите внимание, что "позже" часто означает "никогда".

Чем дольше вы будете писать код без подготовки тестов для него, тем более вероятно, что вы создадите код, который сложно протестировать. Даже если вы предполагаете, что он будет тестируемым, зачастую вы не догадываетесь, что написанное вами будет сложно протестировать по какой-либо из причин. Написание тестов дает вам подтверждение, что вы идете по правильному пути.

16.5. Пусть ваш код будет DRY

Акроним DRY означает "Don’t Repeat Youself" ("Не повторяйся") и является ключевым принципом того, что ваш код будет не только тестируемым, но и просто лучше. Простейшим примером создания кода, который не соответствует DRY, является копирование с незначительным изменением имени метода:

```
public int[] sortAllTheNumbers(int[] numsToSort) {
    return quickSort(numsToSort);
}
public int[] sortThemThereNumbers(int[] numsToSort) {
    return quickSort(numsToSort);
}
```

Если вы думаете, что это нелепый надуманный пример, то я лично видел и исправлял подобный код. Хотя в данном случае все кажется очевидным, потому что методы располагаются рядом, дублирующий код любит прятаться в маленьких закоулках. Подобное случается, когда у вас класс состоит из тысяч строчек кода. Программисту нужен метод, и он быстро добавляет его вместо того, чтобы поискать, не существует ли уже такой метод; кто-то создает "тестовую" версию метода, а потом забывает о ней, или же кто-то просто начинает копировать фрагменты кода, найденные онлайн. Если все продолжает работать нормально, то для поиска подобных проблем очень мало стимулов, не говоря уже об их устранении. И даже если кто-то

заметит проблему, то ее устранение может рассматриваться как выход за рамки того, над чем он работает в текущий момент. А при исправлении кода почти наверняка потребуется рефакторинг зависимого кода. Однако если ваш код не соответствует принципу DRY, это означает создание большего количества тестов (часть из которых будет ненужна), опасность при рефакторинге и просто избыточность. Выбор одного метода вместо нескольких копий сэкономит ваше время и энергию в долгосрочной перспективе, т. к. вы сократите затраты на дополнительные юнит-тесты и избавите от путаницы тех, кто будет заниматься этой работой в дальнейшем.

Что произойдет, если вы решите, что больше не хотите использовать метод `quicksort()`, или пожелаете поддерживать числа с плавающей запятой помимо целых? Вам придется внести правки в двух (или больше... нет ограничений того, сколько раз код можно скопировать и вставить) местах, о которых легко забыть. Это удваивает вероятность ошибки или того, что вы сделаете что-то не в том месте, где необходимо. Тестирование подобных вещей может оказаться трудным. Постарайтесь удалить повторяющийся код как можно раньше.

Приведенный выше пример с дублированием метода является довольно простым, но вы можете столкнуться с более сложными случаями дублирования кода. Каждый раз, когда вы обнаруживаете повторяющийся код, то даже если он находится где-то среди других операторов, лучше всего будет поместить его в собственный метод. Рассмотрим следующий код SQL, который должен вызываться каждым из приведенных ниже методов, чтобы определить, сколько видов пород существует в базе данных:

```
public int getNumberOfCats(String catBreed) {
    int breedId = DatabaseInterface.execute(
        "SELECT BreedID FROM CatBreeds WHERE BreedName = " + catBreed);
    int numCats = DatabaseInterface.execute(
        "SELECT COUNT(*) FROM Cats WHERE BreedID = " + breedId);
    return numCats;
}

public int getNumberOfPigeons(String pigeonBreed) {
    int breedId = DatabaseInterface.execute(
        "SELECT BreedID FROM PigeonBreeds WHERE BreedName = " + pigeonBreed);
    int numPigeons = DatabaseInterface.execute(
        "SELECT COUNT(*) FROM Pigeons WHERE BreedID = " + breedId);
    return numPigeons;
}
```

Хотя запросы не совсем одинаковые, они достаточно похожи, чтобы поработать с ними с точки зрения принципа DRY. В конце концов, мы всегда можем передать параметры, чтобы настроить поведение так, как нам нужно:

```
public int getNumAnimals(String animalType, String breed) {
    int breedId = DatabaseInterface.execute(
        "SELECT BreedID FROM " + animalType + "Breeds WHERE BreedName = " + breed);
```

```
    int numAnimals = DatabaseInterface.execute(
        "SELECT COUNT(*) FROM " + animalType + "s WHERE BreedID = " + breedId);
    return numAnimals;
}
public int getNumberOfCats(String catBreed) {
    return getNumAnimals("Cat", catBreed);
}
public int getNumberOfPigeons(String pigeonBreed) {
    return getNumAnimals("Pigeon", pigeonBreed);
}
```

Теперь код гораздо более гибкий и поддерживаемый. Добавление новых животных потребует лишь добавления методов, которые будут вызывать `getNumAnimals()` с соответствующими параметрами. Прямое тестирование базы данных, которое может быть довольно сложным, теперь ограничено одним конкретным методом. Энергию и усилия можно направить на тестирование более абстрактного метода, чем рассредоточивать тестирование по нескольким методам с широкими обязанностями.

16.6. Внедрение зависимости

Как мы видели, использование жестко прописанных зависимостей в ваших методах может стать причиной сложности их тестирования. Давайте предположим, что у вас есть ссылка уровня класса на объект `Duck`, созданный объектом `Pond`. У класса `Pond` имеется метод `sayHi()`, который произносит "Hi!", приветствуя всех животных в пруду:

```
public class Pond {
    Duck _d = new Duck();
    Otter _o = new Otter();
    public void sayHi() {
        _d.say("Hi!");
        _o.say("Hi!");
    }
}
```

Как это протестировать? Нет простого пути, позволяющего проверить, что утка получила сообщение "Hi!". **Внедрение зависимости** (dependency injection) позволит вам избежать этой проблемы. Хотя этот термин звучит солидно и позволяет сойти за специалиста по тестированию, на самом деле это довольно простая концепция, с которой вы наверняка сталкивались в своей практике. В нескольких словах, внедрение зависимости означает передачу зависимостей в качестве параметров методу, в отличие от жесткого прописывания ссылок на них. Благодаря этому гораздо легче передавать тестовые двойники или моки.

Давайте перепишем класс `Pond` так, чтобы он позволял внедрение зависимостей:

```
public class Pond {
    Duck _d = null;
    Otter _o = null;
```

```
    public Pond(Duck d, Otter o) {
        _d = d;
        _o = o;
    }
    public void sayHi() {
        _d.say("Hi!");
        _o.say("Hi!");
    }
}
```

Обратите внимание, что вы передаете `Duck` и `Otter` в конструктор, таким образом, переместив ответственность за их создание за пределы класса `Pond`. Благодаря этому вам не нужно отправлять настоящих уток и выдр. Вместо этого вы можете отправить моки, которые затем проверят, что метод `say()` был вызван с параметром "Hi!". Внедрение зависимости позволит вам тестировать ваши методы гораздо проще, чем если бы зависимости создавались автоматически.

16.7. Недружественные к тестированию функции и конструкции

TUF (test-unfriendly features) — это недружественные к тестированию функции, а TUC (test-unfriendly constructs) — это недружественные к тестированию конструкции. TUF являются программными функциями, которые по своей сути трудно тестировать. Например, запись в базу требует либо хорошо продуманных тестовых двойников, либо зависимости от множества внешних факторов (драйверы базы данных, сама по себе база данных, использование диска и т. д.). Другими недружественными к тестированию функциями могут быть взаимодействие через сеть, использование библиотеки по работе с окнами или запись на диск. Во всех этих случаях, если вы не предоставляете соответствующие двойники, протестировать эту функции будет очень сложно. Если вы предоставляете двойники, тогда задача становится "относительно сложной".

Недружественные к тестированию конструкции являются конструкциями в коде, где тестирование сложно по своей сути. В Java к ним можно отнести закрытые методы (для прямого доступа к которым требуется Reflection API), конструкторы или `final`-методы. Может оказаться сложным переопределить соответствующие методы или без проблем подготовить тестовые двойники для множества конструкций TUC. Это дополнительный уровень сложности, который в сочетании с недружественной для тестирования функцией может сделать ваши тесты запутанными и сложными для понимания.

Один из способов поддержания сложности ваших тестов на приемлемом уровне — не размещать функции TUF внутри конструкций TUC. Поместите их в собственные методы или иным образом сегментируйте и спроектируйте код так, чтобы конструкции TUC содержали минимальный и легко тестируемый код.

16.8. Работа с чужим унаследованным кодом

Не у каждого была возможность прочитать отличную книгу с главой, посвященной написанию тестируемого кода. Многие существующие кодовые базы были написаны людьми, незнакомыми с современными техниками программного инжиниринга либо по незнанию, либо потому, что эти техники не были распространены в то время. Было бы глупо ожидать от людей, писавших код на FORTRAN IV в 1966 году, использования техник тестирования, которые получили распространение в девяностые годы. Код, который используется в производстве, но не соответствует передовым практикам современной разработки программного обеспечения и часто имеет некачественное — или даже не имеет — автоматизированное тестовое покрытие, известен как **чужой унаследованный код** (legacy code).

Если сказать кратко, работать с таким кодом трудно. И это не обойти. Вы не сможете использовать множество преимуществ хорошего тестового набора; вы не сможете взаимодействовать с авторами-разработчиками в случае проблем, неоднозначностей или недокументированного кода; и может быть трудно понять устаревший код, написанный в старом стиле. Однако такое происходит часто, особенно если вы работаете с компанией, сотрудники которой пишут код довольно давно. Не имеет смысла переписывать миллионы строк кода каждый раз, когда вы хотите добавить новую функцию в используемый вами программный набор.

Когда вы обнаружите, что работаете с чужим унаследованным кодом, самым важным для вас станет создание **тестов фиксирования** (pinning tests). Тесты фиксирования являются автотестами, обычно юнит-тестами, которые проверяют существующее поведение системы. Обратите внимание, что существующее поведение не всегда является ожидаемым или правильным поведением. Цель теста фиксирования — посмотреть, как ведет себя программа перед внесением в нее каких-либо изменений. Очень часто оказывается, что странные граничные случаи в действительности используются теми, кто работает с системой. И не в ваших планах непреднамеренно воздействовать на эти случаи при добавлении новой функции, если только вы специально не решили это сделать. Внесение непреднамеренных изменений может оказаться опасным и для вас, и для пользователей программы. Обратите внимание, это не означает игнорирование вами факта, что программа работает неправильно. Однако ее исправление следует рассматривать как процесс, отдельный от создания тестов фиксирования.

При работе с чужим унаследованным кодом вы должны ясно представлять функции, которые вы добавляете, и дефекты, которые вы исправляете. Легко начать исправлять каждую ошибку, которую вы видите, но и так же легко создать проблемы. Вы начинаете забывать, что изначально начинали исправлять, вы не сосредоточены на чем-то одном и вносите изменения в массивные скопления вместо того, чтобы двигаться последовательно "шаг за шагом".

Лично мне нравится держать открытым небольшой тестовый файл с изменениями, которые я хотел бы внести в будущем, но которые в настоящее время находятся за рамками моей работы. Например, я редактирую класс, чтобы добавить новый

метод. Я замечаю, что в файле присутствует множество "магических чисел", которые нигде не определены. Я делаю пометку, что необходимо подвергнуть этот класс рефакторингу. Это не означает, что я не буду использовать соответствующие константы с понятными именами в добавляемом мною методе. Однако если я попытался бы все исправить в файле, с которым работаю, то потратил бы в три раза больше времени, чем если бы занимался минимумом, который мне нужен. Не самое лучшее расходование моего времени.

Добиваться минимума считается не самой лучшей жизненной стратегией, но вполне допустимо при написании программного обеспечения. Вы можете вносить небольшие изменения в кодовую базу, двигаться небольшими поступательными шагами, потому что большие изменения чреваты трудными для поиска ошибками. Вы можете сделать правку из 10 000 строк, и если что-то пойдет не так, обзор кода может оказаться кошмаром. Но допустим, что вы сделали тысячу изменений из правок по 10 строк. После каждого этапа вы можете запустить все юнит-тесты и посмотреть, не проявит ли себя проблема. Отслеживание проблем после крошечных шагов гораздо проще, чем после одного гигантского обновления.

Кроме того, работа над минимумом может гарантировать, что вы разумно расходуете время. Возможно, не стоит тратить время на добавление документации к каждому методу в классе, над которым вы работаете, особенно если вы предполагаете, что он не будет изменен в ближайшее время. И не из-за того, что это не лучшая идея, а из-за того, что это не лучшее распределение вашего времени. Помните, что вам отпущено ограниченное время не только на этой Земле, но и для выполнения проекта. Потратите слишком много времени не на то — и он не будет завершен. Хотя стремление исправить все проблемы, которые вы видите в кодовой базе, является благородным порывом, грязный маленький секрет индустрии программного обеспечения заключается в том, что за кулисами прячется много всякого рода уродливого кода, который в большинстве случаев работает отлично. У вашего любимого банка в осуществляющем денежные переводы коде расположились тысячи переходов `GOTO`. У вашей любимой трехбуквенной правительственной службы имеются сотни тысяч строк кода, для которого никогда не использовали юнит-тесты (а также моки или заглушки). Можно поплакать о несовершенстве мире, но иногда нужно просто принять его таким, каков он есть.

Если возможно, при поиске кода для изменения начните с обнаружения **швов** (seams). Швы являются локациями в коде, где вы можете изменять *поведение* без модификации *кода*. Пожалуй, это легче всего увидеть на примерах. В первом методе нет шва — и нет возможности изменить поведение программы без модификации кода метода:

```
public void createDatabaseTable() {
    DatabaseConnection db = new DatabaseConnection(DEFAULT_DB);
    int status = db.executeSql("CREATE TABLE Cats "
        + "(CatID int, Name varchar(255), Breed varChar(255));");
}
```

Если вы измените базу данных, с которой работает этот код, вам потребуется изменить константу `DEFAULT_DB` или же изменить строчку кода. Нет возможности пере-

дать тестовый двойник или использовать фейковую базу данных в вашем тесте. Если вам потребуется добавить столбец, необходимо редактировать строку внутри метода. Теперь давайте сравним этот метод с методом, который является швом:

```
public int executeSql(DatabaseConnection db, String sqlString) {
    return db.executeSql(sqlString);
}
```

Если вы хотите использовать соединение со второй базой данных, просто передайте ее в качестве параметра при вызове метода. Если вы хотите добавить столбец, можно просто изменить строку. Даже при отсутствии документации можно проверить граничные и угловые случаи вроде выбрасывания исключения или возврата конкретного статуса ошибки. Так как вы можете исследовать эти случаи без непосредственного изменения какого-либо кода, будет гораздо проще определить, что вы ничего не сломали при написании тестов. В конце концов, вы ведь ничего не изменили! Если вы должны вручную редактировать код для получения наблюдаемого поведения, как в первом примере, любые полученные из тестов фиксирования результаты должны показаться подозрительными. Модификации кода не всегда оказываются такими простыми, как в примере выше. Вам может потребоваться отредактировать множество методов, и вы никогда не сможете быть полностью уверены, были бы результаты такими же при выполнении исходного кода, или же они стали итогом тех правок, которые вы внесли при попытках тестирования.

Обратите внимание, что наличие шва никак не означает, что код хороший. В конце концов, возможность передавать для выполнения произвольный SQL-запрос является довольно большой угрозой безопасности, если он не был очищен где-либо еще. Написание кода с большим количеством швов может быть излишеством и определенно увеличит кодовую базу. Просто швы позволяют вам начать выяснять, как система в настоящий момент реагирует на входные данные. Поиск швов позволит вам легко начать писать всеобъемлющие тесты фиксирования для того, чтобы убедиться, что вы охватываете множество граничных случаев.

И возможно, еще более важно, что с психологической точки зрения вы "не поддаетесь" кодовой базе и не опускаетесь на ее уровень. Если для какой-то функции нет юнит-тестов, то это не означает, что можно заниматься правками без добавления юнит-тестов. Если для кода какого-то класса нет комментариев, это не дает вам карт-бланш для того, чтобы не делать комментариев для вашего кода. По мере развития код имеет тенденцию скатываться к наименьшему общему знаменателю. Вы должны активно бороться с этой деградацией. Пишите юнит-тесты, исправляйте ошибки по мере их нахождения и документируйте код и функции, как это необходимо.

Если вам интересно более подробно ознакомиться с работой с чужим унаследованным кодом, обратите внимание, что существуют (по крайней мере) две замечательные книги по этой теме. Первая — это "Рефакторинг: Улучшение существующего кода" Мартина Фаулера (Martin Fowler "Refactoring: Improving the Design of Existing Code"), а вторая — "Эффективная работа с унаследованным кодом" Майкла Физерса (Michael Feathers "Working Effectively with Legacy Code"). Последняя окажется

особенно полезной при работе с кодом, для которого пока нет всеохватывающего тестового набора.

Что мне всегда казалось интересным в программном инжиниринге, так это то, что это область, в которой все меняется чрезвычайно быстро. Многие из техник, которые я обсуждаю, вероятно, покажутся такими же странным, какими во время написания этой книги кажутся инструкции `GOTO` и номера строк. Если это так, пожалуйста, будьте великодушны в вашей критике полвека спустя.

16.9. Заключительные мысли о написании тестируемого кода

Тестирование программного обеспечения — это не только написание тестов, но и написание кода, благодаря которому тесты можно создавать легко и качественно. В современном мире программного инжиниринга написание кода и написание тестов не являются взаимоисключающими. Многие (и я бы даже рискнул сказать, большинство) разработчики пишут свои юнит-тесты совместно с кодом, и многие тестировщики должны писать код для автоматизации своих тестов или создания инструментов тестирования. Даже если вы не пишете код как тестировщик, вас могут попросить изучить его или определить, как увеличить охват автоматизированного тестирования для программной системы.

Глава 17

Попарное и комбинаторное тестирование

Представьте, что вы тестируете текстовый редактор, а именно тестируете добавление эффектов к шрифтам, т. е. делаете их полужирными, курсивными, трехмерными, в верхнем индексе и т. д. Конечно, в любом достойном внимания текстовом редакторе эти эффекты могут быть объединены в одной области теста, так что у вас может быть слово, написанное одновременном полужирным и курсивом, или буква в верхнем индексе и трехмерная, или даже предложение, которое выделено полужирным шрифтом, курсивом, подчеркнуто, трехмерное и в верхнем индексе. Число возможных комбинаций — от абсолютного отсутствия эффектов (т. е. обычного текста) до использования всех возможных эффектов — равняется 2^n, где n равно количеству эффектов. Таким образом, если имеется 10 различных эффектов шрифта, то общее количество тестов, которые вам потребуется запустить, чтобы проверить все возможные комбинации шрифтов, равняется 2^{10}, или 1024. Это нетривиальное количество тестов для создания.

Если вы действительно хотите осуществить тестовое покрытие, вам придется протестировать каждую из всех этих различных комбинаций (т. е. только полужирный; полужирный и курсив; полужирный и верхний индекс; полужирный, верхний индекс, курсив, зачеркнутый, трехмерный; и т. д.). Представьте, что возникла бы проблема, когда буква отображается только курсивом, подчеркнутой, полужирной, в верхнем индексе и зачеркнутой. Вы не сможете найти этот дефект, пока у вас не появится время для выполнения всеохватывающего теста по всем комбинациям!

Тем не менее, оказывается, что такая ситуация встречается реже, чем вы думаете. Вам может не понадобиться тестировать все эти комбинации, чтобы найти большой процент дефектов в системе. На самом деле, согласно публикации "Практическое комбинаторное тестирование" Рика Куна (Rick Kuhn "Practical Combinatorial Testing"), до 90% всех дефектов в программной системе могут быть найдены простым тестированием всех комбинаций двух переменных. В нашем примере, если бы вы протестировали все возможные пары (полужирный и курсив, полужирный и верхний индекс, верхний индекс и трехмерный, и т. д.), вы нашли бы множество дефектов в программе, но использовали только часть времени тестирования. Из всех программных проектов, которые были проанализированы, максимальное количество переменных, взаимодействие которых вызывало дефект, оказалось равным шести. Помните об этом при создании тестов, вы можете найти практически

все дефекты, которые обнаружил бы всеобъемлющий тестовый подход, но при этом израсходуете на порядок меньше времени и ресурсов.

Эта техника называется **комбинаторным тестированием**. Тестирование всех возможных пар значений является разновидностью комбинаторного тестирования, но имеет свое название — **попарное тестирование** или **тестирование всех пар**. В начале этой книги мы обсуждали, что всеобъемлющее тестирование нетривиальной программы практически невозможно. С использованием описываемой в этой главе техники мы можем значительно уменьшить число тестов (в некоторых случаях на несколько порядков) и все равно тщательно протестировать систему.

Давайте рассмотрим очень простой пример — программу, в которой всего три переменные для форматирования символа: полужирный, курсив и подчеркивание. Их можно объединить так, что, например, символ может быть курсивным, или курсивным и полужирным, или полужирным, курсивным и подчеркнутым. Поскольку каждая из этих трех переменных булева (они могут принимать только значения "истина" или "ложь" — у вас не может быть "полукурсивного" символа), список всех возможных состояний символа может быть описан в **таблице истинности**. Она показывает все возможные значения, которые способна принимать заданная группа переменных.

Интересно заметить, что если вы рассматриваете "ложь" как 0, "истину" как 1, то построение таблицы истинности аналогично двоичному счету от 0 до размера таблицы минус один. В примере ниже мы начинаем с "000" (0), затем "001" (1), "010" (2) и так далее до "111" (7). Этот способ пригодится, если вы создаете таблицы самостоятельно.

```
   Полужирный | Курсив | Подчеркнутый
   -----------+--------+--------------
1. ложь       | ложь   | ложь
2. ложь       | ложь   | истина
3. ложь       | истина | ложь
4. ложь       | истина | истина
5. истина     | ложь   | ложь
6. истина     | ложь   | истина
7. истина     | истина | ложь
8. истина     | истина | истина
```

Для того чтобы исчерпывающе протестировать программу, нам потребуется запустить восемь тестов, использующих указанные выше данные. Теперь давайте представим, что наш крайне строгий менеджер настаивает, что время есть только для запуска шести тестов. Как можно понизить количество запускаемых тестов и при этом поддерживать высокую степень обнаружения дефектов?

Как объясняется выше, проверка всех пар переменных может помочь нам найти большой процент дефектов. В данном примере имеется три возможные пары переменных: полужирный и курсив, полужирный и подчеркнутый, курсив и подчеркнутый. Как мы узнали, что имеются только три возможные пары? Для небольшого количества переменных мы можем просто посмотреть, что нет других возможных

пар, но когда количество станет довольно большим, нам придется заняться подсчетами. Здесь потребуется математическое разъяснение.

17.1. Перестановки и комбинации

Возможно, вы забеспокоились, что название книги оказалось обманчивым и "дружеское" знакомство будет заполнено множеством математических операций. Однако не бойтесь — математика здесь сравнительно проста для понимания, а уравнения сведены к минимуму.

Перед тем как мы обсудим базис комбинаторного тестирования, необходимо понять, что такое **перестановки**. Перестановка — это возможное расположение **множества** объектов. В него не добавляются и из него не удаляются элементы, только изменяется их порядок. Это можно сравнить с тасованием колоды карт; каждый раз при перемешивании карт возникает очередная перестановка, т. к. изменяется порядок, но карты при этом не добавляются и не удаляются. В качестве примера рассмотрим массив `X`, содержащий следующие значения: `[1, 12, 4]`. Список возможных перестановок выглядит следующим образом:

```
1, 12, 4
1, 4, 12
12, 1, 4
12, 4, 1
4, 1, 12
4, 12, 1
```

Количество возможных перестановок набора равняется **факториалу** r, или $r!$, или $r \cdot (r-1) \cdot (r-2) \cdot \cdots \cdot 1$. Например, $3! = 3 \cdot 2 \cdot 1 = 6$, а $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.

Также можно выяснить, сколько можно получить различных упорядоченных подмножеств.

Предположим, что r и n являются положительными целыми значениями, количество r перестановок заданного набора n определяется следующей формулой:

$$P(n,r) = \frac{n!}{(n-r)!}.$$

Будет полезным изучить эту тему на примере и не только понять, как всё считается, но и как это может быть полезно в реальном тестировании. Давайте предположим, что мы тестируем игру про волейбол "Черепаха", в которой любые две из четырех черепах будут играть друг против друга. Имена черепах — Алан, Боб, Шарлин и Дарлин. Мы хотим убедиться, что для любого заданного матча игра будет работать правильно, и для нас также важен порядок. То есть если Боб играет с Аланом, и Боб является подающим, то это будет считаться другой игрой по отношению к той, в которой они играют вместе, но Алан подает первым.

Здесь будет $P(4, 2)$ возможных матчей (т. е. две перестановки). Помещая значения в уравнение выше, мы увидим, то, что нам необходимо:

$$P(4, 2) = 4! / ((4 - 2)!) =$$
$$= 24 / 2! =$$
$$= 24 / 2 =$$
$$= 12 \text{ матчей.}$$

То есть:

1. Алан/Боб.
2. Алан/Шарлин.
3. Алан/Дарлин.
4. Боб/Алан.
5. Боб/Шарлин.
6. Боб/Дарлин.
7. Шарлин/Алан.
8. Шарлин/Боб.
9. Шарлин/Дарлин.
10. Дарлин/Алан.
11. Дарлин/Боб.
12. Дарлин/Шарлин.

Помните: поскольку мы ищем перестановки, необходимо проверить, что каждая черепаха играет против любой другой черепахи.

Как вы уже, наверное, догадались из названия, т. к. мы обсуждаем комбинаторное тестирование, нас будет интересовать проверка **комбинаций** набора элементов, а не перестановок. Комбинация является неким *неупорядоченным* подмножеством множества. Используя подобную номенклатуру, как в случае с перестановками, мы также можем обратиться к ***r*-й комбинации** набора, чтобы указать комбинации подмножеств. Обычно нас интересуют комбинации меньшего размера, чем исходный набор, т. к. *r*-я комбинация набора, где *r* равняется размеру набора, является самим набором.

Количество комбинаций *r* заданного размера *n* называется **биномиальным коэффициентом**. Он определяется следующей формулой:

$$C(n, r) = \frac{n!}{r! \cdot (n - r)!}.$$

Например, количество возможных тройных комбинаций набора, состоящего из пяти элементов:

$$C(5, 3) =$$
$$= 5! / (3! \cdot (5 - 3)!) =$$
$$= 120 / (6 \cdot 2) =$$
$$= 10 \text{ (тройных комбинаций).}$$

Функция $C(n, r)$ часто описывается как "n выбирает r". Например, "10 выбирает 4" будет означать, сколько четверных комбинаций существует в наборе из 10 элементов (210).

Давайте вернемся к нашей волейбольной игре с четырьмя черепахами. В этом случае, однако, нас не будет беспокоить, какая черепаха подает первой — когда Алан в игре с Бобом подает первым равносильно случаю, когда первым подает Боб. Для того чтобы определить количество тестов, необходимых для тестирования, нам нужно определить количество двойных комбинаций для набора из четырех элементов, или $C(4, 2)$, или 6 тестов.

1. Алан/Боб.
2. Алан/Шарлин.
3. Алан/Дарлин.
4. Боб/Шарлин.
5. Боб/Дарлин.
6. Шарлин/Дарлин.

Идея с факториалом становится понятной, когда вы рассматриваете возможные результаты. Обратите внимание, что наша первая черепаха, Алан, должна взаимодействовать с тремя другими черепахами (всеми остальными черепахами). Вторая черепаха, Боб, должна взаимодействовать с двумя черепахами, т. к. уже существует пара, где она с Бобом. Когда мы переходим к Шарлин, мы должны взаимодействовать только с одной черепахой, т. к. Шарлин уже взаимодействует с Аланом и Бобом. В конце концов, Дарлин уже взаимодействует со всеми другими черепахами, поэтому дополнительных тестов для нее добавлять не нужно.

Обратите внимание, что для проверки всех возможных комбинаций требуется меньше тестов, чем для перестановок. Оказывается, что число комбинаций, необходимых для полного тестирования, будет расти сравнительно медленно с увеличением n. Пользуясь преимуществом концепций комбинаторики, мы сможем значительно сократить количество тестов, которые нам придется выполнить для получения представления о качестве конкретной программы.

Перестановки и комбинации являются частью области, известной как *дискретная математика*, в которой имеется множество приложений для разработки ПО в целом. Большинство специалистов в области компьютерных наук пройдут хотя бы один курс по дискретной математике в течение своей карьеры.

17.2. Попарное тестирование

Теперь, когда мы понимаем базовые концепции комбинаторики, давайте применим их при разработке плана попарного тестирования. Для каждой из этих пар мы хотим получить полную таблицу истинности из двух элементов, охватывающую все возможности. Например, чтобы гарантировать, что мы тестируем все возможные комбинации полужирного и курсива, можно составить следующую таблицу истинности:

```
Полужирный | Курсив
-----------+-------
ложь       | ложь
ложь       | истина
истина     | ложь
истина     | истина
```

Теперь нам надо убедиться, что наши тесты охватывают все возможные комбинации полужирного и курсива. Мы видим, что Test 1 дает ложь/ложь, Test 4 дает ложь/истину, Test 5 дает истину/ложь, Test 8 дает истину/истину:

```
   Полужирный | Курсив | Подчеркнутый
   -----------+-------+--------------
1. ложь       | ложь   | ложь         <--
2. ложь       | ложь   | истина
3. ложь       | истина | ложь
4. ложь       | истина | истина       <--
5. истина     | ложь   | ложь         <--
6. истина     | ложь   | истина
7. истина     | истина | ложь
8. истина     | истина | истина       <--
```

Обратите внимание, что когда мы работаем с этими тестами, мы в то же время проверяем другие комбинации и тесты! Например, запуск Test 1 также дает нам комбинацию ложь/ложь для полужирного и подчеркнутого, и комбинацию ложь/ложь для курсива и подчеркнутого. Фактически мы получаем несколько тестов по цене одного. Можете показать это людям, которые говорят, что не существует ничего бесплатного.

Следующим шагом мы должны обеспечить, чтобы для других пар переменных были покрыты все их комбинации, и для этого используем технику, подобную той, которую мы применяли для полужирного и курсива. В отличие от случайного выбора любого тест-кейса, который соответствует нашим критериям (как мы делали в первой выборке), мы хотим проверить, что не тестируем комбинацию в тесте, который уже выполняется. Это необходимо, чтобы избежать дублирования усилий; если Test 1 уже проверяет ложь/ложь для полужирного/подчеркнутого, зачем тогда добавлять Test 3? Давайте осуществим проверку для полужирного/подчеркнутого, держа в голове, что мы хотим при возможности задействовать уже существующие тесты. Test 1 дает нам комбинацию ложь/ложь; Test 4 дает ложь/истина; Test 5 дает истина/ложь и Test 8 дает истина/истина. Ого! Все эти комбинации уже были покрыты тест-кейсам, которые мы собирались делать. Теперь мы протестировали все комбинации полужирный/курсив и полужирный/подчеркнутый при помощи всего четырех тестов!

В итоге нам необходимо проверить последнюю комбинацию — курсив/подчеркнутый. Test 1 дает нам ложь/ложь, а Test 4 — истина/истина. Впрочем, к сожалению, ложь/истина и истина/ложь пока не покрыты ни одним из планируемых к использованию тестов, и поэтому нам потребуется добавить еще тесты. На самом деле, т. к. имеются две различные комбинации, и эти комбинации являются взаи-

моисключающими, нам нужно добавить два теста. Давайте добавим Test 2 для ложь/истина и Test 3 для истина/ложь. Наш окончательный тест-план будет выглядеть так:

```
   Полужирный | Курсив | Подчеркнутый
   -----------+--------+--------------
1. ложь       | ложь   | ложь
2. ложь       | ложь   | истина
3. ложь       | истина | ложь
4. ложь       | истина | истина
5. истина     | ложь   | ложь
8. истина     | истина | истина
```

Мы выполнили требование нашего строгого менеджера и уменьшили количество тестов на 25%. Мы использовали **покрывающий массив** (covering array), чтобы определить кратчайший путь проверки всех двусторонних (попарных) взаимодействий. Покрывающий массив является массивом, представляющим комбинации тестов, охватывающих все возможные *n*-сторонние взаимодействия (в данном случае двухсторонние). Обратите внимание, что поскольку исходный массив, который охватывал все возможные значения таблицы истинности, также покрывал все возможные комбинации этих значений, то технически это тоже покрывающий массив. Однако в общем случае этот термин относится к уменьшенной версии массива, который по-прежнему содержит все тесты, необходимые для проверки различных *n*-сторонних взаимодействий.

Созданный нами покрывающий массив на самом деле не является оптимальной конфигурацией для тестирования всех комбинаций, т. к. мы использовали наивный алгоритм и не проверяли другие возможности. Вероятно, отбирая различные тесты, при наличии выбора мы могли бы охватить то же количество комбинаций с меньшим количеством тест-кейсов. Подбор вручную является чрезвычайно трудоемким занятием, особенно с увеличением количества взаимодействий и переменных. По этой причине шаблоны комбинаторного тестирования часто создаются с использованием специальных программных инструментов. Хотя рассмотрение конкретных алгоритмов и эвристики находится за рамками данной книги, эти инструменты сгенерируют для вас покрывающие массивы быстро и автоматически. Эти комбинаторные генераторы тестов, такие как бесплатный Advanced Combinatorial Testing System Национального института стандартов и технологии (National Institute of Standards and Technology, NIST), оказываются бесценными, когда необходимо сгенерировать покрывающие массивы для нетривиального числа переменных.

Используя алгоритм IPOG в программе NIST ACTS, я смог сгенерировать следующий оптимальный покрывающий массив. В нем всего четыре теста, и практически невозможно смоделировать более эффективную ситуацию, поскольку каждая попарная таблица истинности сама по себе потребует четырех различных тестов. Еще более важно, что мы превзошли ожидания менеджера и сократили число тестов на 50%, а не на требуемые 25%, и при этом по-прежнему тестируем все попарные взаимодействия:

```
   Полужирный | Курсив | Подчеркнутый
   -----------+--------+--------------
1. ложь       | ложь   | ложь
4. ложь       | истина | истина
6. истина     | ложь   | истина
7. истина     | истина | ложь
```

Давайте проверим, что все возможные пары существуют в тест-плане:

1. *Полужирный/курсив*: ложь/ложь обрабатывается в тест-кейсе 1, ложь/истина в тест-кейсе 4, истина/ложь в тест-кейсе 6 и истина/истина в тест-кейсе 7.
2. *Курсив/подчеркнутый*: ложь/ложь обрабатывается в тест-кейсе 1, ложь/истина в тест-кейсе 6, истина/ложь в тест-кейсе 7 и истина/истина в тест-кейсе 4.
3. *Полужирный/подчеркнутый*: ложь/ложь обрабатывается в тест-кейсе 1, ложь/истина в тест-кейсе 4, истина/ложь в тест-кейсе 7 и истина/истина в тест-кейсе 6.

Хотя данный пример реализован с булевыми переменными, можно использовать любые конечные переменные. Если у переменной возможно бесконечное количество значений (скажем, переменная длина строки), вы можете задать определенное количество значений (т. е. "a", "abcde" или "abcdefghijklmnop"). При этом необходимо сперва подумать о различных классах эквивалентности, если таковые имеются, и убедиться, что у вас имеется значение для каждого класса эквивалентности. Также вам следует проверить различные граничные случаи, особенно те, которые могут вызвать проблемы при совместном взаимодействии с другими переменными. Предположим, что в нашем предыдущем примере мы хотели бы проверить не символ, а слово. Мы может добавить различные варианты слов, которые необходимо протестировать. Начнем с "a" (одинарный символ) и "bird" (простое слово). Наш сгенерированный покрывающий массив будет похож на предыдущий, просто используются "a" и "bird" в качестве значений переменной `word` вместо "истина" и "ложь", используемых для других переменных:

```
   word   | Полужирный | Курсив | Подчеркнутый
   -------+------------+--------+-------------
1. "a"    | истина     | ложь   | ложь
2. "a"    | ложь       | истина | истина
3. "bird" | истина     | истина | ложь
4. "bird" | ложь       | ложь   | истина
5. "bird" | истина     | ложь   | истина
6. "a"    | ложь       | ложь   | ложь
```

Обратите внимание, что мы по-прежнему проверяем все пары. Давайте проверим пару word/полужирный, чтобы убедиться, что это так. В Test 1 мы проверяем слово "a" с включенным полужирным шрифтом. В Test 2 мы проверяем слово "a" с выключенным полужирным шрифтом. В Test 3 — слово "bird" с включенным полужирным шрифтом и в Test 4 — слово "bird" с выключенным полужирным шрифтом. А вдруг нас обеспокоит, что спецсимволы вызовут проблемы с форматированием? Нам необходимо добавить третий вариант для переменной `word`, что

определенно возможно, хотя до сего момента у переменных было только два возможных значения:

```
     word | Полужирный | Курсив | Подчеркнутый
    ------+------------+--------+-------------
1. "a"    | истина     | ложь   | ложь
2. "a"    | ложь       | истина | истина
3. "bird" | истина     | истина | ложь
4. "bird" | ложь       | ложь   | истина
5. "!@#$" | истина     | истина | истина
6. "!@#$" | ложь       | ложь   | ложь
```

В качестве примечания отметим, что сейчас мы тестируем еще больше парных комбинаций, а у нас по-прежнему осталось шесть тестов.

17.3. *n*-сторонние взаимодействия

Хотя многие ошибки обнаруживаются при проверке всех парных взаимодействий, часто нам хотелось бы пойти дальше и проверить ошибки в трехстороннем, четырехстороннем или даже в большем количестве взаимодействий. Для этого работает та же теория: должны быть созданы тесты, охватывающие всю таблицу истинности для каждого трехстороннего взаимодействия переменных. Подобно тому, как мы проверяли все четыре комбинации значений истина/ложь при тестировании каждого двустороннего взаимодействия в первом примере, мы проверим, что тестируются все восемь комбинаций значений для каждого трехстороннего взаимодействия.

Давайте расширим количество переменных форматирования в нашей системе и добавим в качестве возможного написания текста верхний индекс (или надстрочный) и перечеркивание. Для того чтобы всеобъемлюще протестировать все варианты, нам потребуется 2^5, или 32 теста. После создания покрывающего массива для всех парных взаимодействий мы получим тест-план для шести случаев. И снова обратите внимание, что, несмотря на то, что в данном случае мы тестируем еще больше переменных, количество тестов осталось прежним. В данном случае мы запускаем всего лишь 18,75% тестов, которые потребовались бы нам для исчерпывающего тестирования, но мы по-прежнему тестируем все парные взаимодействия:

```
   Полужирный | Курсив | Подчеркнутый | Надстрочный | Перечеркнутый
   -----------+--------+--------------+-------------+--------------
1. истина     | истина | ложь         | ложь        | ложь
2. истина     | ложь   | истина       | истина      | истина
3. ложь       | истина | истина       | ложь        | истина
4. ложь       | ложь   | ложь         | истина      | ложь
5. ложь       | истина | ложь         | истина      | истина
6. ложь       | ложь   | истина       | ложь        | ложь
```

Тестирование всех возможных трехсторонних взаимодействий потребует больше тестов. Если вы задумались об этом, значит, это логически необходимо. Так как существует восемь возможных комбинаций для каждого трехстороннего взаимо-

действия, вполне возможно покрыть любую комбинацию с использованием всего шести тестов.

	Полужирный	Курсив	Подчеркнутый	Надстрочный	Перечеркнутый
1.	истина	истина	истина	истина	истина
2.	истина	истина	ложь	ложь	ложь
3.	истина	ложь	истина	ложь	истина
4.	истина	ложь	ложь	истина	ложь
5.	ложь	истина	истина	ложь	ложь
6.	ложь	истина	ложь	истина	истина
7.	ложь	ложь	истина	истина	ложь
8.	ложь	ложь	ложь	ложь	истина
9.	ложь	ложь	истина	истина	истина
10.	истина	истина	ложь	ложь	истина
11.	истина	истина	истина	истина	ложь
12.	ложь	ложь	ложь	ложь	ложь

Давайте рассмотрим заданное трехстороннее взаимодействие для полужирного/курсива/подчеркнутого и перепроверим, что мы тестируем все возможности. Ложь/ложь/ложь покрывается Test 12; ложь/ложь/истина покрывается Test 9; ложь/истина/ложь покрывается Test 6; ложь/истина/истина покрывается Test 5; истина/ложь/ложь покрывается Test 4; истина/ложь/истина покрывается Test 3; истина/истина/ложь покрывается Test 10; истина/истина/истина покрывается Test 1. Обратите внимание, что имеется восемь комбинаций и десять тестов, поэтому имеем повторы (например, истина/истина/истина покрывается Test 1 и Test 11).

Число взаимодействий можно увеличивать, насколько вам хочется, хотя, если вы планируете тестирование *n*-сторонних взаимодействий, где *n* равняется количеству имеющихся переменных, вы просто займетесь исчерпывающим тестированием. Согласно эмпирическим исследованиям NIST максимальное количество вызывавших ошибку взаимодействий равнялось шести, поэтому проверка с большим значением взаимодействий во многих ситуациях окажется чрезмерной.

17.4. Работа с большими наборами переменных

Комбинаторное тестирование неплохо справляется с небольшими наборами данных, экономя нам большой процент времени путем снижения количества необходимых тестов. Однако снижение количества тестов с 32 до 12 не так впечатляет; в конце концов, и 32 теста, вероятно, могут пройти за приемлемое время. А как комбинаторное тестирование работает для больших наборов переменных и возможных значений?

Ответ — невероятно хорошо. Давайте предположим, что у нас вместо пяти булевых переменных имеется пятьдесят. Для того чтобы всеобъемлюще протестировать все возможные комбинации, потребуется запустить 2^{50} (1 125 899 906 842 624) тестов.

Это больше нониллиона тестов — вы можете выполнять по тесту в секунду до конца жизни, и вам все равно не хватит времени. Однако если вас устраивает проверка всех двусторонних взаимодействий, мы можете снизить это число до 14 тестов! Эта экономия на тестах воистину непостижима в числах процентов — речь идет о множестве порядков. То, что раньше казалось пугающим, теперь становится легко достижимым. Увеличение количества взаимодействий увеличивает количество тестов незначительно: например, тестирование всех трехсторонних взаимодействий потребует сорока тестов, а тестирование всех четырехсторонних взаимодействий потребует всего сотню тестов. Более того, NIST ACTS сгенерировал соответствующие тест-планы на моем маломощном ноутбуке менее чем за несколько секунд.

Вы можете увидеть, что сублинейно растет не только количество необходимых тестов; чем больше у вас переменных, тем больший процент времени вы сэкономите благодаря комбинаторному тестированию. В начале этой книги мы говорили о том, что для многих программ исчерпывающее тестирование практически невозможно. Применение комбинаторного тестирования является одним из способов облегчить эту проблему. Используя небольшой процент усилий, необходимых для исчерпывающего тестирования, мы можем найти подавляющее большинство дефектов, которые могли бы быть найдены во время его проведения.

Глава 18

Стохастическое тестирование и тестирование на основе свойств

Термин "стохастический" происходит от греческого слова στοχαστικός, которое является формой στοχάζεσθαι, означающего "направляйся к" или "предполагай". Это хорошая этимология для **стохастического тестирования**, использующего случайные процессы, которые могут быть проанализированы при помощи статистики, но не предсказаны точно. На первый взгляд может показаться смешным использовать случайность в тестировании ПО; в конце концов, разве фундаментальная концепция тестирования не заключается в определении ожидаемого поведения и проверке того, что оно равняется наблюдаемому поведению? Если вы не знаете, что подается на вход, как вы узнаете, какой должен быть ожидаемый выход?

Ответ заключается в том, что существуют ожидаемые поведения и свойства, про которые вы знаете, что они имеются в системе, и которые не зависят от того, что вы подаете на вход. Например, вне зависимости от того, какой код передается компилятору, вы ожидаете, что компилятор не выйдет из строя. Он может выдать сообщение о невозможности компиляции. Он может сгенерировать исполняемый файл. Этот исполняемый файл может запуститься, а может и нет. Вы ожидаете, что в системе нет ошибки сегментации. Таким образом, вы по-прежнему можете запускать тесты, где ожидаемым поведением для любых входных данных будет "система не выходит из строя".

Предоставляя системе метод использования случайных данных в качестве входных данных, вы также снижаете стоимость тестирования. Вам больше не нужно продумывать множество конкретных тест-кейсов, а затем тщательно записывать их или программировать. Вместо этого вы просто связываете некий генератор случайных чисел и некий способ генерирования тестовых данных на его основе, а ваш компьютер выполняет всю работу по генерации тест-кейсов. Несмотря на то что генератор случайных чисел может быть не так хорош, как специалист по тестированию при разработке граничных случаев, разделении классов эквивалентности и т. п., он часто обнаруживает множество проблем просто из-за способности сгенерировать огромное количество тестов и быстро их выполнить.

Стохастическое тестирование часто называется **обезьяньим тестированием** (monkey testing) по аналогии с обезьяной, стучащей по кнопкам клавиатуры. Однако "стохастическое тестирование" звучит гораздо более впечатляюще, особенно

если вы разговариваете с вашим менеджером или пишете книгу о тестировании программного обеспечения.

18.1. Бесконечные обезьяны и бесконечные пишущие машинки

Существует старая притча об обезьянах и пишущих машинках, которые при обычных условиях не взаимодействуют друг с другом. Притча гласит, что миллион обезьян при наличии бесконечного времени однажды напишут работы Шекспира (предполагается, что обезьяны в данном случае бессмертны). Стохастическое тестирование следует подобному принципу — большой набор случайных входных данных (миллион обезьян, стучащих по кнопкам) и достаточно большой период времени позволят обнаружить дефекты. В данной аналогии наш тестировщик является Уильямом Шекспиром (не принимайте это близко к сердцу). Тестировщик определенно может написать работы Шекспира быстрее, чем миллион обезьян. Однако обезьяны (как и компьютерный генератор случайных чисел) гораздо дешевле, чем реанимация зомби Уильяма Шекспира. Даже если генератор случайных чисел не такой хороший автор тестов, как вы (или Шекспир), благодаря их большому количеству он неизбежно столкнется с многочисленными интересными граничными случаями и, возможно, обнаружит дефекты.

Так как вы — а более точно, стохастическая система тестирования — можете не знать, каким должно быть ожидаемое поведение для заданных входных данных, вам необходимо проверять свойства системы. На уровне юнит-тестирования, где вы проверяете отдельные методы, это называется **тестированием на основе свойств** (property-based testing).

18.2. Тестирование на основе свойств

Давайте еще раз предположим, что мы тестируем нашу функцию сортировки `billSort`. Как вы помните, предполагается, что она работает в 20 раз быстрее, чем любой другой алгоритм сортировки. Однако существуют вопросы о правильности ее работы, поэтому вам поручили проверить, как она работает во всех случаях. Какой вид входных данных использовать при тестировании? Предположим, что сигнатура метода выглядит так:

```
public int[] billSort(int[] arrToSort) {
    ...
}
```

Нам бы определенно хотелось передавать широкий диапазон значений, которые задействуют различные основные, граничные и угловые случаи. Массивы из одного значения. Пустые массивы. Массивы с отрицательными числами. Массивы, которые уже отсортированы. Массивы, которые отсортированы наоборот необходимой сортировке (по увеличению вместо уменьшения, и наоборот). Очень большие массивы. Массивы, которые содержат разные значения, и массивы, состоящие из одно-

го повторяющегося значения. И это не принимая во внимание, что Java является статически типизированным языком, и поэтому нам не нужно беспокоиться, что произойдет, если массив содержит строки, ссылки на другие массивы, комплексные числа и множество других возможных вариантов. Если бы сортировка была реализована на таком динамически типизированном языке, как Ruby, тогда у нас бы добавилось немало забот. Однако, просто рассматривая приведенный выше метод Java, давайте подумаем о различных входных данных и соответствующим им выходным значениям для этого метода:

```
[] => []
[1] => [1]
[1, 2, 3, 4, 5] => [1, 2, 3, 4, 5]
[5, 4, 3, 2, 1] => [1, 2, 3, 4, 5]
[0, 0, 0, 0] => [0, 0, 0, 0]
[9, 3, 1, 2] => [1, 2, 3, 9]
[-9, 9, -4, 4] => [-9, -4, 4, 9]
[3, 3, 3, 2, 1, 1, 1] => [1, 1, 1, 2, 3, 3, 3]
[-1, -2, -3] => [-3, -2, -1]
[1000000, 10000, 100, 10, 1] => [1, 10, 100, 10000, 1000000]
```

Даже без необходимости наморщить лоб мы получили огромное количество возможных комбинаций для тестирования. И это еще малая часть. Будет огромное количество классов эквивалентности, которые можно протестировать, и это не считая того, что возможна проблема, скажем, с каким-то конкретным числом — например, сортировка может не работать, если среди чисел присутствует 5. Написание тестов для всех этих различных входных данных окажется утомительным и подверженным ошибкам. Как можно избежать написания такого большого количества тестов?

18.2.1. Взбираясь по лестнице абстракции

Почему бы не запрыгнуть вверх на ступеньку лестницы абстракции и вместо того, чтобы думать о конкретных значениях входных и выходных данных, подумать о *свойствах*, которые вы ожидаете от вашего входа и выхода? Таким образом, вы не должны рассматривать каждый отдельный тест. Вы можете позволить компьютеру знать, что ожидаете определенные свойства у всех выходных данных, и какой тип значений вы ожидаете у входных данных, а компьютер напишет и выполнит тесты за вас.

Например, какие виды свойств были у всех правильных выходных значений метода `billSort()` по отношению ко входным значениям? Их было несколько. Эти свойства должны сохраняться для всех сортированных списков. Таким образом, они называются **инвариантами**.

Приведем некоторые из возможных инвариантов для функции сортировки:

1. В выходном массиве содержится столько же элементов, сколько и во входном.
2. Каждое значение выходного массива соответствует значению входного массива.

3. Значение каждого последующего элемента в выходном массиве больше или равно предыдущему значению.
4. Ни один из не присутствующих во входном массиве элементов также не находится в выходном массиве.
5. Функция идемпотентная; т. е. вне зависимости от того, сколько раз эта функция вызывается, возвращаться должны те же выходные данные. Если я запущу метод сортировки на списке, а затем снова запущу сортировку на полученных данных, то второй вызов сортировки должен создать тот же выходной массив, что и после первого запуска. Обратите внимание, что было бы труднее тестировать функцию, обратную идемпотентной (**неидемпотентную** функцию), т. к. в этом случае выходные данные могут изменяться, а могут не изменяться (например, предположим, что функция увеличивает значение и возвращает это значение по модулю 6; каждый раз при вызове этой функции с множителем 6 она вернет одно и то же значение).
6. Функция является чистой; если запустить ее дважды с одинаковым входным массивом значений, то на выходе всегда должен быть одинаковый выходной массив. Каждый раз, когда я вызываю сортировку для списка `[3, 2, 1]`, всегда будет возвращаться `[1, 2, 3]`. Неважно, будет ли луна в фазе роста или убывания, или какие установлены значения глобальных переменных, для одного и того же входного массива будет возвращаться одинаковое выходное значение.

Теперь, когда у нас есть некоторые свойства, которые мы ожидаем от *любого* выхода метода `billSort()`, мы может позволить компьютеру выполнять основную работу по созданию массивов данных, передаче их в наш метод и последующей проверке, соответствует ли полученный выходной массив всем тем свойствам, что мы установили. Если выходной массив не соответствует одному из инвариантов, мы сообщаем об ошибке тестировщику. Генерация выходных данных, которые не соответствуют определенному инварианту, называется **фальсификацией инварианта**.

Существует множество библиотек для Java, которые выполняют тестирование на основе свойств, но стандарта такого тестирования нет. Тестирование на основе свойств гораздо более популярно в мире функционального программирования, причем такие программы, как QuickCheck для Haskell, используются чаще, чем стандартные юнит-тесты. Фактически концепция автоматизированного тестирования на основе свойств исходит из функционального мира и сообщества Haskell в частности. Для получения дополнительной информации обратитесь к "QuickCheck: Легкий инструмент для случайного тестирования программ на Haskell" Коэна Классена и Джона Хьюза (Koen Claessen, John Hughes "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs").

18.3. Умные, тупые, злые и хаотические обезьяны

Как было упомянуто выше, стохастическое тестирование часто называется обезьяньим тестированием. Но не так хорошо известно, что существует несколько видов обезьян, которые могут выполнять работу по тестированию за нас.

Давайте представим простую программу, которая принимает строковый аргумент от пользователя, пытается прочитать эту строку как арифметическое выражение и отображает его результат. Например, `"5 + 6"` даст `"11"`, а `"5 - (2 * 1) + 10"` даст `"13"`. Если программа не может обработать строку из-за неправильного арифметического выражения (например, `"$%--00"`), она напечатает сообщение "ERROR".

Тупое обезьянье тестирование — это отправка всего того ввода, который вы только можете придумать (или, как это чаще случается, который создает ваша программа генерации входных данных). `"Mfdsjbkfd"`, `"1 + 2"` и `"(*@()"` являются хорошими входными данными с точки зрения тупой обезьяны. Им нет никакого логического объяснения, просто много разных случайных входных данных. Оно может быть полезно для выявления граничных случаев, но не является хорошо сфокусированным. Помните, что мир возможных значений огромен. Шансы нахождения определенного дефекта при использовании тупого обезьяньего тестирования могут быть минимальными. Случайно мы можем найти хотя бы одно правильное арифметическое выражение, но каковы шансы, что это будет происходить на регулярной основе? Абсолютное большинство случайно сгенерированных входных данных приведет к одному поведению: простому отображению сообщения "ERROR". Вероятность того, что будут обнаружены общеизвестные и легко проверяемые ошибки, такие как целочисленное переполнение, исчезновение порядка или проблемы округления числа с плавающей запятой, минимальна.

В качестве более реального примера давайте предположим, что у вас имеется дефект, когда можно выполнить произвольный код JavaScript после ввода его в текстовое поле веб-приложения. Однако если этот код JavaScript не является синтаксически правильным, ничего плохого не случится. Представьте, сколько времени потребуется тупой обезьяне, чтобы случайным образом сгенерировать правильную строку JavaScript, которая сможет воспользоваться этой очевидной уязвимостью! Возможно, годы. И даже тогда это может быть код JavaScript, который не вызывает никаких заметных проблем, таких как комментарий или запись сообщения в консоль. Между тем даже начинающий тестировщик попытается ввести JavaScript в любое текстовое поле, которое сможет найти.

У тупого обезьяньего тестирования есть несколько очевидных недостатков, главный из которых заключается в том, что без указания направления большинство входных данных являются неправильными и неинтересными. Однако реализовать его и выполнить большое количество тестов можно очень быстро.

Умное обезьянье тестирование включает использование не случайного набора данных, а тех входных данных, которые может ввести пользователь. Например, представим, что вы тестируете программу-калькулятор, которая принимает строку из чисел и арифметических операторов и отображает результат. Разумно предположить, что обычный пользователь, скорее, будет вводить что-то вроде такого:

```
> 1 + 2
3
> 4 + + 6
ERROR
> 4 + 6
10
```

чем:

```
> jiwh0t34h803h8t32h8t3h8t23
ERROR
> aaaaaaaaaaaaa
ERROR
> 084_==wjw2933
ERROR
```

Хотя это предположение может не выполняться, если программой займутся малыши, в целом оно, скорее всего, верно. Таким образом, чтобы сосредоточить наши ресурсы тестирования на выявлении дефектов, мы можем использовать умное обезьянье тестирование, чтобы действовать как пользователь. Однако поскольку данное тестирование автоматизируется, оно сможет работать гораздо быстрее и на гораздо большем количестве возможных входных данных, чем выполняемое пользователем ручное тестирование.

Создание теста умной обезьяны может быть сложным, потому что вам нужно не только понимать, как именно пользователи будут работать с приложением, но и также разработать модель и генератор для этого. Однако то, что умная обезьяна найдет дефект в тестируемой системе, является более вероятным.

Злое обезьянье тестирование симулирует поведение злонамеренного пользователя, который активно пытается повредить вашу систему. Это может быть реализовано через отправку чрезвычайно длинных строк, возможные инъекционные атаки, неправильно сформированные данные, неподдерживаемые символы или прочие входные данные, которые задумывались с целью вызвать разрушения в вашей системе. В современном связанном сетями мире системы почти всегда находятся под атакой, даже если они подключились к Интернету всего на несколько миллисекунд. Гораздо лучше иметь в наличии злую обезьяну, определяющую, уязвима ли система, нежели ждать, когда за нее это сделает реальный злоумышленник!

Давайте предположим, что мы сохраняем все вводимые данные нашей арифметической программы в базе данных. Тест злой обезьяны может проверить, способна ли программа каким-то образом перезаписать или изменить эти данные путем передачи ей вредоносного запроса SQL или неких символов, которые могут быть интерпретированы как пустые (null), или слишком длинной строки, способной вызвать переполнение буфера.

```
> '); DELETE FROM entries; --"
ERROR
> \000\000\000
ERROR
> Eh bien, mon prince. Gênes et Lucques ne sont plus que des apanages, des поместья,
de la famille Buonaparte...
ERROR
```

По последней строке можно предположить, что злой обезьяной был фактически введен весь текст "Войны и мира" Льва Толстого. Я хотел добавить его, чтобы увеличить количество слов в моей книге, но решил отказаться от этой идеи, чтобы

уменьшить вес печатной версии. Это только несколько примеров того, как злое обезьянье тестирование может искать дефекты в системе. Для того чтобы узнать больше о тестировании безопасности систем, обратитесь к *главе 20 "Тестирование безопасности"*.

Пожалуй, лучшим видом обезьян является **хаотическая обезьяна** (chaos monkey). Это инструмент компании Netflix, который для имитации возникновения проблем случайным образом выключает запущенные серверы, на которых работает их система. У любой большой системы серверы будут регулярно отключаться, и в определенный момент времени некий процент систем будет недоступен. Хаотическое обезьянье тестирование подтверждает, что система в целом будет способна эффективно работать, даже если отдельные машины не отвечают.

Однако вам не обязательно использовать официальный инструмент Chaos Monkey для этого вида тестирования. Подумайте, что может пойти не так с состоящей из множества серверов системой, и смоделируйте это. Что произойдет в случае изменения топологии сети? Останется ли система активной, если кто-то выдернет кабель питания или сети? Что произойдет, если задержка увеличится до нескольких секунд? В распределенной системе зреют проблемы. Проверка того, что вы можете справиться с ними сейчас, позволит вам подготовиться к тому, когда они произойдут в реальности. В конце концов, лучший способ избежать проблемы — это неоднократно вызывать ее; вскоре у вас будут автоматизированные процедуры, созданные с целью облегчить проблемы или гарантировать, что они не произойдут.

18.4. Мутационное тестирование

Еще одно использование случайности в тестировании позволит нам протестировать наши тесты! В **мутационном тестировании** мы изменяем сам код случайным образом (конечно, сохраняя копию исходного кода для последующего к ней возвращения). Таким образом, мы осуществляем **посев** тестируемой системы дефектами. Посев преднамеренно добавляет дефекты в систему, чтобы определить, способен ли наш процесс тестирования поймать их. Это может нам дать общее представление о качестве наших тестов. Например, если в систему намеренно добавлены десять различных дефектов и команда контроля качества обнаруживает все десять из них, то и другие оценки качества с большой вероятностью будут аккуратными. Если команда контроля качества обнаруживает только один или вообще ни одного из десяти посеянных дефектов, то мы с полным основанием можем думать, что существует множество *реальных* дефектов, которые не обнаруживаются. Это, безусловно, поставит под сомнение любые гарантии качества, которые команда предоставила в отношении качества тестируемой системы.

После каждого случайного изменения кода запускается тестовый набор или его часть, связанная с измененным кодом. Если наш тестовый набор полностью покрывает модифицированный код, по крайней мере один тест должен завершиться неуспешно. Если этого не происходит ни с одним из тестов, то вполне возможно, что наше тестовое покрытие неполное. Существует вероятность, что мутация оказалась

полностью доброкачественной; скажем, она изменила значение по умолчанию переменной, которое затем сразу же было перезаписано. Также существует вероятность, что модификация внесла изменения, которые при работе окажутся не видны.

Давайте проработаем пример мутационного тестирования. Предположим, что у нас есть метод, который на основе длины шеи животного пытается определить вид этого животного.

```
public class Guess {
    public static String animalType (int neckLength) {
        String toReturn = "UNKNOWN";
        if (neckLength < 10) {
            toReturn = "Rhinoceros";
        } else if (neckLength < 20) {
            toReturn = "Hippopotamus";
        } else {
            toReturn = "Giraffe";
        }
        return toReturn;
    }
}
```

Наши тест-кейсы проверяют по одному значению из каждого класса эквивалентности.

```
@Test
public void animalTypeRhinoceros() {
    assertEquals("Rhinoceros", Guess.animalType(5);
}
@Test
public void animalTypeHippopotamus() {
    assertEquals("Hippopotamus", Guess.animalType(15);
}
@Test
public void animalTypeGiraffe() {
    assertEquals("Giraffe", Guess.animalType(25);
}
```

Сейчас, если вы помните нашу главу о юнит-тестировании, это не очень хорошее покрытие. Конечно, это не самое худшее юнит-тестирование метода, которое я видел. Мутационное покрытие может дать нам лучшее представление об общем качестве наших тестов, чем простая оценка с точки зрения покрытия кода. Давайте рассмотрим несколько мутаций, которые вызовут сбой, показывая нам сильные стороны юнит-тестов и то, как можно заставить их завершиться неуспешно.

```
public class Guess {
    public static String animalType(int neckLength) {
        String toReturn = "UNKNOWN";
        // Значение изменено с 10 на 1
        if (neckLength < 1) {
            toReturn = "Rhinoceros";
```

```
        } else if (neckLength < 20) {
            toReturn = "Hippopotamus";
        } else {
            toReturn = "Giraffe";
        }
        return toReturn;
    }
}
```

В этом случае мы изменили условную проверку с `neckLenght < 10` на `neckLenght < 1`. Это закономерно приведет к сбою теста animalTypeRhinoceros, т. к. наша мутированная версия метода вернет "Hippopotamus" для аргумента `neckLength`, равного 5.

```
public class Guess {
    public static String animalType(int neckLength) {
        String toReturn = "UNKNOWN";
         // Условие "меньше, чем" изменено на "больше, чем"
        if (neckLength > 10) {
            toReturn = "Rhinoceros";
        } else if (neckLength < 20) {
            toReturn = "Hippopotamus";
        } else {
            toReturn = "Giraffe";
        }
        return toReturn;
    }
}
```

В данном случае *все* юнит-тесты завершатся сбоем: первый покажет, что вместо ожидаемой пометки "Rhinoceros" получается "Giraffe", а два других теста дадут то, что животное является "Rhinoceros". Хорошая работа, юнит-тесты!

Более интересны случаи, которые могут помочь нашей кодовой базе, когда мутации *не приводят* к сбою тестов. В общем случае это означает, что наши методы не дают того покрытия метода, которого мы ожидаем. Различные ошибки в методе могут сейчас или в будущем проскользнуть через тестовый набор. Запомните, неудача юнит-теста в поиске мутации не означает, что существует проблема с кодом прямо сейчас. Код мог быть реализован совершенно правильно. Это сбой *юнит-тестов* в отлавливании возможной проблемы. Мы просто не сможем узнать с точки зрения тестирования, что метод не работает при определенных обстоятельствах.

Давайте посмотрим на пример мутации, которая не приведет к сбоям юнит-тестов, но окажет влияние на выполнение кода.

```
public class Guess {
    public static String animalType(int neckLength) {
        String toReturn = "UNKNOWN";
         // Значение было изменено с 10 на 8
        if (neckLength < 8) {
            toReturn = "Rhinoceros";
```

```
        } else if (neckLength < 20) {
            toReturn = "Hippopotamus";
        } else {
            toReturn = "Giraffe";
        }
        return toReturn;
    }
}
```

В этом случае все наши юнит-тесты пройдут: животное с длиной шеи, равной 5, будет идентифицировано как носорог (rhinoceros), с длиной шеи, равной 15, — как бегемот (hippopotamus), а с длиной шеи, равной 25, — как жираф (giraffe). Но, к сожалению, функциональность метода определенно изменилась! Если кто-то вызовет этот мутированный метод, используя в качестве аргумента `neckLength`, равный 9, животное будет определено, как бегемот, хотя это носорог. Это показывает нам важность проверки граничных значений между классами эквивалентности. Если бы мы проверили все эти границы, тогда один из наших юнит-тестов завершился бы сбоем при изменении этих значений.

Как мы говорили, некоторые мутации могут не вызывать сбоев тестов, но при этом быть доброкачественными. Зачастую это означает, что измененный код является излишним или что он реализован неправильно, но на его выполнение это не оказывает влияния.

```
public class Guess {
    public static String animalType(int neckLength) {
        String toReturn = "AMBER";
         // Значение строки изменено с "UNKNOWN" на "AMBER"
        if (neckLength < 10) {
            toReturn = "Rhinoceros";
        } else if (neckLength < 20) {
            toReturn = "Hippopotamus";
        } else {
            toReturn = "Giraffe";
        }
        return toReturn;
    }
}
```

Это не вызовет никаких сбоев, т. к. значение `"UNKNOWN"` было просто заполнителем и ни для чего не использовалось. Это говорит нам о том, что, возможно, нам не требуется значение по умолчанию или что существует лучший способ реализации метода. Например, немного более эффективный способ реализации того же метода:

```
public class Guess {
    public static String animalType(int neckLength) {
        String toReturn = "Giraffe";
         // Значение по умолчанию теперь Giraffe, и не надо использовать
         // else для завершения
```

```
        if (neckLength < 10) {
            toReturn = "Rhinoceros";
        } else if (neckLength < 20) {
            toReturn = "Hippopotamus";
        }
        return toReturn;
    }
}
```

Тем не менее в этом коде мы забыли реализовать некий функционал. Возможно, мы намеревались возвращать это значение по умолчанию, если передано отрицательное значение длины шеи, но не удосужились прописать это в коде!

```
public class Guess {
    public static String animalType(int neckLength) {
        String toReturn = "UNKNOWN ";
         // Если передано неверное значение neckLength, возвращаем
         // значение по умолчанию
        if (neckLength < 0) {
            return toReturn;
        }
        if (neckLength < 10) {
            toReturn = "Rhinoceros";
        } else if (neckLength < 20) {
            toReturn = "Hippopotamus";
        } else {
            toReturn = "Giraffe";
        }
        return toReturn;
    }
}
```

Мутационное тестирование является важным способом оценки того, насколько хорошо ваши тесты тестируют ваш код. Тестировщик может легко пропустить класс эквивалентности, граничное значение или режим сбоя. Впрочем, когда обнаруживается мутация, которая не определяется вашими тестами, вы можете увидеть не только проблему в вашей стратегии тестирования, но и ту ошибку в коде, которая ее вызвала.

Глава 19

Тестирование производительности

А что такое тестирование производительности? Перед тем как мы определим, что такое "тестирование производительности", нам необходимо определить, что такое "производительность". Я полагаю, что к этому разделу книги все уже понимают, что означает "тестирование".

Большинство технически подкованных людей будет иметь представление о том, что означает производительность — системы являются "быстрыми", они "не занимают много памяти" и т. д. Впрочем, чем дольше вы пытаетесь сформулировать, тем более расплывчатым становится определение. Быстрые по сравнению с чем? Что, если система занимает много оперативной памяти, а конкурирующая система занимает такой памяти меньше, но для нее требуется больше места на жестком диске? Если система А возвращает ответ за три секунды, а система В за пять секунд, какая из них более производительная? Ответ может показаться очевидным, если только запросы, на которые отвечают системы, сами по себе не различаются. Производительность — это одна из тех концепций, которую сложно определить, потому что она будет означать разное для различных систем.

Представим видеоигру, где вы играете за тестировщика, сражающегося со злыми багами. Вы можете использовать клавиши-стрелки для движения и пробел для стрельбы из вашей инсектицидной винтовки. Каждый раз, когда вы нажимаете клавишу пробела, вы предполагаете, что ваш экранный персонаж начнет стрелять инсектицидами, скажем, в течение 200 мс — если это время окажется больше, вам покажется, что игра "лагает" после нажатия клавиш. Теперь представим вторую систему — прогнозирующий погоду суперкомпьютерный кластер, способный одновременно вести миллионы расчетов. Но из-за того, что прогноз погоды требует множества расчетов, требуется полчаса, чтобы после нажатия кнопки запуска получить прогноз на завтра. А до этого момента на экране будет отображаться "Ведется расчет...". У суперкомпьютера время отклика на несколько порядков больше, чем у видеоигры, но означает ли это, что суперкомпьютер менее производительный, чем игровая приставка? Можно поспорить о таком **показателе производительности**, как время отклика, но результаты времени отклика для погодного компьютера оказались значительно отличающимися.

Как определить, у какой системы лучшая производительность? Короткий ответ — никак! Системы выполняют различные — и в основном несовместимые — дейст-

вия. То, как вы измеряете производительность, будет зависеть от типа тестируемой системы. Впрочем, это не означает, что не существует правил или эвристики. В этой главе мы рассмотрим некоторые различные виды производительности и способы их тестирования. Однако это далеко не исчерпывающий список. У тестируемой вами системы могут быть уникальные аспекты производительности, которые не будут описаны ни в одной из книг конечной длины (которой, несомненно, является и эта книга).

19.1. Категории показателей производительности

Как было упомянуто выше, определение производительности для конкретной системы означает определение того, какие показатели производительности важны для пользователя или потребителя. Хотя существует огромное количество всевозможных показателей производительности, наиболее важными являются две основные категории: ориентированные на сервис и ориентированные на эффективность.

Показатели, ориентированные на сервис, измеряют, насколько хорошо система предоставляет услугу конкретному пользователю. Они измеряются с точки зрения пользователя и количественно определяют аспекты системы, которые важны для пользователя. Примерами ориентированных на сервис показателей могут быть среднее время загрузки страницы, время отклика или процент времени, в течение которого доступна система.

Существуют две основные подкатегории показателей, ориентированных на сервис. Первой является доступность — насколько доступна система для пользователя? Здесь может подразумеваться все, что угодно, от процентов времени работоспособности системы до того, насколько различные функции доступны пользователям в различное время. Второй основной подкатегорией является время отклика — как долго система отвечает на действия пользователя или возвращает результат?

Показатель, ориентированный на эффективность, измеряет, насколько эффективно система использует доступные вычислительные ресурсы. Вы можете смотреть на них как на показатели "с точки зрения программы" или по крайней мере с точки зрения компьютера. Примерами ориентированных на эффективность показателей могут быть загрузка процессора при выполнении определенного функционала, объем используемого дискового пространства или количество пользователей, которых одновременно может обслуживать система на конкретном сервере. Хотя последний пример может показаться ориентированным на сервис, то, что он связан с системным уровнем — сколько пользователей может обслуживать система, а не как это выглядит для определенного пользователя, — означает, что это показатель, ориентированный на эффективность. Каждый раз, когда вы измеряете с точки зрения системы, а не с позиций того, с чем *напрямую* имеет дело пользователь, вы тестируете ориентированный на эффективность показатель.

Как и в показателях, ориентированных на сервис, здесь имеются две основные подкатегории ориентированного на эффективность тестирования. **Пропускная спо-**

собность измеряет количество событий, которые может обработать система за определенный промежуток времени, например сколько пользователей могут авторизоваться одновременно или сколько пакетов может маршрутизировать система за 30 секунд. **Утилизация** измеряет процент или абсолютное значение компьютерных ресурсов, которые используются для выполнения заданной операции. Например, сколько дискового пространства требуется базе данных для хранения определенной таблицы? Сколько оперативной памяти потребуется для ее сортировки?

Показатели, ориентированные на сервис, подобны тестированию черного ящика в том, что они измеряют систему со стороны, без точного определения причины проблемы. Как прямолинейные инструменты они в целом эффективны в поиске того, где следует присмотреться к проблемам. Показатели, ориентированные на эффективность, подобны тестированию белого ящика, т. к. они требуют как понимания системы, так и общих технических знаний. Зачастую вы можете использовать ориентированные на сервис показатели для определения областей, где производительность может оказаться проблемой для пользователей, а затем использовать ориентированные на эффективность показатели для точного определения того, что вызвало проблему. Длительное время отклика с точки зрения пользователя может быть результатом задействования физической оперативной памяти и необходимости обмена большой части информации с диском. Ориентированное на сервис тестирование может обнаружить первую проблему, а ориентированное на эффективность тестирование сможет выяснить вторую.

19.2. Тестирование производительности: пределы и цели

Для определения, прошел ли успешно тест производительности или нет, вам необходимы **цели производительности**, или определенные числовые значения, которых предположительно должны достигать показатели производительности. Например, у вас может быть ориентированная на эффективность цель производительности, согласно которой инсталлятор тестируемой программы должен быть меньше 10 Мбайт (в наше время программа "Hello, world!" может занимать больше десяти мегабайт, но давайте не пока не будем жаловаться). У вас может быть ориентированный на сервис показатель, согласно которому система при нормальной нагрузке должна отвечать в течение 500 миллисекунд. Приводя цель к численному показателю, вы можете написать тест и определить, соответствует ли этому показателю система или нет.

Впрочем, цели являются идеалом. Зачастую, конкретный показатель производительности системы не может достигнуть своей цели. **Порог производительности** показывает точку, при которой показатель производительности достигает абсолютно минимальную приемлемую производительность. Например, хотя время отклика системы может составлять 500 миллисекунд, системные инженеры определили, что система приемлема для работы при времени отклика 3000 миллисекунд. Не очень

хорошо, если система будет соответствовать этому показателю, но при большем времени отклика, вероятно, систему допускать к работе нельзя.

Система, показатели производительности которой соответствуют только порогу, не должна рассматриваться, как "проходящая" эту конкретную метрику производительности. Здесь еще можно поработать! Вы увидите, что часто используемый в функциональном тестировании стандартный подход "прошел/не прошел" нельзя применять для метрик производительности. Существуют оттенки серого, и ими зачастую можно манипулировать. На каком "железе" запускаются тесты? На какой операционной системе? Какие еще процессы запущены? Какой браузер вы используете для доступа к сайту? Если изменить что-то из этих данных, та же самая система может показать совершенно другие результаты теста производительности.

19.3. Ключевые показатели производительности

Конкретные показатели производительности могут быть более или менее важными. Возвращаясь к нашему погодному суперкомпьютеру, можно сказать, что в этом примере было требование ко времени ответа; предполагалось, что результаты выдаются через полчаса. Хотя время отклика может быть очень важно — если приближается ураган, пользователям нужен результат ровно через 30 минут, чтобы жители могли приступить к эвакуации. В других случаях, таких как тестирование архивированных данных, время отклика может оказаться в два и более раза больше, и это не будет иметь весомого значения. Некоторые показатели производительности могут быть очень важными (в этом примере время отклика при работе с самым высоким приоритетом), а другие могут быть совсем не важны (время отклика при тестировании архивированных данных).

Хотя существует множество возможных показателей производительности системы, всегда будет подмножество, в котором вы заинтересованы более всего. Для видеоигры потребуется быстрое время отклика и высокая доступность, но не будет необходимости в минимизации использования дискового пространства, процессора и памяти. У работающего в фоне приложения будут совершенно другие требования — важно минимальное использование памяти и процессора, но время отклика совсем не важно. Лишь изредка пользователь или другой процесс взаимодействуют с ним, и когда это происходит, то осуществляется посредством сигналов, и почти мгновенность отклика не предполагается.

Выбирая наиболее важные показатели производительности, которые называются **ключевые показатели производительности** (key performance indicators, KPI), вы можете сфокусироваться на тех аспектах производительности, которые наиболее значимы для пользователей. Тогда это позволит разработчикам настраивать систему для удовлетворения потребностей пользователей, а не тратить время на те части системы, которые не особо нужны пользователям или уже достаточно производительны. В зависимости от специфики требований вполне возможно определить необходимые KPI на этапе разработки системы. Во многих случаях вам придется

определять их через пользовательское тестирование, изучая аналогичные приложения в данной области, или использовать для оценки свои знания по тестированию.

Выбор ключевых показателей производительности должен быть осуществлен перед началом тестирования; вы не должны тестировать множество показателей, а затем осмотреться и понять, какие из них важны. Вам нужно определить самые важные из них заранее, при разработке тест-плана для системы. Если вы следуете более упрощенному процессу разработки с менее значимой предварительной проработкой, вы должны, по крайней мере, определить, является ли конкретный показатель производительности ключевым перед запуском соответствующего теста. Это заставит вас помнить, какие аспекты производительности системы важны, а какие — нет, прежде, чем вы получите результаты. Если вы попытаетесь выбрать KPI позже, у вас может возникнуть соблазн (даже подсознательно) просто выбрать те показатели, которые соответствуют целям. Оставьте такое мышление рекламодателям.

19.4. Тестирование показателей, ориентированных на сервис: время отклика

Самый простой способ измерить время отклика, конечно же, просто следовать данному алгоритму:

1. Сделать что-то.
2. Нажать кнопку "Старт" на секундомере.
3. Ждать отклика.
4. Когда появляется отклик, нажать кнопку "Стоп" на секундомере.
5. Записать, сколько прошло времени.

Хотя это может быть простейший способ измерить время отклика, он далеко не лучший. У данного подхода переизбыток проблем. Во-первых, невозможно измерить доли секунды; время реакции человека слишком переменное и недостаточно быстрое. Вы не можете измерить нечто внутри системы, если вашим единственным интерфейсом системы является то, что вы видите. Такая работа занимает очень много времени, что затрудняет сбор больших наборов данных. Здесь возможна человеческая ошибка. (Вы когда-нибудь тратили целый день на измерения чего-либо? В какой-то момент вы обязательно забудете нажать кнопку "Старт" или случайно нажмете кнопку "Сброс" до того, как вы записали время.) Это идеальный способ испортить настроение тестировщику. (Вы когда-нибудь тратили целый день на измерения чего-либо? В какой-то момент вы начнете задумываться о поиске другой работы.) Из-за всех этих (и других) проблем, тестирование производительности часто осуществляется при помощи специальных программ.

Хотя тестирование в целом все больше и больше автоматизируется, конкретно тестирование производительности значительно зависит от автоматизированных тестов. Показатели производительности могут значительно отличаться от запуска к запуску из-за влияния различных переменных, которые вы можете практически

не контролировать. Примерами таких переменных среди прочих могут быть другие запущенные на сервере процессы, объем использованной оперативной памяти, работа сборщика мусора и время запуска виртуальной машины. Очень часто единственными способами получить по-настоящему правильный результат являются многократный запуск теста производительности и его последующий статистический анализ (например, получение среднего, медианного и максимального времени отклика). Единственным способом получения разумной выборки данных за разумное время является автоматизация процесса.

19.4.1. Что такое время?

Хотя это может звучать как философский вопрос, на самом деле у него имеются разные прочтения при тестировании времени отклика. Для компьютера существует несколько разных способов измерения времени. Для тестировщика производительности отчеты и измерение времени будут зависеть от тех факторов, которые его интересуют для определенного теста производительности.

Первый и, пожалуй, самый простой для понимания вид времени — **реальное время** (real time). Это то время, которое измеряется настенными часами, и поэтому его часто называют **временем настенных часов**. (А вы думали, что технические термины сложно понимать!) Оно подобно времени секундомера — как долго, с точки зрения пользователя, программа выполняет что-то? Таким образом, это обычно то время, которое волнует пользователей больше всего. Однако это еще не всё.

Реальное время принимает во внимание *всё*, что нужно системе для выполнения необходимого задания. Если это означает чтение данных с диска или из сети, то это учитывается в общем значении реального времени — независимо от того, насколько медленный диск или какова пропускная способность сети. Если работа идет в системе, в которой запущено множество других процессов, и это означает, что у задачи есть шанс получить только 5% процессорного времени, то это засчитывается реальным временем. Очень часто реальное время оказывается не лучшей метрикой для изучения, потому что оно учитывает много факторов, которые находятся вне контроля разработчика системы.

Пользовательское время (user time), наоборот, отдельно измеряет, сколько времени было потрачено на выполнение пользовательского кода. Оно не учитывает время, потраченное на ожидание ввода, чтение с диска, ожидание окончания работы с центральным процессором другого процесса и даже выполнение системных вызовов. Оно сфокусировано на той части системы, над которой разработчики имеют прямой контроль. В конце концов, вы можете легко изменить алгоритм сортировки в вашем коде, но если вам потребуется узнать частоту системного таймера, сделать это можно только через системные вызовы, над кодом которых у вас нет контроля (только если вы не захотите, например, создать свою версию Linux).

Время, потраченное на системные вызовы — когда код исполняется, но не вашей программы, а ядра, — называется **системным временем** (system time). Хотя у разработчиков нет прямого контроля над системным временем, т. к. они не пишут код ядра, опосредованный контроль у них имеется, поскольку вполне возможно

уменьшить количество выполняемых программой системных вызовов, изменить их порядок, вызывать другие функции или использовать прочие способы уменьшения системного времени.

Суммируя пользовательское время и системное время, вы получите **общее время** (total time) — количество времени, потраченного на выполнение кода в пользовательском пространстве или пространстве ядра. Это хороший показатель того, сколько времени тратится на выполнение вашего кода. Это позволяет избежать расчета времени, в течение которого другие процессы выполнялись процессором или система ожидала ввода, и дает возможность просто сосредоточиться на том, как долго выполнялся код. Тем не менее чрезмерное внимание к системному времени может отвлечь вас от проблем, связанных с соответствующими внешними факторами. Например, если вы тестируете работающую с базой данных программу, то бóльшая часть потраченного времени будет связана с чтением диска и записью на него. Было бы глупо сбрасывать со счетов это время — даже если у разработчиков нет прямого контроля над ним, его все равно следует учитывать.

В зависимости от того, что вы измеряете и какой контроль у вас над тестовой средой, отслеживание пользовательского, общего или реального времени может быть оптимальным показателем. Системное время используется редко, если только вы не тестируете новое ядро, операционную систему или сильно беспокоитесь о них по непонятным причинам. Помните, что хотя разработчики могут большей частью контролировать пользовательское время, опосредованно общее время и в большинстве случаев минимально контролируют реальное время, пользователей обычно волнует только реальное время! Если пользователь системы посчитает ее медленной, его не заинтересуют объяснения, что вы не можете контролировать скорость чтения с диска. С точки зрения тестирования, вам следует сфокусироваться на измерении реального времени, избегая при возможности посторонних факторов, таких как одновременный запуск других программ. Тем не менее для определенных процессов, которые связаны с центральным процессором или обычно работают в фоновом режиме, более подходящим будет фокусирование на общем или пользовательском времени.

В большинстве UNIX-подобных систем (таких как Linux или OS X) вы можете очень легко получить эти значения для оценки программы. Просто запустите команду `time <command>`, и после завершения программы будет отображено реальное, пользовательское и системное время (общее время легко вычисляется путем сложения пользовательского и системного времени):

```
$ time wc -w * | tail -n 1
   66156 total
real    0m0.028s
user    0m0.009s
sys     0m0.014s

$
```

Для Windows есть подобные разнообразные программы, например timeit.exe, которая поставлялась с Windows Server 2003 Resource Kit, но по умолчанию в состав операционной системы такие утилиты не входят.

19.4.2. Какие события следует измерять?

Это будет зависеть от того, какой показатель производительности вы используете. Если показатель у вас определен либо требованиями, либо договоренностями с командой, тогда измерение является прямолинейным процессом. Однако если вам нужно определить его самостоятельно, тогда первым шагом станет выяснение того, какие события важны пользователю системы, где медленное время отклика может вызвать трудности или раздражение.

Если вы тестируете запущенную на сервере систему, приведем некоторые события, которые можно проверить на время отклика:

1. Время загрузки страницы (для веб-серверов).
2. Время скачивания файла.
3. Время установки соединения.
4. Время до появления данных на экране клиента.
5. Время между установкой соединения и готовностью к вводу данных пользователем.

Для обычных программ на время отклика можно проверить следующие события:

1. Общее время выполнения.
2. Время с момента запуска до готовности к вводу данных.
3. Время между вводом данных и ответом.
4. Время между вводом и подтверждением ввода (т. е. индикатор "загрузка").

Кроме того, у вас могут быть более конкретные времена отклика, которые нужно проверить для используемой вами программы. Например, представим программу, которая позволяет студентам записываться на занятия. Здесь может быть показатель производительности, связанный со временем отображения всех занятий кафедры, и отдельный показатель производительности для времени регистрации на выбранное занятие. Не все времена откликов равны, и одни из них могут быть ключевыми показателями производительности, а другие — нет. Например, т. к. студенты, по всей видимости, будут чаще просматривать информацию о занятиях, чем записываться на них, отображение занятий может являться KPI, в то время как время записи на них измеряется, но не имеет какого-то отдельного предела или цели.

Тот факт, что у конкретного индикатора нет цели или предела, не означает, что его не надо измерять. Очень часто, особенно если показатель производительности измеряется автоматически, имеет смысл просто собирать времена отклика для различных событий и сохранять их. Затем их можно проанализировать в исследовательских целях, чтобы понять, имеются ли какие-нибудь странности или проблемы. Например, даже если время записи не было отмечено как показатель производительности, но мы замечаем, что обычно у студента уходит 5 минут, чтобы записаться на занятие, имеет смысл заняться данной проблемой для снижения этого времени. Не все требования к производительности определяются явно перед нача-

лом разработки, поэтому часто тестировщику приходится обращать внимание команды на проблемы, даже если перед ним не стояла задача заниматься подобными вопросами.

Определение допустимых границ времени отклика может быть трудным. Тем не менее существуют некоторые приблизительные рекомендации. Приведенные ниже взяты из книги "Инжиниринг юзабилити" Джейкоба Нильсена (Jakob Neilsen "Usability Engineering").

- <100 миллимекунд — время отклика, необходимое для ощущения, что система работает мгновенно;
- <1 секунда — время отклика, необходимое для того, чтобы ход мыслей не прерывался;
- <10 секунд — время отклика, необходимое, чтобы пользователь оставался сконцентрированным на приложении (и не отвлекался, чтобы посмотреть, что происходит в Интернете).

Хотя это приблизительные рекомендации для целей, они сами по себе не являются законами, и хорошее время отклика будет зависеть от целого ряда факторов (какую сеть использует система, какие вычисления выполняет система и т. д.). Существуют основанные на исследованиях Google эмпирические данные, что люди, даже неосознанно, будут выбирать сайты, которые загружаются быстрее на 250 миллисекунд и даже меньше. Любое время загрузки веб-страницы выше 400 миллисекунд вызовет снижение числа посетителей, и этот порог постоянно уменьшается. Время идет, и пользователи становятся все более нетерпеливыми.

И напоследок, вполне возможно, что время отклика может быть слишком быстрым! Подумайте о скролл-боксе, который двигается так быстро, что пользователь не может контролировать его, или об экране, который изменяется так быстро, что это дезориентирует читателя. В некоторых случаях для показателя производительности могут устанавливаться верхние и нижние границы.

19.5. Тестирование показателей, ориентированных на сервис: доступность

Доступность, часто называемая временем безотказной работы, показывает процент времени, при котором пользователь может получить доступ к системе при его ожиданиях осуществить это. Таким образом, в то время, когда система оказывается неработоспособной, снижается доступность — из-за техобслуживания, из-за вызвавшего сбой необработанного исключения или из-за поломки жесткого диска.

Многие поставщики облачных услуг предоставляют **соглашение об уровне обслуживания** (service level agreement, SLA), в котором устанавливается уровень доступности предоставляемых услуг. Часто он определяется в формате "*n* девяток", показывающем, "как много девяток" надежности обеспечивается. Эти "девятки" относятся к 99% (две девятки), 99,9% (три девятки), 99,99% (четыре девятки) и т. д. Во время написания этой книги, например, Amazon S3 обещал предоставлять дос-

тупность трех девяток (доступность 99,9% времени, что означает время простоя менее 45 минут в месяц). Для некоторых систем, таких как автономные машины и космические аппараты, требования ко времени безотказной работы могут быть значительно выше.

Как мы определяем подходящий уровень доступности для тестируемой системы? Если это ключевой показатель производительности, соответствующее значение, вероятно, будет определено как требование. Однако если нет, потребуется обсуждение с системными инженерами для установления подходящего уровня доступности вашей системы. Обратите внимание, что увеличение уровня доступности на один "уровень" девяток оказывается все более и более сложным и потребует значительного объема инженерной работы и проектирования. Например, переход от одной девятки (90% доступности, или менее примерно 40 дней простоя в году) к двум девяткам доступности (99%, или менее примерно 4 дней простоя в году) может потребовать всего лишь добавления сохраняющей резервные копии программы и процесса-демона, перезапускающего процессы в случае возникновения проблем с ними. Следующий шаг к доступности 99,9% (время простоя менее 9 часов в году) может потребовать установки дополнительных серверов для сохранения резервных копий и пересмотра дизайна системы с целью сделать ее более распределенной. Если идти дальше, вам может потребоваться выполнение формальной верификации вашего программного обеспечения, покупка поддерживающих горячую замену жестких дисков и т. д. Например, вы пытаетесь получить доступность в шесть девяток, что соответствует примерно 30 с простоя в году. Все, что может пойти не так, — перебои в подаче электроэнергии, отключение от сети, падение метеоритов и т. д. — должно устраняться в автоматическом режиме. При таких уровнях доступности ни у одного человека не будет достаточно быстрой реакции для предотвращения простоя, который разрушит вашу цель производительности.

Все это на самом деле не отвечает на вопрос, потому что не существует одного ответа. Уровень доступности будет зависеть от области применения тестируемого вами программного обеспечения, от конкретного бизнеса и от различных потребностей пользователя. Даже у больших веб-сервисов бывают значительные времена простоя, во время которых вы не сможете проверять почту или смотреть бессмысленные истории, которые ваши друзья публикуют в социальных сетях.

Некоторые системы, такие как ПО для авионики или управления космическими аппаратами, непрерывно доступны годами или даже десятилетиями без простоев. Другие, например исследовательские программы, могут находиться в нерабочем состоянии чаще, чем в рабочем. Впрочем, как тестировщик, вы можете задать несколько вопросов, чтобы определить приемлемый для вас уровень доступности.

1. *В чем негативный эффект, если система недоступна?* Для исследовательских программ это может привести к задержке эксперимента. В случае веб-сайта это может означать потерю денег для бизнеса. Для авиационной программы это может означать падение самолета.
2. *Сколько усилий команда готова и может выделить для доступности?* Если у команды нет внутреннего стимула работать над повышением доступности

и нет денег или ресурсов для работы над этим, то, возможно, не стоит тратить много времени на ее тестирование.

3. *Как ориентация на доступность вступает в противоречие с другими аспектами программы?* Каждая минута, потраченная вами на тестирование доступности, означает меньше времени на тестирование других аспектов программы. В некоторых системах ваши тесты доступности помогут найти дополнительные дефекты. В других системах оно может сказаться негативно на прочих видах тестирования.

Задавая такие вопросы, вы в тандеме с разработчиками и другими заинтересованными лицами сможете определить, какой уровень доступности будет подходящим для вас. Помните, что цель и пороговые уровни должны быть установлены! Конечно, большинство пользователей скажут, что они хотели ли бы, чтобы у системы была доступность 100%, и нет никаких проблем, чтобы поставить себе такую цель. Однако, проводя более глубокие исследования — например, готовы ли вы заплатить в три раза больше за систему, которая доступна 99,999% времени, а не 99,9% времени, — вы часто сможете определить достаточное пороговое значение для этого показателя производительности.

Когда мы определили цель и пороговые уровни для доступности, как мы протестируем их? Простым ответом может стать запуск системы на год, определение периода, когда она работала, и деление этого времени на общее время, чтобы получить процент безотказной работы в году. Бум, у нас есть результаты! Это, конечно же, предполагает, что бизнес без проблем подождет год ради получения результатов и никакая дальнейшая разработка в это время вестись не будет. В конце концов, если использовать другой код, то это разрушит весь эксперимент! Существует не так много систем, для которых подошел бы такой способ. Такое простое решение, подобно использованию секундомера для тестирования времени отклика, практически невозможно использовать на практике.

Вместо этого можно смоделировать тестируемую систему. В простой модели можно запустить систему на сутки, подсчитать, сколько сбоев произойдет, и оценить, сколько времени уйдет на их исправление. Разделите это время на время в сутках, и теоретически вы получите доступность системы в течение любого промежутка времени. Если система сталкивается с двумя проблемами с интервалом 11 часов и 50 минут, каждая из которых привела к недоступности системы в течение 10 минут, тогда вы можете просуммировать минуты (20 минут), разделить их на 24 часа (1440 минут) и получить время непрерывной работы для этого дня (20/1440), составляющее 98,6% доступности — не совсем две девятки.

На самом деле, это обобщение стандартной модели доступности. Как только вы рассчитаете **среднее время между отказами** (mean time between failures, MTBF) и **среднее время восстановления** (mean time to repair, MTTR) для этих отказов, вы можете просто использовать следующее уравнение для приблизительной оценки доступности системы:

```
(MTBF / (MTBF + MTTR)) * 100%
```

Для нашего примера, приведенного ранее, среднее время между отказами составило 710 минут, а среднее время восстановления — 10 минут. Таким образом, MTBF + MTTR равняется 720 минутам.

```
(710 / (710 + 10)) * 100% = 98.61% доступности
```

Эта простая модель не учитывает то, что компоненты, как правило, не выходят из строя с одинаковой скоростью во времени. Хотя вы сможете увидеть некоторые случайные сбои, с которыми неизбежно столкнется система, и, вероятно, значительную часть проблем, связанных с первым выполнением кода в реальной среде, эта частота ошибок, скорее всего, не будет отражать долгосрочный уровень ошибок в системе. Наряду со "сбоями запуска" и "случайными сбоями" вы также должны учитывать, что оборудование изнашивается. Вторым фактором является то, что нечто, способное произойти, но не произошедшее во время тестирования, безусловно произойдет, если система будет работать в течение какого-либо разумного промежутка времени — например, перебои в подаче электроэнергии или отказы системы охлаждения. Просто потому что вы не видите этих проблем в данное время, не означает, что они никогда не произойдут, подобно тому однократное подбрасывание игральной кости и выпадение четверки не означает, что вы никогда не получите шестерку. В конце концов, может случиться то, что Дональд Рамсфелд удачно назвал "неизвестными неизвестностями" — ситуации, которые вы и создатели системы никогда не рассматривали и даже не предполагали возможность их возникновения. Согласно астроному Филу Плейту (Phil Plait), шансы гибели от падения метеорита составляют примерно 1 к 700 000 — немного меньше шансов умереть в результате фейерверка и чуть больше шансов умереть в результате террористической атаки. Вы можете предположить, что дата-центр, в котором размещен ваш сайт, будет существовать бесконечно долго и не будет поражен метеоритом? Чем дольше работает система, тем выше вероятность, что эти маловероятные события произойдут, и их будет трудно отследить с помощью наивной модели простого умножения результатов за один день на количество дней в году.

Системы имеют тенденцию следовать так называемой **кривой ванны** (bathtub curve). Это означает, что многие сбои имеют тенденцию происходить в начале, когда система просто стартует, затем количество сбоев уменьшается, а затем в итоге снова начинает увеличиваться, т. к. компоненты изнашиваются, библиотеки перестают обновляться, прекращается поддержка внешних API и т. д. При построении графика кривая выглядит как вид ванны сбоку. Сначала большое количество простоев и ошибок, затем резкое снижение времени простоя и количества ошибок, т. к. очевидные проблемы проявляются и устраняются, затем длительный период относительно небольшого времени простоя, а потом наклон вверх по мере того, как система становится старше и все начинает идти не так. Включение кривой ванны в планирование вашей модели может быть полезным, хотя это лишь приблизительное руководство. Вам следует предполагать, что у системы будет больше простоев в начале, когда ваша команда обнаруживает и исправляет дефекты, ставшие заметными только после того, как система запущена "по-настоящему".

Для вашей конкретной системы вы можете определить, насколько сложной нужно сделать модель, сколько времени вы хотите потратить на сбор данных и сколько

внешних факторов, таких как отключение внешних API или прочих проблем, следует принимать во внимание. Помните, что при тестировании производительности в целом редко рекомендуется использовать одну точку данных для обобщения. Если вы собираетесь использовать сутки в качестве основы для своих расчетов, не используйте тестирование всего одного дня, чтобы определить среднее время до отказа и среднее время восстановления. Не забудьте выделить пространство для вещей, о которых вы даже не думали и не предполагали, что что-то пойдет не так, — помните закон Мерфи. Учитывая вышесказанное, базовая концепция (MTBF/(MTBF + MTTR)) должна обеспечивать хороший старт определения уровня доступности любой заданной системы.

Теперь, когда вы знаете, что вам необходимо и как вы будете работать с результатами, пришло время сгенерировать эти числа. Осуществить это можно при помощи **нагрузочного тестирования**. В нагрузочном тестировании система настраивается и проводится ее контроль при прохождении событий. Вид событий зависит от тестируемой системы, например для веб-сервера это может быть загрузка страницы, а для роутера таким событием станет маршрутизация пакетов. Используя различные виды нагрузочных тестов на основе реалистичных предположений, можно определить MTBF и MTTR и, следовательно, ожидаемую пользователями доступность.

Базовый тест запускает систему, выполняющую несколько процессов или не выполняющую вообще, просто чтобы удостовериться, что система фактически может работать в течение заданного периода времени. Зачастую это нереалистичный сценарий — система, которая никогда ничего не делает, вряд ли будет разработана в первую очередь, — но он обеспечивает "базовый уровень", с которым будущие тесты могут сравнивать результаты. Противоположностью базового теста является **стресс-тест**, в котором система подвергается "стрессу" и должна обрабатывать большое количество событий за малый период времени. Обычно стресс-тесты не запускаются на длительное время, но они полезны для моделирования тех периодов, когда система находится в состоянии сильного стресса. Примерами таких случаев могут быть DDoS-атаки на веб-сайты или начало осеннего семестра для системы регистрации на занятия.

Тест стабильности (stability test или soak test) часто является единственным и самым реалистичным из запускаемых нагрузочных тестов. В такого рода нагрузочных испытаниях в течение длительного времени обрабатывается постоянное (небольшое или среднее) количество событий, чтобы определить, действительно ли система стабильна при выполнении реалистичной работы.

Если это возможно, определите, что такое реалистичное использование, чтобы вы могли смоделировать свою систему соответствующим образом. Нередко комбинируют различные нагрузочные тесты, чтобы прийти к единым значениям MTBF и MTTR. Например, вы можете определить, что система при запуске тестов 10% времени находится в "стрессовом состоянии", а 90% времени в "состоянии стабильности". Если MTBF составляет 10 минут при стрессовом состоянии и 12 часов при состоянии стабильности с MTTR, равным 5 минутам в каждом случае, вы можете выполнить более детальный расчет:

```
MTBF(стресс) = 10
MTBF(стабильность) = 720
MTTR = 5
MTBF = (0.1 * MTBF(стресс)) + (0.9 * MTBF(стабильность)) = 649
(MTBF / (MTBF + MTTR)) * 100
(649 / (649 + 5)) * 100 = 99.23% доступности
```

Это всего лишь пример, и он, безусловно, может быть улучшен (а любое состояние системы, которая достигает таких высоких уровней недоступности во время стресс-тестирования, вероятно, следует рассмотреть более внимательно). Некоторые системы имеют пики использования, а другие — нет. Некоторые системы обрабатывают множество маленьких событий, другие будут обрабатывать меньшее число более сложных событий. Если вы сможете собрать реальные данные и показатели использования вашей системы, вы сможете лучше смоделировать ее поведение и, таким образом, получить лучшие показатели доступности. Чем реалистичнее данные, тем реалистичнее модель, которую вы можете создать. Однако даже с самыми реалистичными данными будьте готовы ошибиться — в сложной системе много того, что может пойти не так, и это невозможно принять во внимание в любой модели.

19.6. Тестирование показателей, ориентированных на эффективность: пропускная способность

Как вы помните, ориентированные на эффективность показатели рассматривают систему с точки зрения системы и с позиций, насколько эффективно она использует доступные ей вычислительные ресурсы. Одним из критериев эффективности является **пропускная способность**, или количество событий, которые могут быть обработаны за заданное количество времени при определенных настройках оборудования. В качестве примеров можно привести количество страниц, которые веб-сервер может отдать за минуту, или сколько SQL-запросов сервер баз данных может обработать за минуту. Это может показаться похожим на время отклика, или, по крайней мере, быть его обратным, но не все так просто. Если время отклика измеряло время отклика, с точки зрения конкретного пользователя системы, то пропускная способность — это показатель того, насколько эффективно система в целом взаимодействует со множеством пользователей. Если я просто использую веб-сервер, меня не беспокоит, как быстро другие пользователи получают свою информацию. Если же я администратор веб-сайта, работающего на конкретном сервере, то, без сомнения, для меня это очень важно!

Подобно тому, как мы действовали, тестируя доступность, мы можем использовать нагрузочное тестирование для определения пропускной способности системы. Однако вместо того, чтобы определять, работает ли система, мы можем отдельно проверить, какая частота событий является устойчивой до того, как будет достигнут некоторый заранее определенный нижний порог производительности. Для нашего

примера с веб-сервером — сколько страниц в минуту может отдавать наш веб-сервер до того, как среднее время отклика опустится ниже 3 секунд? Для системы регистрации на занятия — сколько студентов могут одновременно просматривать курсы, прежде чем заполнится очередь запросов к базе данных? Точные значения того, что считается "ниже порога производительности", зависят от системы, но обычно это среднее или максимальное время отклика. Мы используем ориентированные на сервис показатели как часть тестирования, ориентированного на эффективность показателя! Это говорит о том, что различные аспекты производительности часто взаимосвязаны; низкая пропускная способность может быть причиной медленного времени отклика, а плохое использование ресурсов может напрямую привести к отсутствию доступности.

Определение значений для самого низкого порога будет сильно зависеть от KPI для этой конкретной системы. Еще раз, они должны быть определены до начала тестирования, чтобы не подгонять результаты. После их определения вы должны увеличивать количество событий до тех пор, пока система не сможет больше работать в пределах определенного порога производительности.

Не забудьте отслеживать уровень событий, эквивалентную пропускную способность и, если возможно, соответствующие показатели утилизации (см. следующий раздел для получения дополнительной информации об измерении утилизации). Это позволит вам увидеть, существуют ли какие-либо шаблоны в уровнях пропускной способности, что позволит вам экстраполировать дальше, чем собранные вами данные. Например, предположим, что вы наблюдаете время отклика, постоянно составляющее 500 миллисекунд до 100 событий в секунду, но при 150 событиях в секунду время отклика замедляется до 800 миллисекунд. При 200 событиях в секунду время отклика увеличивается до 2500 миллисекунд. Мы можем наблюдать некоторый сверхлинейный рост, начинающийся примерно со 100 событий в секунду, а разработчики получают дополнительную информацию для поиска узких мест. Это та точка, где данные начинают выгружаться на диск? У нас имеются 100 потоков, и эта точка, в которой пул потоков иссякает? Хотя у вас может не быть ответов заранее, представление о росте, а особенно о той области, где пропускная способность начинает снижаться, значительно облегчит повышение производительности. Как мы уже говорили в разделе о регистрации дефектов, быть более точным почти всегда лучше, чем менее точным!

Уровни пропускной способности очень чувствительны к тому, на каком оборудовании вы работаете с ПО, поэтому даже более важно, чем в других тестах, убедиться, что вы работаете на одном и том же оборудовании от запуска к запуску, чтобы обеспечить достоверные результаты. Вы также можете попробовать определить пропускную способность, используя одно и то же ПО на разных аппаратных конфигурациях, что также способно помочь отследить причину замедления работы. Например, система, у которой значительно понизилась пропускная способность после запуска на машине с незначительно меньшим объемом ОЗУ, может иметь проблему с памятью.

19.7. Тестирование показателей, ориентированных на эффективность: утилизация

Тестирование **утилизации** означает определение количества компьютерных ресурсов (на абсолютной или относительной основе), которые конкретная система использует при выполнении некоторых функций. "Вычислительные ресурсы" — очень широкий термин, охватывающий всё, что программа может "использовать" на машине. В качестве примеров можно привести:

- центральный процессор;
- использование диска;
- ОЗУ;
- использование сети.

Хотя это некоторые из наиболее часто измеряемых ресурсов, существуют более специфичные, которые также можно измерить. Количество прочитанных с диска байтов? Резервный кэш байтов нормального приоритета? Количество транзакций C3 на вашем процессоре за последнюю секунду? Все это (наряду с буквально тысячами другого) можно протестировать прямо из коробки на вашем компьютере с Windows с помощью инструмента perfmon. Для других операционных систем существуют подобные инструменты, такие как Activity Monitor для OS X и sar для Linux. Однако эти очень специфические измерения обычно необходимы только после обнаружения проблемы в результате проведения обобщенных измерений, перечисленных выше.

Зачастую, вам всего лишь нужна оценка использования ресурсов на уровне "палец на ветру" или понимание того, какой процесс потребляет ресурсов больше всего. Для этого можно воспользоваться инструментами, которые имеются в различных операционных системах: в Windows есть Task Manager, а в OS X и Linux это top. Каждая из этих программ позволит вам увидеть, какие из запущенных приложений потребляют больше всего ресурсов процессора или памяти. Я опущу информацию, связанную с их запуском; вы можете воспользоваться man top для получения руководства UNIX для top или найти соответствующую документацию Microsoft для Task Manager.

Оба этих инструмента похожи в том, что показывают данные об использовании ресурсов в определенный момент времени. Они позволят вам определить, был ли скачок использования ресурсов процессора при выполнении какого-либо действия, или как использовалась память с течением времени. Например, для программы на Java вы часто будете наблюдать результат работы сборщика мусора — уменьшающееся количество памяти будет возвращаться к нормальному состоянию в форме графика, напоминающего пилу.

В то время как ориентированные на сервис показатели могут предупредить, что с системой что-то не так, использование такого инструмента, как top, позволит вам немного углубиться и определить, скажем, что большое время отклика (показатель, ориентированный на сервис), похоже, является результатом того, что каждый

запрос загружает центральный процессор на 99%. Однако исследовать утилизацию ресурсов таким образом очень неэффективно. Современные процессоры сложные, современные программы сложные, современная память сложная, и все вместе это означает, что пытаться понять использование ресурсов системы сложно. Инструкции могут быть конвейерными, а потоки могут запускаться на разных ядрах, существуют разные уровни кэша и различные виды использования памяти, программное обеспечение может работать по-разному и по-разному использовать ресурсы в различных средах. Но что более важно — после того, как вы определили, что проблема в повышенной загрузке центрального пропроцессора или большом объеме используемой памяти, как это описать как дефект, которым может заняться разработчик?

Ответ заключается в том, что вы получаете более конкретную информацию через **профилировщика** (также известного как **инструмент профилирования**). Профилировщик, такой как JProfiler или VisualVM для Java-программ, позволит вам не только увидеть, что процесс сильно загружает процессор, но и сколько ресурсов CPU потребляет каждый метод. Разработчику гораздо проще отследить проблему, если он знает, в каком методе она наиболее вероятна! Вместо получения информации, что система занимает 100 Мбайт памяти, вы можете увидеть, какие объекты были созданы и экземплярами каких классов они являются. Но хотя это и поможет вам сузить проблему, попытка изначально начинать работу с профилировщиком может оказаться избыточной. Очень легко оказаться перегруженным данными, если у вас нет конкретной цели.

Если вы решите, что проблема связана с использованием сети, вы можете прибегнуть к помощи **анализатора пакетов**, например Wireshark, для проверки отдельных пакетов или проведения статистического анализа. Это позволит вам точно определить, какие отправляемые или принимаемые пакеты могут оказаться "узким местом", либо выяснить, отправляются ли лишние данные, которые могут быть причиной снижения производительности. Например, хотя временные метки могут быть необходимы для регулярной отправки, действительно ли необходимо включение часового пояса исходной системы? Могут ли временные метки всегда быть в UTC-формате и корректироваться локально в соответствии с настройками каждой принимающей системы?

19.8. Общие советы и рекомендации для нагрузочного тестирования

1. Используйте инструменты для любых нестандартных измерений производительности. Нельзя полагаться на людей при проведении множества тестов производительности из-за допускаемых ими ошибок. Все это также может привести к снижению духа тестировщиков.
2. Сравнивайте яблоки с яблоками; поддерживайте переменные одинаковыми между запусками одного теста! Не запускайте первую версию вашей программы на ноутбуке, а вторую — на суперкомпьютере, чтобы потом сообщить о значительном прорыве в скорости.

3. Помните о затратах на запуск и/или отключение. Если системе необходимо запустить виртуальную машину (например, JVM для всех процессов Java), вам может потребоваться способ стандартизовать это для всех запусков, игнорировать первый результат, поскольку он будет включать время запуска, или учитывать его иным образом.
4. Будьте внимательны к кэшированию. Если система кэширует результаты, вы можете увидеть очень разные измерения производительности для первого запуска и последующих второго и третьего.
5. Сохраняйте контроль над системой тестирования. Вам может потребоваться убедиться, что другие пользователи не вошли в систему, что настройки одинаковы между запусками, что дополнительное программное обеспечение не установлено и т. д. Помните, что тестирование производительности похоже на проведение научного эксперимента — если вы не можете снизить влияние посторонних переменных, вы не можете доверять ему.
6. Подготовьте хорошие входные данные. Если возможно, попробуйте тестировать с реальными или по крайней мере близкими к реальным данными. Зачастую наблюдаемая производительность будет зависеть от данных. Если вы использовали пузырьковую сортировку, но ваши тестовые данные располагались по порядку, вы можете не заметить плохую производительность, которая обычно проявляется с неупорядоченными данными, т. к. пузырьковая сортировка — это $O(n)$ для отсортированных данных, но $O(n^2)$ в среднем.
7. Не доверяйте одиночному запуску. Даже если вы попытаетесь удалить все посторонние переменные из своего теста производительности, всегда будут присутствовать элементы, которые вы не можете контролировать, начиная от того, сколько памяти выделено, и заканчивая порядком выполнения потоков. Хотя эти факторы могут или не могут оказать существенное влияние на производительность системы, присутствовать они будут всегда. Единственный способ минимизировать их влияние — запустить тест несколько раз и предположительно сгладить данные, взяв средний результат или иным образом проанализировав его статистически.
8. Отслеживайте свои тестовые прогоны и сохраняйте данные. Это позволит вам определить, когда проблемы начали проявляться — проблемы, которые вы, возможно, не заметили изначально. Это также позволит вам определить тенденции. Например, вы можете заметить, что с каждой последующей версией тестируемого программного обеспечения использование памяти увеличивается, а нагрузка процессора снижается.
9. Наконец, подумайте, сколько тестов производительности необходимо и что вам следует делать. Если ваша система работает лишь иногда, так ли важно время отклика? Если система потребляет слишком много ресурсов процессора, следует ли сделать эту проблему приоритетной для исправления? Как и в других видах тестирования, время, потраченное на поиск одних проблем, неизбежно приводит к уменьшению времени на поиск проблем других типов. Будьте уверены, что вы используете свое время (а оно ограниченное) с умом.

Глава 20

Тестирование безопасности

Когда компьютеры только начинали становиться вещью (это технический термин), безопасность не была ключевой движущей силой при разработке программного обеспечения для них. Системы редко объединялись в сети, поэтому, чтобы повлиять на работу системы или ее данные, вам необходимо было физически добраться до компьютера. Еще в 1950-е годы люди знали, как обезопасить физические места (замки и охранники довольно полезны в реальном мире). В тех редких случаях, когда компьютеры были сетевыми или общедоступными, было мало пользователей, и они, как правило, были аспирантами, разработчиками или другими авторизованными пользователями. Тогда не предполагалось, что кто-то попытается взломать систему. Ричард Столлман (Richard Stallman), основатель Free Software Foundation, выступил против использования паролей в операционной системе ITS. Любой пользователь мог сломать работу системы с запущенной ITS, просто напечатав `KILL SYSTEM`. Другими словами, люди обычно доверяли друг другу. Если вам интересно прочесть об этом золотом веке компьютеров, рекомендую "Хакеры: герои компьютерной революции" Стивена Леви (Steven Levy "Hackers: Heroes of the Computer Revolution").

Название этой книги позволяет ненадолго отклониться в сторону. Хотя термин "хакер" (hacker) часто используется для обозначения человека, который "злонамеренно взламывает компьютерные системы", на самом деле у него довольно давно имеется другое значение. Это слово означает того, кто сделал что-то умное, новое или интересное. Несмотря на то что мир настаивает на термине "хакер", я буду использовать термин "взломщик" (cracker) для обозначения человека, взламывающего системы, доступ к которым для него не разрешен.

В любом случае человечество такое, какое оно есть, и идиллическое состояние, когда пользователи компьютеров в основном доверяли друг другу, не могло продолжаться вечно. Хотя в 1960-х годах было мало сетевых компьютеров, существовала огромная сетевая система, которая охватила огромный процент населения — телефонная система. Люди, известные как "телефонные фрики", исследовали телефонную систему, выясняя, как создавать позволяющие звонить бесплатно устройства или как одурачить таксофоны, чтобы они решили, будто внутрь брошена монетка. В то время как основной целью большинства телефонных фриков было понять лабиринтную сложность телефонной системы, сотрудники служб безопасности теле-

фонных компаний и полиция увидели в этом раннюю форму "цифрового нарушения".

По мере того как мир становился все более объединенным в сеть, истории о сбоях безопасности программного обеспечения стали обычным явлением. Можно вспомнить такой фильм, как "War Games", в котором управляющий ядерным арсеналом США искусственный интеллект был одурачен подростком-компьютерщиком, или книгу "Яйцо кукушки" ("The Cuckoo's Egg"), в которой правдиво рассказывалась история бывшего астронома, выслеживающего западногерманского компьютерного взломщика. Реальные события, такие как отключивший большую часть Интернета в 1988 году "Червь Морриса" (Morris Worm), также помогли обозначить безопасность как риск, которым должны заниматься программисты. Родилась концепция **информационной безопасности**. В программном обеспечении начали появляться функции, предотвращающие несанкционированный доступ или изменение данных.

Сегодня взлом компьютерной безопасности — это большой бизнес, и многие взломщики внедряются в системы, чтобы шантажировать компании, красть информацию о кредитных картах или блокировать работу веб-сайтов, которые их по каким-то причинам не устраивают. Аналогично защита систем от несанкционированного доступа и помощь в поиске слабых мест в системе до того, как их найдут "плохие парни", также является большим бизнесом.

20.1. Вызовы в тестировании безопасности

Тестирование безопасности отличается от других видов тестирования программного обеспечения тем, что существует умный противник — на самом деле много противников, хотя не все из них действительно "умные", — который также ищет дефекты. Вы можете представить пытающихся попасть в вашу систему взломщиков как тестировщиков, потому что они постоянно ищут слабые места в вашей программе, которые могут использовать для получения полного доступа или кражи данных. Когда вы проводите юнит-тестирование, вам не нужно беспокоиться, что система изменится из-за ваших действий или *пытаться* заставить ваши тесты провалиться, но это определенно относится к тестированию безопасности.

В некотором смысле вам намного сложнее, чем вашим противникам. Обычая метафора (некоторые скажут, что это клише) — защита за́мка. Если враг найдет одну незапертую и незащищенную дверь, он сможет пронестись по всему замку. Вы, с другой стороны, должны убедиться, что защитили каждый вход. Точно так же вам нужно проверить все слабые места в вашей программе, в то время как противнику достаточно найти только одно. Не имеет значения, что вы защитились от SQL-инъекций, если злоумышленник может легко найти пароли из-за уязвимости небезопасного хранения.

Хотя до сих пор существует *миф* о компьютерной безопасности, что те, кто пытается взломать компьютер, являются просто любопытными детьми (хотя иногда это так и есть!), многие атаки выполняются за финансовое вознаграждение. Люди продают время ботнета по высокой цене для рассылки спама. Они разрабатывают и продают эксплойты "агрессивным хакерским" группам или неназванным прави-

тельственным агентствам. Они залезают в базу данных кредитных карт интернет-магазинов не для того, чтобы увидеть, можно ли это сделать, а потому, что хотят покупать вещи с помощью кредитных карт (или продавать их на черном рынке и получать деньги таким образом). Существует огромный рынок, связанный с безопасностью информации и "секретных" данных.

Тестирование безопасности требует от вас мыслить как противник. Для эффективного тестирования системы вам нужно перестать думать как честный человек, которым, я уверен, вы являетесь. Вместо этого вам нужно думать как некто, готовый на хитрости для проникновения в систему, и подойти к ней так, как не стали бы подходить обычные пользователи. В конце концов, если имелись уязвимости, которые могут быть обнаружены при обычном использовании программы, они, вероятно, уже были обнаружены и исправлены. В наиболее опасных дефектах система используется так, как обычные люди никогда не додумаются. Опять же, это сильно отличается от стандартного функционального тестирования, где наиболее опасные дефекты — это те, с которыми рядовые пользователи сталкиваются чаще всего! Например, если ваш текстовый редактор дает сбой всякий раз, когда вы нажимаете клавишу <E>, то это серьезный дефект с огромным влиянием, который делает текстовый редактор бесполезным. Конечно, предполагается, что вы не Жорж Перек, пишущий роман "La Disparition" ("Исчезновение") — неясный угловой случай. Если сбой происходит только тогда, когда пользователь вводит "%&#@@!_!<Ctrl-R><Ctrl-Q>", то этот дефект можно не исправлять. Единственные люди, которые попробуют проверить это, — те, кто читают данную книгу и задаются вопросом: действительно ли этот дефект присутствует в каком-то текстовом редакторе (я никогда не скажу)? Однако если те же самые дефекты позволяют получить доступ к банковскому счету, то обратные утверждения могут оказаться истинными. То, что <E> вызывает сбой, будет легко обнаружено; возможно, только целенаправленный злоумышленник сообразит набрать "%&#@@!_!<Ctrl-R><Ctrl-Q>", что позволит ему месяцами летать под радарами незамеченным, а все больше и больше счетов будут загадочно истощаться.

Сбои в работе системы безопасности могут быть катастрофическими и иметь более серьезные последствия, чем аналогичные функциональные сбои. При условии хорошей стратегии резервного копирования и других простых процедур безопасности большинство функциональных или нефункциональных дефектов не вызовут значительных разрушений. Конечно, не все дефекты тривиальны; существует множество примеров дефектов программного обеспечения, вызывающих проблемы вплоть до гибели людей (об известном случае можно прочитать в "Расследовании аварий Therac-25" Н. Левенсона и С. С. Тернера, в котором подробно описывается, как ошибка "состояние гонки" вызвала передозировку радиацией нескольких человек). Тем не менее подавляющее большинство дефектов не оказывают настолько серьезного воздействия. Однако влияние дефектов безопасности может быть приумножено злоумышленником. Если из-за "состояния гонки" тормоза́ автомобиля временно перестают работать, это плохо. Однако если этой уязвимостью может воспользоваться злоумышленник по своему желанию, она становится значительно опаснее — в конце концов, если ваши тормоза не сработают при парковке или при очень мед-

ленном движении, единственным ущербом окажется выброс адреналина и потертое крыло автомобиля. Если же тормоза отключаются преднамеренно и с плохим умыслом, злоумышленник может дождаться, когда вы начнете спускаться вниз по крутому склону, где урон может быть гораздо серьезнее. Если ваша программа позволяет случайно зачислить на банковский счет дополнительные тысячи долларов при определенных, но недетерминированных обстоятельствах, это может вообще не быть большой проблемой, т. к. в конечном итоге банк обнаружит дополнительные деньги на счете и удалит их. Если же кто-то может преднамеренно осуществить подобное зачисление, он как можно скорее направится к банкомату для снятия наличных, что значительно усложнит решение вопроса.

Наконец, даже если вы нашли все уязвимости в коде вашей системы и довели ее до полной пуленепробиваемости, весь ваш тяжелый труд может оказаться бесполезным, если кто-то позвонит одному из ваших пользователей, представится полицией паролей округа Аллегейни и запросит подтверждение пароля. Это случается чаще, чем вы думаете, — прочитайте "Призрак в Сети" Кевина Митника (Kevin Mitnick "Ghost in the Wires"), чтобы узнать несколько удивительных историй об успешном использовании подобных методов. Люди часто путаются в технологиях и информационной безопасности или просто не обращают достаточно внимания на то, о чем их кто-то спрашивает. Как говорит Джорджиа Вейдман, автор книги "Тестирование на проникновение" (Georgia Weidman "Penetration Testing"): "Пользователи — это уязвимость, которую никогда нельзя исправить".

20.2. Основные концепции компьютерной безопасности

Хотя мы уже использовали некоторые из них, пришло время дать кое-какие формальные определения терминов, связанных с безопасностью, чтобы мы могли их обсудить более подробно.

Ключевым элементом безопасности является триада **InfoSec**, также называемая триадой **CIA**. Она состоит из следующих трех атрибутов системы: **конфиденциальность** (Confidentiality), **целостность** (Integrity) и **доступность** (Availability). Конфиденциальность означает, что посторонние пользователи не могут прочитать какие-либо данные. Это могут быть либо данные всей системы (например, пользователь с правами root в системе UNIX имеет доступ ко всему, что хранится в ней), либо определенная часть данных (инженер может иметь права для просмотра исходного кода системы, но не иметь доступа к информации о заработной плате сотрудника; финансовый аудитор может просматривать последнее, но не первое). Целостность означает, что никакие неавторизованные пользователи не могут писать данные; например, любой может свободно просматривать код Rails (веб-фреймворк Ruby), но лишь немногие могут напрямую вносить в него свои записи (хотя вы можете отправить запрос (pull request), чтобы имеющие необходимые права люди могли объединить код с вашими изменениями (merge)). По этим определениям очень легко создать безопасную систему — для этого создайте систему, раз-

бейте ее кувалдой, залейте бетоном и бросьте на дно Марианской впадины. Теперь никакие неавторизованные пользователи не смогут читать или записывать данные!

Конечно, также ни один авторизованный клиент не сможет ее использовать, и лишь немногие клиенты будут готовы платить за систему, к которой никто не может получить доступ. Для того чтобы предотвратить такое тривиальное, но бесполезное решение создания безопасных систем, необходим последний элемент триады InfoSec — доступность. То есть система должна быть доступна авторизованным пользователям для чтения и записи данных. Система, у которой при любых обстоятельствах присутствуют все элементы триады InfoSec, считается безопасной. На практике для большинства программ часто возникают особые обстоятельства, при которых один или несколько элементов отсутствуют.

Атаки на безопасность системы могут быть **активными** или **пассивными**. Активные атаки каким-то образом изменяют атакуемую систему, например путем добавления дополнительной программы, которая работает фоновом режиме, изменения пользовательских паролей или модификации данных, хранящихся в системе. Пассивные атаки не вызывают никаких изменений в системе. Примерами пассивных атак могут быть прослушивание сетевого трафика или мониторинг незащищенных беспроводных сетей. Хотя активные атаки часто наносят больший урон, их также гораздо легче обнаружить, тогда как пассивные атаки могут быть очень сложными для обнаружения.

Существует несколько видов атак, как пассивных, так и активных, на элементы триады InfoSec. Атака **прерывания** — это атака на доступность. Она уменьшает или уничтожает доступность заданной системы. Возможно, самой простой версией этой атаки будет проникновение в здание и отключение всех серверов от сети. Более сложные атаки включают отправку такого количества неавторизованных запросов, что авторизованные не могут быть обработаны (такая атака называется атакой **отказа в обслуживании** (denial of service)), или изменение паролей всех пользователей.

Атака **перехвата** — это атака на конфиденциальность. Она позволяет неавторизованному пользователю читать данные без их изменения, даже если у него нет прав на чтение. Тот факт, что напрямую данные изменять нельзя, не означает, что это менее серьезная форма атаки. Подумайте о том, с каким ущербом вы столкнетесь, если злоумышленник получит доступ к номеру вашей кредитной карты, номеру социального страхования и другой идентифицирующей информации. Простейшая атака перехвата заключается в подглядывании через плечо, когда некто вводит свой пароль. Существуют более технически продвинутые программы-**кейлоггеры** и аппаратура, которые сохраняют и/или передают любые нажатия клавиш, или **снифферы пакетов**, анализирующие проходящие в сети пакеты на предмет наличия в них паролей или других интересных данных.

Существуют два взаимосвязанных вида атак на целостность: атаки **модификации**, которые изменяют уже существующие данные, и атаки **фабрикации**, которые добавляют дополнительные данные в систему. Атака модификации может изменить текущий баланс подарочной карты в учетной записи на сайте электронной торгов-

ли, тогда как атака фабрикации может добавить учетную запись полностью вымышленного пользователя.

Если для системы существует один из способов организации атаки, значит, эта система имеет **уязвимость**. Например, предположим, в системе зарегистрирован пользователь-администратор DEFAULT с паролем DEFAULT. Если никто никогда не обнаружит эту уязвимость (хотя это маловероятно для системы даже с минимальным числом пользователей), то системе никогда не будет нанесен реальный ущерб. Однако если пользователь обнаруживает и использует это, тогда можно говорить об **эксплойте**. Эксплойт — это метод или механизм, используемый для компрометации элементов триады InfoSec некоторой системы. Эксплойты могут варьироваться от знания о наличии пароля по умолчанию до сложных программ, которые взаимодействуют с системой определенными способами, вызывая нежелательное поведение.

Существует огромное разнообразие этих инструментов, которые известны как **вредоносные программы** (malware). Вредоносное программное обеспечение — это любое ПО, которое специально создано для нежелательного воздействия на компьютерную систему, как правило, без ведома авторизованного пользователя системы. Приведем неполный список вредоносных программ.

1. **Бактерия** — программа, которая потребляет избыточное количество системных ресурсов, возможно, занимая все файловые дескрипторы или дисковое пространство.
2. **Форк-бомба** (Fork bomb) — особый вид бактерий, который постоянно разветвляется, в результате чего все ресурсы процессора расходуются на создание новых копий форк-бомбы.
3. **Логическая бомба** — код в программе, который выполняет несанкционированную функцию, такую как удаление всех данных в первый день месяца.
4. **Ловушка** (Trapdoor) — программа или часть программы, которая обеспечивает секретный доступ к системе или приложению.
5. **Троянский конь** — программа, которая притворяется другой, чтобы обманом заставить пользователей установить и запустить ее. Например, троянский конь может предлагать добавить забавные курсоры мыши, но после установки он удалит все на вашем диске.
6. **Вирус** — компьютерная программа, часто небольшая, которая воспроизводит себя при вмешательстве человека. Это может быть что-то вроде щелчка по ссылке или запуска программы, отправленной вам в виде вложения.
7. **Червь** — компьютерная программа, часто небольшая, которая воспроизводит себя без вмешательства человека. Например, после установки на компьютере червь может попытаться взломать другие компьютеры и скопировать на них свой код.
8. **Зомби** — компьютер с установленным программным обеспечением, которое позволяет неавторизованным пользователям получать доступ к нему для выполнения несанкционированных функций. Например, в систему может быть встроена почтовая программа, которая позволит другим пользователям отправлять спам

с вашего компьютера таким образом, что фактические отправители не могут быть отслежены.

9. **Бот-сеть** — множество зомби-компьютеров, управляемых неким администратором.
10. **Шпионское ПО** — программное обеспечение, которое тайно контролирует действия пользователя системы, например программа, которая ежедневно отправляет отчет обо всех клавишах, которые нажимал пользователь.
11. **Инструменты DoS** — инструменты, позволяющие реализовывать атаки отказа в обслуживании (Denial of Serivce).
12. **Вирусы-вымогатели** — программное обеспечение, которое выполняет нежелательные действия (например, шифрует ваш жесткий диск) и требует деньги или другую компенсацию за отмену этих действий. Деньги обычно идут создателям или пользователям программного обеспечения, а не самому программному обеспечению (искусственный интеллект пока не настолько продвинутый).

Вредоносные программы могут относиться к нескольким категориям. Например, программа может распространяться как компьютерный червь и, находясь на компьютере, сообщать о действиях пользователя, что делает его шпионским ПО. Другой случай — программа рекламируется как способная "очистить ваш реестр", но на самом деле она превратит ваш компьютер в зомби, находящийся под контролем бот-сети.

Для атаки на систему совершенно необязательно использовать вредоносное ПО. Подобно тому что существуют автоматизированные тесты и ручные тесты, существуют автоматизированные атаки и ручные атаки. Если я могу угадать пароль пользователя или отправить в текстовое поле строку, которая вызовет сбой системы, я атакую систему без какой-либо программы. Хотя в настоящее время многие из этих уязвимостей находятся и используются предназначенными для взломов инструментами, старые добрые ручные эксплойты по-прежнему в ходу.

20.3. Распространенные атаки и как использовать тестирование против них

20.3.1. Инъекция

При проведении **атаки-инъекции** злоумышленник пытается заставить ваш компьютер запустить некий предоставленный им произвольный код. Одним из наиболее распространенных типов инъекционных атак является **SQL-инъекция**, поскольку многие программисты, особенно новички, не занимаются **очисткой** входных данных. Очистка подразумевает, что введенные пользователем данные будут "очищены" и благодаря этому не будут выполняться напрямую. В качестве примера можно привести код, который запрашивает имя пользователя, получает строку с данными, ищет в базе данных всех пользователей с таким именем и возвращает уникальный идентификатор (uid) для первого пользователя с таким именем:

```
public int findUidByName(int neckLength) {
    Result dbResult = DatabaseConnection.executeSQL(
        "SELECT uid FROM users WHERE name = '" + name + "';");
    if (dbResult == null) {
        return NO_RESULTS_CODE;
    } else {
        return dbResult.get("uid");
    }
}
```

Это сравнительно простой метод, но в нем имеется очевидная проблема с безопасностью — ничто не мешает пользователю отправлять другие команды SQL вместо имени. Эти команды SQL будут выполняться машиной беспрекословно, даже если команды говорят об удалении базы данных. Представьте, что пользователь передает значение "`a'; DROP TABLE Users;`" в качестве своего имени. Будет выполнена следующая SQL-команда:

```
SELECT uid FROM users WHERE name = 'a'; DROP TABLE Users;
```

Если не верите мне, можете попробовать это со своей локальной базой данных, но в результате выполнения таблица пользователей `users` будет удалена. Это, вероятно, не то, чего вы ждете от того, кто ищет пользователей в вашем приложении! Существует множество других видов атак с использованием инъекций, например путем неправильного использования `eval()` в коде JavaScript или добавления `"OR 1=1"` к запросу SQL для отображения всех строк вместо одной, которую должен был найти код. Общим знаменателем в инъекционных атаках является то, что все они исполняют код, выполнения которого разработчики и администраторы системы не хотели бы.

Как уже говорилось, избежать этого можно путем очистки различными способами входных данных. Например, точка с запятой, некоторые спецсимволы и пробелы могут быть запрещены для использования в имени пользователя. Можно добавить проверку, что если в параметре `name` присутствовал какой-либо из этих символов, то возвращается результат `NO_RESULTS_CODE` до выполнения любого запроса SQL. Этот метод представляет собой простой черный список, и существуют более продвинутые способы предотвращения атак с использованием инъекций, но их рассмотрение выходит за рамки этой книги.

Как проверить, что инъекционные атаки невозможны? Самый простой способ — найти все места, где принимаются пользовательские данные, и убедиться, что отправка кода не приведет к его выполнению. Зачастую это оказывается формой тестирования серого ящика, т. к. не очень разумно проверять, скажем, что код COBOL выполнится в Java-системе. Тестирование белого ящика, включая юнит-тестирование, также будет полезно для гарантирования того, что в принимающих пользовательские данные методах осуществляется очистка этих данных или что методы, обращающиеся к базе данных, никогда не могут быть вызваны с кодом, который может использовать уязвимость. Существуют также инструменты статического анализа, которые могут статически проверять кодовую базу для защиты от возможных атак с использованием инъекций.

Стохастическое тестирование — особенно тестирование "злой обезьяны", которое работает с исполняемым кодом, — может быть полезно для больших систем, которые запрашивают и обрабатывают данные разными способами и в разных местах. Передавая большие объемы случайно сгенерированных данных, вы можете найти те виды данных, которые вызывают странную работу системы или ее сбои. Эти необычные события помогут вам определить, какие конкретно части системы уязвимы для атак с использованием инъекций. Например, если какой-то конкретный параметр обрабатывается с помощью `eval()`, ваше тестирование с помощью случайных данных может передать неверный код, что приведет к сбою тестируемой системы. Более тщательное изучение кода, который обрабатывает данные такого типа, позволит определить, возможны ли в нем инъекции кода.

20.3.2. Переполнение буфера

Что произойдет, если вы попытаетесь положить 10 кг в сумку, рассчитанную на 5 кг? **Переполнение буфера**, вот что. Во многих языках программирования вы должны выделять конечное пространство для хранения помещаемых в него данных. Например, в Java, если вы хотите хранить пять целых чисел в массиве, вы можете сделать что-то вроде этого:

```
int[] _fiveInts =  new int [5];
```

Теперь предположим, что у вас есть метод, принимающий строку целых чисел, разделенных запятыми (т. е. `7,4,29,3,2`), и затем помещающий каждое из этих целых чисел в массив `_fiveInts`:

```
public void putDataIntoFiveInts(String data) {
    int[] intData = data.split(",");
    for (int j=0; j < intData.length; j++) {
        _fiveInts[j] = intData[j];
    }
}
```

Метод успешно обработает переданную в него строку `7,4,29,3,2`. Однако если передать `1,2,3,4,5,6,7`, то при попытке записать шестое значение в `_fiveInts` будет выброшено исключение `ArrayIndexOutOfBounds`. Это может привести к сбою программы, если исключение не обрабатывается соответствующим образом. В некоторых языках, таких как C, исключение не выбрасывается, потому что нет **проверки границ** (проверка во время выполнения, что данные не записываются за пределы массива). Система без проблем будет записывать данные после конца массива, что может перезаписать исполняемый код или другую системную информацию. Если эти данные тщательно подготовлены, то злоумышленник может даже получить контроль над системой.

Проверить это можно, передав большие объемы данных во все места, где возможен ввод данных в систему. Объем передаваемых данных будет изменяться, и его часто можно определить, изучив код (т. е. тестирование "серого" или "белого ящика" оказывается очень эффективным при проверке данного вида уязвимости). Инструмен-

ты статического анализа также могут помочь при определении, где возможно переполнение буфера. Наконец, использование языка со встроенной проверкой границ, такого как Java, может помочь смягчить проблему. Обычно лучше, чтобы программа прекратила работу из-за возникновения исключения, чем из-за перезаписи кода или данных.

20.3.3. Неправильная настройка безопасности

Хотя ваша система может быть пуленепробиваемой в работе при правильной настройке, не все будут настраивать свои системы так же. Люди оставляют пароли по умолчанию, предоставляют всем права на чтение/запись потому, что так проще, или не включают двухфакторную авторизацию, потому что не любят, когда программа выдает им сообщения при каждой авторизации. В сложных программах люди могут пропустить один флажок, который шифрует данные, включает HTTPS, или требует, чтобы пользователи ввели пароль перед изменением данных.

Для того чтобы избежать этого, неплохо иметь разумный набор значений по умолчанию для настроек вашей программы и гарантировать, что пользователи понимают слабые места приложения, которое они настраивают. Они также должны легко определять, как исправить эти слабые места, которые они создали. Если единственный способ правильно настроить систему — это внимательно прочитать 500-страничное руководство по эксплуатации и установить некие непонятные параметры командной строки, то практически каждая поставляемая вами система будет настроена неправильно.

Тогда как же их протестировать? Зачастую вам придется проводить своего рода **пользовательское тестирование**. Оно подразумевает, что пользователь выполняет какую-то задачу, часто с минимальными рекомендациями команды разработчиков или их представителей или вообще без рекомендаций. Хотя это обычно делается для того, чтобы определить лучший пользовательский интерфейс системы, это также можно использовать для выяснения, как типичные пользователи конфигурируют систему и какие части системы они конфигурируют неправильно. Например, увидев, что пользователи часто забывают изменить пароль по умолчанию, разработчики могут добавить красное предупреждение ко всем административным страницам, напоминающее, что пароль по умолчанию еще не был изменен. Дальнейшее пользовательское тестирование может подтвердить, что такая мера вызывает желаемое изменение в поведении.

20.3.4. Небезопасное хранение

Даже если код вашей системы сам по себе безопасен: в нем отсутствуют все известный эксплойты, все входные данные очищаются, формально проверен на отсутствие переполнения данных — все это окажется слабым утешением, если данные не хранятся нужным образом! Примерами небезопасного хранения могут быть запись конфиденциальных данных в лог-файлы, предоставление пользователям прямого доступа к базе данных или хранение закрытых ключей в вашем коде, который располагается в общедоступном репозитории.

Обратите внимание, что проверка может оказаться сложнее, чем просто изучение лог-файлов или поиск прописанных в коде программы паролей. Например, в нормальных условиях в лог-файл отправляются только скучные отладочные данные, и поэтому может показаться, что, хотя лог является общедоступным, это не представляет большой угрозы безопасности. Однако если ошибка произойдет при обработке кредитной карты, выбрасывается исключение, которое содержит отладочные данные. В этих данных находится информация о кредитной карте, которую система пыталась обработать. И если исключение будет записано в лог-файл, тот факт, что этот файл является общедоступным, оказывается большой проблемой.

Тестирование на небезопасность хранения может быть таким же простым, как попытка обратиться к данным непосредственно в базе данных или в файловой системе. Тем не менее оно может оказаться более сложным в больших системах. В общем, необходимо следовать **принципу наименьших привилегий**, что означает установку для пользователей того минимального объема доступа, который им необходим для работы. Разработчики не требуют доступа к записям о персонале; точно так же руководителю отдела кадров не требуется доступ к исходному коду системы. Проверка небезопасного хранения может также включать автоматические проверки, запускаемые перед загрузкой кода в репозиторий, чтобы убедиться, что в нем нет никаких закрытых ключей, паролей или подобных секретов.

20.3.5. Социальная инженерия

А вот и король атак, самый распространенный способ получить доступ к любой системе — пройти через "уязвимость, которую невозможно исправить", через людей. **Социальная инженерия** включает в себя манипулирование людьми (часто авторизованными пользователями системы), чтобы в своих интересах заставить их выполнять действия, которые поставят под угрозу безопасность системы. В качестве примеров можно привести разговор с пользователем системы и убеждение, что некто из ИТ-отдела должен узнать его пароль для "планового обслуживания", или электронное письмо с поддельным полем "от", предлагающим пользователю запустить UNIX-команду `chmod -R 777 *` в его домашнем каталоге для начала тестирования.

Хотя все это может показаться читателям моего текста смешным (а если это не так, тогда напомните мне, чтобы я познакомил вас с одним свергнутым аристократом, который хочет отдать часть своих миллионов), социальная инженерия используется для доступа ко многим системам. Один из распространенных методов — **фишинг**, т. е. попытки получить личную или другую конфиденциальную информацию через e-mail или иным способом. Фишинговые письма обычно рассылаются на самые разные адреса электронной почты в надежде, что кто-то поверит им и попадется на удочку.

Фишинговые атаки часто кажутся плохо подготовленными, с неграмотным английским языком и техническими неточностями, но на самом деле это часть плана! Обычно после фазы первоначального контакта необходимо выполнить несколько шагов для достижения основной цели фишинговой атаки. Предположим, что некто

атакуемый получил электронное письмо, в котором говорилось, что его учетная запись электронной почты была взломана, и ему нужно щелкнуть здесь, чтобы подтвердить свой пароль (если вы читаете это не в Интернете, то ссылки работают не так хорошо, но не волнуйтесь, вы ничего не упустите). Если пользователь щелкнет по вредоносной ссылке, он попадает на созданную злоумышленниками страницу (которая может выглядеть точно так же, как реальная страница с информацией об учетной записи), которая позволяет им украсть пароль пользователя. В лучшем случае для злоумышленников пользователь не будет спрашивать об этом письме в своем ИТ-отделе; атакующим необходимо время для использования электронной почты в целях, которые обычно существуют у злонамеренных людей, крадущих чужие почтовые аккаунты. Другими словами, им нужны люди, которые не очень сознательны или технически грамотны, не замечают мелких проблем и доверяют всему, что видят перед собой. Это те самые люди, которые не обращают внимания на плохую грамматику и неточности исходного письма. Менее доверчивые люди потребуют больше работы для злоумышленников, поскольку они могут намеренно сообщать ложные данные или даже попытаться выследить рассылающих письма. Плохо написанное электронное письмо на самом деле является механизмом проверки.

Гораздо более опасным вариантом фишинга является **целевой фишинг**, при котором мишенью оказывается определенный пользователь. В этом случае атакующий прикладывает все усилия, чтобы пользователь ничего не заподозрил при прочтении письма. Для этого письмо составляется предельно тщательно: в письме будут упомянуты другие реально существующие пользователи, грамматика будет отличной (или, по крайней мере, подходящей), будет использоваться настоящее имя пользователя, заголовки письма будут подделаны так, как будто письмо исходит от босса данного пользователя, и т. д. Целевые фишинговые атаки очень сложные и требуют много времени для подготовки, но они также имеют тенденцию быть более эффективными; можете считать их высокоточными ударами по сравнению с ковровыми бомбометаниями обычных фишинговых атак. Даже опытные пользователи могут затрудниться с определением того, что письмо является поддельным.

Проверить, что социальная инженерия не будет работать в системе, сложно. В конце концов, здесь нет технических аспектов для проверки; программные составляющие системы будут работать правильно под управлением тех авторизованных пользователей, которые на самом деле выполняют приказы неавторизованных пользователей. Мы обсуждали сложность тестирования нечистых функций, которые могут полагаться на глобальные переменные или другое внешнее изменяемое состояние, но люди — это совершенно непредсказуемое внешнее состояние. Человек, который никогда не попадется на фишинговую почту, может быть обманут телефонным звонком; тот, кто обычно никогда не позволит себя во что-то втянуть, мог ранее пережить тяжелую ночь и в определенный момент не понимать все ясно; кто-то другой мог быть перегружен дополнительной работой и щелкнуть по вредоносной ссылке в письме.

Если учетные записи будут соответствовать принципу наименьших привилегий, это ограничит степень любого ущерба, который может быть нанесен злоумышлен-

ником, действующим через авторизованного пользователя. Например, обычный пользователь нашей системы Rent-A-Cat никогда не должен иметь доступа к системе расчета заработной платы сотрудников. Это поможет разделить атаку на данные и функционал, к которому имеет доступ жертва социальной инженерии. Проверка того, есть ли у пользователя какой-либо способ доступа к данным (чтение или запись), в которых он не нуждается, поможет найти возможные области, где этот доступ может оказаться опасным.

Также может быть полезно проведение тестов на людях! Некоторые компании рассылают поддельные фишинговые письма и узнают, сколько людей переходят по подозрительным ссылкам. Те, кто щелкает на ссылки, попадают на страницу с предупреждением, что это может стать причиной несанкционированного доступа к системе, и эти сотрудники в дальнейшем (надеюсь) вряд ли поступят так же, когда получат настоящее фишинговое письмо.

20.4. Тестирование на проникновение

Хотя я рассмотрел различные способы компрометации безопасности системы, часто лучший способ проверить систему — это думать как злоумышленник. При **тестировании на проникновение** пользователь, часто не входящий в группу разработчиков, пытается получить несанкционированный доступ к системе, используя любые имеющиеся в распоряжении средства (ограниченные условиями контракта, например "без удаления данных"). Это следует из теории "чтобы поймать вора, нужен вор" — те, кто действуют как люди, пытающиеся взломать вашу систему, будут наиболее эффективны при обнаружении любых дыр в ваших системах безопасности.

Во время теста на проникновение человек будет действовать так, как если бы он был взломщиком, пытающимся проникнуть в вашу систему. Он может собирать данные, чтобы определить, какие операционные системы или языки программирования вы используете, сканировать ваши сети, подбирать часто используемые пароли или использовать прочие тактики и методы, специфичные для вашей системы. Часто этим занимаются внешние консультанты или, по крайне мере, не связанные с разработчиками системы сотрудники, у которых отсутствует предустановленное понимание о том, как "должна" работать система. Весь *modus operandi* людей, пытающихся взламывать системы, заключается в том, что они манипулируют системами, чтобы делать то, что им "не положено" делать.

Затем тестировщик на проникновение разрабатывает отчет о слабых местах системы, а также о разветвлениях этих слабых мест. Например, он мог обнаружить уязвимость SQL-инъекции для одной подсистемы, которая позволяет получить доступ к конкретной базе данных. Хотя менеджеру это может показаться технической ерундой, объяснение разветвлений, что все данные о заработной плате компании окажутся доступны любому подключенному к Интернету человеку, сделает все более ясным (и страшным).

20.5. Общие рекомендации

Перед разработкой плана тестирования безопасности вы должны попытаться определиться, сколько тестирования необходимо. Это будет зависеть от области, в которой вы работаете, но в основном от рисков, которые могут возникнуть, если противник сможет взломать систему. Помните, что время и ресурсы, потраченные на тестирование безопасности, — это время и ресурсы, которые не тратятся на прочие работы. Если вы тестируете систему, которая контролирует коды запуска ядерного оружия, имеет смысл бóльшую часть времени тратить на тестирование безопасности системы; если вы запускаете стартап по аренде кошек, вероятно, потребуется меньше времени и меньше ресурсов.

Если вы работаете в регулируемой сфере, убедитесь, что вы соблюдаете все стандарты этой области. Например, в США, если вы храните медицинские данные, вы должны быть знакомы с HIPAA; если вы имеете дело с данными учащихся, вам следует ознакомиться с FERPA. Несмотря на то что они не являются основными критериями безопасности системы, это дополнительные параметры, о которых вам следует знать при определении ее безопасности.

Определите, какие наиболее важные аспекты системы следует защищать. Атаки, которые могут привести к перезаписи данных, в большинстве случаев будут более разрушительными, чем атаки, позволяющие получить доступ к данным. Компрометация некоторых составляющих системы может оказаться более разрушительной, чем других. Если сотрудниками ведется wiki, которая в основном полна шуток и расписаний игр в "Dungeons and Dragons", то она менее важна, чем информация о банковских счетах. Это не означает, что вики-страницы сотрудников следует игнорировать, но определенно нужно уделять больше внимания данным банковских счетов.

Также помните о принципе Парето. Выполняя общую проверку и не тратя много времени на каждый отдельный компонент, вы можете найти гораздо больше дефектов, затратив на это меньше усилий, нежели будете тратить месяцы на конкретную составляющую. Часто такими дефектами оказываются первые уязвимости, которые ищут потенциальные злоумышленники. Даже если в данной системе имеется сложная для использования уязвимость, большинство злоумышленников прекратит свою деятельность, если многие из распространенных атак окажутся неэффективными.

Наиболее эффективное тестирование безопасности — это то, которое выполняется. Если можете, отстаивайте те тесты безопасности, которые, по вашему мнению, будут не только лучшими для системы, но и смогут выполнять свою задачу снова и снова. В современном сетевом мире безопасность становится все более и более постоянной обязанностью, о чем знает любой, кто получил всплывающее окно с просьбой обновить свое программное обеспечение. Вам следует разработать план, который позволит осуществлять текущее обслуживание и проверку безопасности все время, пока система потенциально уязвима.

Глава 21

Взаимодействие с заинтересованными лицами

На протяжении большей части книги мы говорили о тестировании самом по себе. Вы пишете тесты, программа либо работает, либо нет (или вы понимаете, что написали тест неправильно), вы отправляете отчет о дефекте, и в конечном итоге проблема устраняется. Существует еще один аспект тестирования, с которым вы столкнетесь при работе над проектами, — человеческий фактор. Менеджеры могут не соглашаться с вами в отношении приоритетов тестирования. Разработчики могут возразить, что обнаруженный вами "дефект" на самом деле является ожидаемым поведением. Коллеги-тестировщики могут вступить с вами в спор из-за тонкостей в техническом задании.

Конфликт неизбежен в любой команде, и в целом, вероятно, это положительный момент. Если все в вашей команде всегда со всем соглашаются, будьте осторожны: вы, вероятно, работает над довольно тривиальным проектом, и ваша работа, скорее всего, в ближайшем будущем будет заменена небольшим программным скриптом. Разрешение конфликтов, хотя и может оказаться трудным, является ключом к успешной подготовке продукта к запуску. Конфликты, если они ведутся профессионально и продуктивно, часто являются топливом, которое с течением времени позволяет команде работать лучше.

Как тестировщик программного обеспечения вы столкнетесь с определенными проблемами при взаимодействии с другими членами команды разработчиков. В конце концов, работа тестировщиков — найти проблемы в программном обеспечении. Они являются профессиональными критиками, которые ищут слабые места системы, чтобы улучшить ее. Если это делается не легкими прикосновениями, то можно сойти за негодяя. Никому не нравится слышать о проблемах или недостатках программного обеспечения, которое они написали, а менеджерам не нравится слышать, что продукт не будет готов. В этой главе мы обсудим, как справиться с этими проблемами и успешно взаимодействовать со всеми людьми, нацеленными на завершение проекта.

21.1. Кто такие заинтересованные лица?

Заинтересованное лицо — это любой человек, который напрямую заинтересован в продукте. Конкретный вид интереса может варьироваться в зависимости от чело-

века и его роли. Например, покупатель продукта, т. е. человек, который платит за него, будет заинтересован в том, чтобы платить за продукт как можно меньше. Высшее руководство и маркетологи преследуют прямо противоположную цель — они хотят, чтобы клиенты платили как можно больше в пределах той дельты, при превышении которой покупатели выбирают конкурента. Разработчики могут быть заинтересованы в том, чтобы продукт был написан правильно либо использовался нужный язык программирования или фреймворк. Занимающиеся обеспечением качества сотрудники могут беспокоиться о качестве продукта; экспертов-консультантов волнует, чтобы продукт отвечал всем законодательным требованиям; и этот список можно продолжать.

Обратите внимание, что заинтересованные лица *прямо* заинтересованы в продукте. В философском смысле самые разные люди будут заинтересованы в продукте или, по крайней мере, окажут на него влияние. Например, владельцы кафе по соседству будут надеяться, что разработчики станут работать допоздна и, таким образом, начнут покупать больше такого сладкого-сладкого кофеина, который входит в состав кофе, и тем самым увеличивать прибыль кафе. Супруги тестировщиков могут хотеть, чтобы дефектов было меньше и тестировщики могли проводить больше времени со своей семьей. Тем не менее это не прямая заинтересованность в продукте, и так можно утверждать, что проект каким-то образом повлияет на любого человека (премьер-министр Люксембурга может надеяться на рост ВВП вашей страны, чтобы улучшить свои позиции на торговых переговорах, а успешный запуск вашего продукта поможет этой цели!). Хотя это может быть нечеткой границей, если человек не знаком с вашим продуктом напрямую, то он, скорее всего, не является заинтересованной стороной.

Как упоминалось выше, разные заинтересованные стороны будут беспокоиться о различных аспектах программного обеспечения. При обсуждении продукта с кем-то, кто принадлежит к иной, не вашей, группе заинтересованных лиц, держите в голове, что *они* думают, когда думают про ваш продукт. Поставьте себя на их место; разве обычному пользователю вашей системы важно, написано ли приложение на Haskell, Ruby или Java? Заботится ли высшее руководство компании из списка Fortune 500 о красивой симметрии иконок? Заботится ли разработчик о последнем квартальном отчете о доходах? Возможно, в некоторых случаях, имеется некое любопытство или косвенный интерес. В конце концов, если квартальная прибыль давно отрицательная, компания не сможет выплачивать зарплату разработчику, или же обычный пользователь может действительно беспокоиться о безопасности типов в языках программирования, используемых для создания приложений. Однако в целом люди больше беспокоятся о тех аспектах системы, которые их волнуют, и поэтому у них остается мало времени беспокоиться об остальном.

Ниже приводится неполный список групп заинтересованных лиц и общее представление о том, что может их волновать. В зависимости от области, в которой вы работаете, могут быть дополнительные классификации, или же некоторые из них могут быть объединены, но общее представление вы можете получить.

- *Заказчики* — это люди, которые будут платить за программное обеспечение. Они будут сосредоточены на получении работающей системы по низкой цене и будут заботиться о стоимости и окупаемости инвестиций в покупку.

- *Пользователи* — это люди, которые будут использовать программное обеспечение. Обратите внимание, что иногда это те же самые люди, что и заказчики, а иногда — нет. Например, когда учебное заведение приобретает ПО для записи на занятия, учебное заведение является заказчиком, а учащиеся — пользователями. Их будут беспокоить удобство использования системы и сомнения, позволит ли она им легко достичь своих целей.
- *Менеджмент проекта* — это люди, которые управляют конкретным проектом, следя за тем, чтобы он разрабатывался с надлежащей скоростью и выпускался вовремя и в рамках бюджета. Их будет беспокоить планирование (с точки зрения ресурсов, времени и объема работ) и качество программного обеспечения.
- *Высшее руководство* — обычно располагается на высоком уровне цепочки управления, обычно в той точке, где уже не очень хорошо знакомы с самим проектом. Высшее руководство заботится о финансовых показателях проекта, например о том, будет ли он приносить прибыль.
- *Разработчики* — те люди, которые пишут программное обеспечение. Они будут заботиться об используемых инструментах, о том, насколько хорошо программа работает с технологической точки зрения, насколько хорошо она решает проблему и соответствует требованиям.
- *Тестировщики/сотрудники отдела обеспечения качества* — люди, которые тестируют ПО и определяют качество системы. Они сосредоточатся на поиске дефектов в программном обеспечении и позаботятся о выпуске качественного продукта.
- *Сотрудники поддержки* — люди, которые поддерживают работу программного обеспечения, например выездные инженеры, сотрудники, отвечающие на обращения, и персонал по внедрению. Они будут заботиться о времени безотказной работы и качестве системы, а также о простоте ее использования клиентами.
- *Эксперты-консультанты* — эти люди сосредоточатся на оценке системы с точки зрения законов, гарантируя, что она соответствует всем требованиям, предъявляемым к ней законами той области, в которой она работает. Они позаботятся о соблюдении юридических или прочих задокументированных требований.

Это всего лишь примерная схема некоторых заинтересованных лиц, с которыми вы будете взаимодействовать в качестве тестировщика программного обеспечения. Помните, что она нарисована очень широкими мазками! Люди — это личности, а не безликие члены групп, чтобы ни говорил ваш друг, прочитавший слишком много философских книг. Хотя вы можете использовать эту информацию, чтобы получить общее представление о потребностях отдельных заинтересованных лиц, ничто не заменит реального взаимодействия и, главное, *вопросов*, что именно важно для них.

21.2. Отчеты и общение

При общении с заинтересованной стороной, чьи интересы не совпадают с вашими, вам следует обеспокоиться тем, чтобы понимать язык этого человека и что его вол-

нует. Вам следует позаботиться, чтобы говорить с ним на его языке. Но перед этим попробуйте встать на его место — сначала спросите себя, важно ли ему то, что вы ему хотите сказать. Если нет, подумайте, существует ли более важный вопрос, который вы хотите обсудить с ним. В некоторых случаях это все же необходимо — вашему руководителю может быть безразлична жалоба о домогательствах, но вы все равно обязаны сообщить о ней. Однако в других случаях в этом нет необходимости, и для кого-то, у кого много дел (как у большинства сегодня), это отнимает драгоценное время, которое можно было бы использовать для чего-то другого. Во-вторых, если их действительно волнует данная тема, постарайтесь выразить ее в терминах, которые они понимают и могут концептуализировать. Не рассматривайте разговор как способ доказать свой интеллект. Используйте его как способ предоставить ценную информацию в форме, которую может использовать другой человек, а также получить ценную информацию, которую вы можете использовать аналогичным образом.

Одна из ключевых пропастей в коммуникации находится между техническим и нетехническим персоналом. Если вы читаете эту книгу, и я говорю вам, что если вы используете алгоритм сортировки $O(n^2)$, то увидите экспоненциальное увеличение использования ОЗУ с большими наборами данных, у вас будет ясное представление о том, что я имею в виду. Возможно, вы даже знаете, как это исправить (конечно, найдя "эффективные алгоритмы сортировки"). Однако это предложение звучит словно греческий язык для нетехнического персонала (кроме тех, кто свободно говорит по-гречески и для кого есть какой-то другой язык, который они не знают). Однако именно так говорят разработчики, тестировщики и прочий технический персонал, потому что именно так вы начинаете в итоге думать, когда занимаетесь этим весь день и каждый день. Однако помните, что другие люди могут проводить большую часть своего дня, занимаясь бухгалтерским учетом, продажами, маркетингом или миллионом других вещей, и просто не иметь того понимания, что и вы, когда дело касается системы, над которой вы работаете.

Если вы общаетесь с теми, кто не разбирается в технических вопросах или не имеет знаний в предметной области тестируемого программного обеспечения, не забудьте объяснить вещи, которые могут быть важны для этих людей. Высшее руководство не заботится напрямую о том, какой алгоритм сортировки использует программа, но его волнует, что только малые предприятия смогут использовать программу, т. к. она не способна обрабатывать наборы данных более крупных компаний. Это то, что они могут понимать. Когда вы обсуждаете дефект с разработчиком, вы можете на профессиональном языке объяснить ситуацию более подробно.

Когда вы учитесь в школе, учителя, как правило, имеют больше знаний по изучаемой теме, чем вы. Вот почему они учителя. Однако в реальном мире ваши менеджеры часто имеют гораздо меньше знаний о конкретной системе или ее части, нежели вы, работающий над ней. Это может стать шоком — не в этом ли смысл существования начальников, чтобы кто-то разбирался в вопросе лучше, чем его подчиненные?

Не совсем. Задача менеджера — управлять. Это означает, что больше времени тратится на оценку, больше времени на решение кадровых вопросов, больше времени

на решение вопросов бюджета и т. д. Это оставляет очень мало времени на написание кода или даже на взаимодействие с системой вообще. Даже если менеджер владеет техническими вопросами, теперь он контролирует большую часть системы или даже всю систему и, таким образом, оказывается не в состоянии разобраться с отдельными особенностями той части системы, над которой работаете вы. Представьте себе человека, работающего в Департаменте транспортных средств, который занимается продлением водительских прав — назовем его Редли. Редли намерен заниматься продлением прав, что бы ни случилось. Кто-то переехал из другого штата? У кого-то проблема с документами? Или это водитель, у которого есть старые права, действие которых было приостановлено из-за уголовного наказания, но помилование губернатора позволило ему снова водить машину? Редли знает все действующие правила и законы, четко представляет, как их использовать, и действует быстро и эффективно.

А теперь представим Гувви, губернатора, который осуществил помилование. Технически Гувви является руководителем Редли (что-то вроде пра-пра-пра-начальника). Если поместить Гувви на место Редли, то весь ад вырвется наружу. Будучи исполнительным директором всего штата, Гувви должен был бы хорошо разбираться не только в делах Департамента транспортных средств, но и в экологических нормах, условиях ведения бизнеса, правительственном бюджете и многом другом. Гувви никоим образом не знает особенности продления водительских прав и того, как фотографировать кого-то, и это приведет к длинной очереди разъяренных автомобилистов в местном отделении Департамента.

Этот пример доведен до абсурда, но смысл его понятен. Как отдельный участник, вы будете сосредоточены на очень небольшой части более крупного проекта, но часто будете иметь весьма детальное представление об этой части. У менеджеров будет более широкий взгляд на проект или проекты, но у них будет меньше знаний о специфике. Чем дальше вы продвигаетесь по цепочке, тем больше это оказывается правдой; ваш непосредственный руководитель будет иметь хорошее представление о том, чем вы занимаетесь каждый день, но генеральный директор компании может даже не знать, что вы или ваш проект существуете.

Если у Редли появится возможность поговорить с Гувви о некоторых проблемах с процессом продления водительских прав, плохой идеей было бы начать с технических деталей правила 437-А, подраздела 4, параграфа С и его определения "резидента". Редли должен говорить в терминах, которые понимает Гувви, — о том, что раздражает избирателей и может повысить вероятность их голосования за Антигувви (который, конечно же, является противником на предстоящих губернаторских выборах). Гувви будет более восприимчивым к пониманию того, что закон не затрагивает людей, которые владеют домами в двух разных штатах, чем к каким-то юридическим тонкостям. Это очень важно для политиков, потому что раздраженные избиратели с меньшей вероятностью проголосуют за того, кто сейчас у власти (уважаемые политологи, пожалуйста, учтите, что я сильно упрощаю, поэтому не присылайте мне гневные письма о теории выборов губернаторов).

Точно так же, обсуждая систему или проблемы, с которыми вы сталкиваетесь, со своим начальником, сформулируйте их в понятных ему терминах. Помните, что

часто у них более широкий взгляд на систему, чем у вас, и то, что кажется важным вам, для них может быть не таким существенным. Фактически они могут даже не осознавать проблемы, которые вы считаете очевидными. Создание открытого потока информации может быть очень полезным в обоих направлениях, позволяя вам, как тестировщику, более целостным образом понять систему и соответствующе распределить свою энергию и время. Это также может дать вашим руководителям подробную информацию о данных, дефектах и проблемах, которые вы обнаружите в ходе тестирования.

21.3. Красно-желто-зеленый шаблон

Один из лучших способов, который я нашел для коммуникации с менеджерами, находящимися выше меня на тотемном столбе, — это красно-желто-зеленый шаблон. Это не мое изобретение, и он даже не уникален для индустрии программного обеспечения, но он оказался очень полезным во многих различных областях. Шаблон позволяет вам предоставлять высокоуровневый обзор о состоянии дел и в то же время дает возможность при необходимости углубиться в более подробные сведения. Помните, что менеджеры часто не обладают такой степенью детализации знаний о системе, которой обладаете вы, как тестировщик, поэтому перевод этих знаний в более простые термины позволит системе быть интеллектуально управляемой.

Если вы знакомы со светофорами, то знаете, что они состоят из трех огней: зеленого, означающего "идите", желтого, означающего "замедлитесь", и красного, означающего "стойте". Стандартный оконный диспетчер операционной системы OS X использует аналогичное соглашение: зеленый цвет означает "развернуть", желтый — "свернуть", красный — "закрыть". В каждом случае зеленый цвет означает, что вас ничто не останавливает (езжайте быстро, максимизируйте размер окна), желтый — что существуют некоторые препятствия для продвижения (цвет светофора скоро изменится на красный, окно свернуто), а красный означает, что имеется несколько более серьезных препятствий (движение идет в ортогональном направлении, окно закрыто).

Используя эту метафору, мы можем разделить систему на подсистемы или составляющие и пометить каждую из них красным, желтым или зеленым статусом. Зеленый статус указывает на отсутствие проблем — данная подсистема работает нормально и/или находится на пути к завершению работ над ней в соответствии с планом и бюджетом. Желтый статус указывает на то, что существует несколько проблем, но они сравнительно малы и их можно исправить, при этом подсистема может иметь пониженное качество или функциональность, или же могут потребоваться дополнительные ресурсы для своевременного завершения работ над ней. Красный статус указывает на серьезную проблему или проблемы, когда этой конкретной подсистеме нужна существенная помощь, чтобы систему можно было подготовить к нужному сроку, если это вообще возможно. Использование данных цветов позволяет глазу, который годами был натренирован на эту комбинацию (не говоря уже о "тренировке" эволюцией в течение миллионов лет), быстро получить общее представление о качестве системы. Помещение статуса в широкие категории

"красный — желтый — зеленый" означает, что вы не сможете получить детальное представление о статусе, но преимуществом оказывается то, что вам необходимо описать статус очень понятными словами. Люди, как правило, лучше справляются с ограниченным количеством вариантов статуса.

Разделение системы на подсистемы часто уже выполнено к этому этапу, например вы можете разделить их в зависимости от уже созданных тест-планов или от того, как разные задачи были распределены между участниками команды. Однако вы можете внести изменения в зависимости от того, что именно лучше подходит для отчетов или имеет логический смысл. Помните при взаимодействии с другими заинтересованными лицами, особенно когда вы пересекаете границу между техническим и нетехническим, что вы должны принимать во внимание аспекты, которые волнуют их, и уменьшать внимание на том, что волнует вас. Предположим, что ваш менеджер проекта разделил систему на подсистемы A, B и C. Вы разработали планы тестирования подсистем A, B, C и D, потому что вы чувствуете, что D имеет смысл как отдельная подсистема, а не как часть C. В большинстве случаев это означает, что ваш лучший способ действий — это сгруппировать ваши статусы D и C вместе в подсистему C.

Наряду с каждой подсистемой и ее статусом должен быть раздел с краткими примечаниями, в котором бы разъяснялось, *почему* был выбран именно этот статус. Этот раздел обычно состоит всего из нескольких предложений, но дает представление о возможных проблемах, которые могут возникнуть. Он также позволяет задать вопросы по теме и не полагаться целиком на автора статуса для понимания. Другое заинтересованное лицо может увидеть то, что тестировщику покажется незначительной проблемой, но на самом деле имеет большое значение для поставки системы.

Давайте рассмотрим несколько примеров, чтобы увидеть, как можно использовать красно-желто-зеленый шаблон, чтобы быстро демонстрировать статус менеджеру. Оба приведенных примера включают систему, которая принимает информацию о клиентах через API, сохраняет ее в базе данных, а затем отправляет еженедельные отчеты в отдел маркетинга по электронной почте:

```
-------------------------------------------------------------------------
Подсистема     Статус      Замечания
-------------------------------------------------------------------------
База данных    Зеленый     Тест-план завершен без дефектов.
                           Производительность соответствует или
                           превосходит все KPI.
Входной API    Зеленый     Последняя версия прошла все тесты, включая
                           все граничные случаи, без каких-либо дефектов.
Генерация      Желтый      При последнем запуске теста обнаружено
отчетов                    несколько арифметических ошибок. Может
                           потребоваться еще один разработчик для
                           завершения функционала в соответствии
                           с графиком, но, скорее всего, все останется как есть.
Email          Зеленый     Обнаружены два незначительных дефекта,
                           но они будут исправлены до завтра.
-------------------------------------------------------------------------
```

В этом первом примере у нас есть система, которая работает хорошо. Нет никаких известных серьезных проблем и нет опасений по поводу серьезного отставания от графика или сокращения объема работ. С первого взгляда мы можем увидеть море зеленого цвета с одним маленьким островком желтого цвета. Важно отметить, что понятно, почему эти подсистемы имеют такие статусы. Если заинтересованные лица хотят узнать больше о дефектах дизайна электронных писем, они могут задать больше вопросов, а если им интересно узнать, какие граничные случаи были проверены во входном API, они также могут сделать это.

Теперь давайте посмотрим на другую версию системы, которая, возможно, работает не так хорошо:

```
-------------------------------------------------------------------------
Подсистема     Статус       Замечания
-------------------------------------------------------------------------
База данных    Красный      База данных не может сохранять информацию
                            больше, чем об одном пользователе, и случайным
                            образом стирает данные каждый четверг.
Входной API    Желтый       Система обрабатывает каждый запрос
                            (размером один килобайт) несколько минут.
Генерация      Красный      Все отчеты содержат аббревиатуру "LOL",
отчетов                     повторяющуюся сотни раз, а ответственный за это
                            разработчик, похоже, улетел в Белоруссию.
Email          Красный      Как определила наша команда тестирования,
                            связанный с электронной почтой функционал
                            в настоящее время отсутствует, и никто из
                            разработчиков не планирует им заниматься.
-------------------------------------------------------------------------
```

Это не тот проект, которым бы я хотел управлять. Наше красивое зеленое море сменилось адским красным пейзажем. Даже единственная желтая подсистема, входной API, вероятно, потребует дополнительной работы, прежде чем будет готова к выпуску. Обратите внимание, что этот желтый статус сильно отличается от желтого статуса в предыдущем примере. Эти статусы очень широкие; наличие нескольких известных арифметических ошибок является проблемой, которую можно решить путем простых правок кода. Исправить описанную выше проблему с производительностью может быть очень трудно, но программа будет работать даже без исправления проблемы, только очень медленно. Помните, что статусы субъективны, и ваш подход к классификации подсистем будет зависеть от области системы, в которой она работает. Например, если программа дает один сбой в году, это может быть приемлемым для видеоигры, но абсолютно не годится для авиационного ПО.

Подводя итоги, можно сказать, что цель использования этого шаблона — сообщить другим о состоянии тестируемой системы. Он не предназначен для углубленного анализа и не должен заменять отчет о тестировании, это способ обобщенно показать, как работает система. Менеджеры и другие заинтересованные лица оценят возможность увидеть состояние системы в целом прежде, чем захотят углубляться в технические детали.

21.4. Отчеты о состоянии

Хотя красно-желто-зеленый шаблон полезен для демонстрации статуса системы, он менее эффективен при описании статуса тестировщика. Во многих местах работы ежедневный отчет о положении дел является обычным делом. Я нашел это очень полезным при работе с географически распределенной командой, особенно если в течение дня вы общаетесь мало (например, потому что части команды находятся в разных часовых поясах). Во многих командах гибкой разработки проводятся ежедневные встречи для обсуждения статуса, поэтому дополнительное электронное письмо будет излишним. Но в тех компаниях, где не проводятся подобные встречи, я обнаружил, что краткое электронное письмо в конце дня позволит менеджерам понять, чем занимаются все их сотрудники. Это также даст возможность руководителям узнать, имеются ли какие-либо проблемы, и осуществить проверку, чтобы убедиться, что их сотрудники работают эффективно и над нужными проблемами.

21.4.1. Пример: ежедневный отчет о состоянии дел для тестировщика программного обеспечения

Ежедневный статус: 7 августа 2015 года.

- Завершено выполнение тест-плана "Регрессия подсистемы базы данных" — записано как тестовый прогон 471, новых дефектов нет.
- Написал сценарий для преобразования старых тестовых данных с разделителями-табуляторами в новый формат с разделителями-запятыми, доступный в каталоге /scripts на сервере.
- Приступил к написанию тест-плана "Графическое отображение символов" — необходимо получить последнюю версию стандарта символов.

Блокеры: необходимо найти последнюю версию стандарта символов до завершения разработки тест-плана.

Это не займет много времени; нескольких предложений должно быть достаточно, а маркированные списки позволяют создать хорошее общее резюме вместо того, чтобы сосредоточиваться на написании рассказа. Это простое электронное письмо позволяет менеджеру узнать, над чем работает тестировщик (пишет новый план тестирования), какие дополнительные ресурсы доступны (скрипт для преобразования данных, который может быть полезен другим членам команды) и где тестировщику могут понадобиться дополнительные ресурсы (последняя версия стандарта символов).

Вооруженный этой информацией, менеджер может попросить членов команды переориентировать усилия или же сообщить своему собственному менеджеру о проблемах. Это поможет тестировщику справиться с любыми проблемами, с которыми он сталкивается; возможно, у них есть копия стандарта символов, которой тестировщик воспользуется. Наконец, команда сможет работать более эффективно, если

знаниями делятся таким образом. Если все участники команды воспользуются разработанным тестировщиком скриптом, они могут сэкономить многие часы, которые они потратили бы на обновление файлов тестовых данных вручную.

21.5. Замечание об управлении ожиданиями

У нового члена команды часто возникает желание пообещать больше, чем он может выполнить. Хотя это кажется способом завоевать сердца и умы, скорее всего, возникнут проблемы. Программные проекты часто разрабатываются с опозданием, и чертовски сложно точно оценить, сколько времени уйдет на создание программы. Менеджер проекта будет расстроен гораздо больше из-за того, что вы ему что-то пообещали и не выполнили, чем если вы пообещаете немного и сделаете немного. Кроме того, то, что вы будете испытывать меньше стресса при попытке достичь менее амбициозной цели, позволит облегчить достижение большего, т. к. менее вероятно, что вы совершите ошибку.

Вам нужно управлять ожиданиями. Люди будут думать, что если вы что-то не упоминаете, значит, с этим все хорошо. Это опасное предположение, но это также и человеческая природа, а идти против человеческой природы — верный способ разочароваться (например, взгляните на любой из проектов по созданию утопического общества в истории человечества). Таким образом, вам нужно начинать общаться раньше и общаться чаще. Если существуют проблемы, не дожидайтесь последней минуты, чтобы сообщить об этом другим заинтересованным лицам, надеясь, что вы сможете разрешить эти проблемы своими героическими усилиями. Разработка программного обеспечения — это не эпическая сказка, в которой благородный герой может победить злобные орды дефектов с помощью силы воли и мудрого помощника.

Один профессор однажды сказал мне: "Если ты сомневаешься, стоит ли общаться или нет, — общайся". Больше информации почти всегда лучше, чем меньше, особенно информации о проблемах. При разработке программного обеспечения могут возникнуть проблемы, и именно тестирование всегда обнаруживает дефекты (если бы этого не было, я бы давно потерял работу). Если у вас есть подходящие пути сообщить об этих обнаруженных дефектах, а также о проблемах и потребностях в ресурсах, другие заинтересованные лица будут благодарны за то, что их держат в курсе.

Это не означает, что вы повсюду должны распространять новости обо всем, что с вами происходит. Высшему менеджменту не нужно знать обо всех обнаруженных вами дефектах, а разработчикам программного обеспечения не нужно знать, куда вы ходили обедать. Используйте правильные каналы коммуникации, которые будут зависеть от команды, с которой вы работаете. Ежедневный отчет вашему непосредственному руководителю обычно очень полезен, потому что он позволит оценить со стороны то, что может быть важным для других заинтересованных лиц. Например, вы могли обнаружить нечто, что вам кажется незначительным дефектом, и просто упомянуть об этом в вашем ежедневном отчете, но ваш менеджер понима-

ет, что это может иметь далеко идущие последствия, и сообщит об этом всей команде.

21.6. Замечание о встречах

Очень мало кто по-настоящему наслаждается встречами, вероятно, за исключением тех, кто посещает курсы в бизнес-школе вместо того, чтобы прочитать книгу о тестировании программного обеспечения. Тем не менее они часто необходимы для определения наилучшего пути движения вперед и обеспечения того, что услышаны мнения всех релевантных заинтересованных сторон. Часто они также могут быть наиболее эффективным способом достижения консенсуса по какому-либо вопросу.

Это не означает, что ваше присутствие необходимо на всех встречах. Если вы поймете, что не получаете никакой полезной информации или не предоставляете достаточно информации — или, по крайней мере, недостаточно, чтобы оправдать потраченное на нее время, — вам следует задуматься о том, чтобы больше не посещать это собрание. Хотя может показаться самонадеянной попытка высказаться, что вам не нужно собрание, помните, что любое время, которое вы проводите на собрании, — это время, которое вы не тратите на что-то еще. Хороший менеджер поблагодарит вас за то, что вы обратили на это внимание, потому что это означает, что вы думаете о вашей работе стратегически, а не просто делаете то, что вас просят.

21.7. Разъяснение требований

Одним из самых частых и важных обсуждений, которые вы ведете, как тестировщик, — это уточнение требований. Записать, что должна делать программная система, довольно сложно. Практически невозможно создать нетривиальную систему, не найдя какую-либо двусмысленность. Тестировщики часто сталкиваются с этими неоднозначными областями, потому что они постоянно выходят за рамки возможностей системы, энергично проверяя граничные и угловые случаи. После обнаружения одной из этих неопределенных или недоопределенных областей им необходимо принять решение, каким должно быть ожидаемое поведение.

Если возможно, сделайте шаг назад и подумайте, как это требование работает в тандеме со всеми другими требованиями. Вы что-то упустили? Обретает ли это требование больший смысл в свете других требований? Можно ли сделать логический вывод о том, как должна работать система, если рассматривать ее в целом, а не сосредоточиваться на точной терминологии данного конкретного требования?

Если у вас все еще есть вопросы или вы полагаете, что обнаружили некоторую двусмысленность, вам следует обсудить проблему с соответствующими заинтересованными лицами. Эти заинтересованные лица будут различаться в зависимости от области программного обеспечения, над которым вы работаете, и типа команды, с которой вы работаете. Во многих крупных компаниях системные инженеры отвечают за определение требований и могут помочь вам понять, что они пытались до-

нести. Если это проблема дизайна, возможно, UI/UX-дизайнеры смогут вам помочь. Если вы работаете в стартапе или очень небольшой команде, вы можете даже напрямую спросить пользователей, что, по их мнению, должна делать программа в данных обстоятельствах.

Пытаясь уточнить требования, помните, что ваша задача как тестировщика — объективно проверить соответствие системы требованиям, а не создавать их! Это означает, что в целом вы должны исходить из того, что другие заинтересованные лица лучше понимают, каким должно быть ожидаемое поведение системы. Это не означает, что вы должны игнорировать свои мысли по данной теме или сидеть сложа руки, когда люди сообщают вам, что "*конечно*, подсистема-калькулятор должна давать сбой, когда кто-то введет число больше 24". Вы должны уважительно не соглашаться и рассказывать людям свое мнение и то, что вы узнали с технической точки зрения. Это обязательная составляющая профессионализма. Однако другие заинтересованные стороны также являются профессионалами в своих конкретных ролях. Те, чьи цели сосредоточены на *создании* требований, а не на их *тестировании*, часто будут более авторитетным источником ответов.

Наконец, если вы не можете найти лучшее решение, и его не знают другие заинтересованные стороны, вам, возможно, придется сделать собственное предположение относительно ожидаемого поведения. Это определенно не лучшее решение, но зачастую единственно возможное. Если никто не подумал о конкретном граничном случае, а вы поняли, что его нужно протестировать, вам нужно определиться с каким-то ожидаемым поведением. Убедитесь, что ваше предположение внутренне и внешне согласуется с другими требованиями системы и имеет смысл, по крайней мере, для вас как разумное поведение. Когда вы поступаете так, необходимо осознавать, что ваше предположение может быть неверным. Как минимум отметьте, какие предположения вы сделали в плане тестирования, или задокументируйте другим надлежащим образом. Вы можете сообщить другим заинтересованным лицам, какие предположения вы сделали и почему.

Помните, что перевод спецификации требований в реально работающую систему не является объективным и простым процессом. Если бы это было так, нам бы не понадобился технический персонал; нам бы просто нужно было поговорить о необходимой системе, и компилятор создал ее для нас. Однако английский (или любой другой естественный язык) является недостаточно точным, чтобы описать систему так, как требуется компьютеру. Даже если бы это было так, человеческий разум, возможно, не смог бы охватить перечень из тысяч, если не миллионов требований, которые могла бы повлечь за собой такая система. Общение с другими людьми попрежнему будет важной частью тестирования программного обеспечения.

21.8. Этические обязательства

Как у тестировщика программного обеспечения у вас имеются обязательства по выполнению вашей работы этично и честно. Кто-то может возразить, что работа — это просто работа, и критерий успеха — просто делать то, что от вас ожидает начальство. Однако я бы сказал, что должность тестировщика — это профессия.

У профессии существуют обязательства, выходящие за рамки работы сотрудника в конкретной компании. Например, ни один (уважаемый) психиатр не станет нарушать конфиденциальность пациента, чтобы облегчить свою работу. Юрист не должен выставлять счет за часы, которые он не отработал, даже если это принесет его фирме больше денег.

Точно так же должен действовать и тестировщик, когда этого потребуют обстоятельства. Если вы определили, что программа хранит личную информацию небезопасным образом, но ваш менеджер приказал игнорировать это, чтобы программу можно было выпустить как можно раньше, что же вам следует делать? Если вы думаете, что команда тратит слишком много времени на добавление новых функций, но не на исправление дефектов, стоит ли вам сообщить об этом своему руководителю? Предположим, что само программное обеспечение делает что-то тайком, например шпионит за пользователями или рассылает нежелательные коммерческие предложения по электронной почте миллионам людей. Стоит ли вам продолжать работать в компании или над программным обеспечением? Следует ли сообщать об этом средствам массовой информации или правоохранительным органам? Конечно, возможно, что потребуется оплата за следование моральной линии в виде дополнительной нагрузки, потери доверия со стороны ваших коллег или руководства, а может, даже потери работы. Хотя в учебнике легко читать про этические дилеммы и обсуждать их, но совсем другое дело, когда вашему ребенку будет нечего есть, если вы не продолжите делать что-то морально сомнительное.

Это трудный путь, и здесь нет простых ответов. Если вы будете ждать того момента, когда определите, что в программном обеспечении нет абсолютно никаких дефектов, которые могли бы вызвать проблему, то программное обеспечение никогда не будет выпущено. Если вы действуете как беспристрастный наблюдатель, сухо отмечая в сноске, что продукт может легко привести к смерти, то вы действуете неэтично — даже если вы следовали букве правил.

Чего вы не можете сделать, так это игнорировать этические затруднения, связанные с тестированием программного обеспечения. Помимо того, что вы сталкиваетесь с повседневными этическими вопросами разработки ПО, такими как правильная оценка времени, отказ от фальсификации отчетов о тестах, вы, по всей вероятности, столкнетесь с более сложно разрешаемыми проблемами. Я рекомендую вам хотя бы немного подумать о вашем этическом кодексе прежде, чем вы столкнетесь с этими ситуациями в реальности.

21.9. Уважение

Возможно, вы заметили, что эта тема полна оговорок. На это есть причина — отношения с людьми полны сложностей и двусмысленностей. Для сравнения можно сказать, что работа с программным обеспечением подобна прогулке по парку. Профессионал обязан справляться с этими трудностями с изяществом и уверенностью, и очень сложно составить для этого пошаговые инструкции или шаблон.

Большую часть этой главы можно резюмировать фразой "будьте уважительны". Помните, что другие заинтересованные стороны будут заботиться об иных аспектах

системы или могут иметь другое представление о наилучшем пути вперед. Они могут не понимать всего, что делаете вы, а вы можете не понимать ничего из того, что делают они. У них может не быть такого же технического образования или знаний в предметной области, как у вас, что приводит к различным взглядам на одни и те же аспекты.

Для завершения программы требуется более одной заинтересованной стороны, а значит, существуют основания для разногласий. Вы вряд ли измените чье-то мнение, обращаясь с ним неуважительно или снисходительно, а небольшие кусочки плохого взаимодействия могут отравить весь колодец. Помните, что вы смотрите на проблему с очень определенной точки зрения, и у вас не всегда есть все ответы. Если вы относитесь к людям с уважением и беспристрастно выслушиваете то, что говорят заинтересованные стороны, вы можете даже изменить свое мнение. Даже если вы это не сделаете, выстраивание мирного диалога между членами команды, особенно теми, с которыми вы не согласны, будет иметь большое значение для успешного выпуска программного продукта.

Глава 22

Заключение

Надеюсь, через несколько десятков тысяч слов вы кое-что узнали о тестировании программного обеспечения. Вам пришлось бы изрядно постараться, чтобы вообще ничего не узнать за то время, пока вы это читали. Однако вы только прошлись по поверхности. Многие темы были рассмотрены лишь бегло, хотя они заслуживают гораздо большего. Было бы легко посвятить всю книгу тестированию производительности (которому я отвел только одну главу), тестированию на проникновение (ему я посвятил раздел) или формальной верификации (о которой я упоминал лишь пару раз). Тестирование программного обеспечения в любой конкретной области сопряжено со своим набором проблем. Если вы примете участие в тестировании какого-либо нетривиального приложения, вы, вероятно, сможете написать немаленькую книгу по специфическим для него угловым случаям.

Как я уже упоминал во введении, эта книга была задумана как обзорный тур по теме. Никто не думает, что понял Париж, проведя там день и не увидев ничего, кроме Эйфелевой башни. Я призываю вас продолжать исследования по интересующим вас темам и быть в курсе событий в области тестирования программного обеспечения. Как и в любой другой области, здесь также постоянно меняются и обновляются лучшие практики, и мы ищем лучшие способы решения задач, лучше понимаем теорию и разрабатываем лучшие инструменты.

Гарантирование качества программного обеспечения — благородное дело, и существует много способов его реализации. Я надеюсь, что ваш аппетит к качеству программного обеспечения разгулялся, и вы будете двигаться дальше, помогая разрабатывать максимально свободные от неизвестных дефектов программы, насколько это возможно. Я соглашусь, что это странное и необычное желание, но что я могу сказать? Я странный и необычный парень.

Глава 23

Шаблоны тестирования

Это справочник по различным шаблонам, используемым в книге.

23.1. Шаблон тест-кейса

ИДЕНТИФИКАТОР: уникальный идентификатор для данного тест-кейса, который в идеале также будет служить простым способом запомнить, что тестируется в тест-кейсе. Пример: VALID-PARAMETER-MESSAGE.

ТЕСТ-КЕЙС: краткое описание того, что делает тест-кейс.

ПРЕДУСЛОВИЯ: любые условия, которые должны быть выполнены перед выполнением тест-кейса.

ВХОДНЫЕ ЗНАЧЕНИЯ: любые входные значения, которые должны быть переданы как часть шагов выполнения.

ШАГИ ВЫПОЛНЕНИЯ: шаги, которые должен предпринять тестировщик для выполнения теста.

ВЫХОДНЫЕ ЗНАЧЕНИЯ: любые конкретные выходные значения, ожидаемые после шагов выполнения.

ПОСТУСЛОВИЯ: любые условия, которые должны выполняться после того, как завершены шаги выполнения. Если эти условия не выполняются, тест не проходит.

23.2. Шаблон отчета о дефектах

РЕЗЮМЕ: краткое (одно предложение или меньше) описание дефекта.

ОПИСАНИЕ: более подробное (абзац или более) описание дефекта.

ШАГИ ВОСПРОИЗВЕДЕНИЯ: конкретные шаги по воспроизведению дефекта.

ОЖИДАЕМОЕ ПОВЕДЕНИЕ: что ожидается после выполнения шагов воспроизведения.

НАБЛЮДАЕМОЕ ПОВЕДЕНИЕ: что на самом деле произошло после того, как были выполнены шаги воспроизведения.

ВЛИЯНИЕ: как это конкретно влияет на пользователя программного обеспечения.

СЕРЬЕЗНОСТЬ: насколько серьезна эта проблема, от ТРИВИАЛЬНОЙ до БЛОКЕРА.

ВОЗМОЖНОЕ РЕШЕНИЕ: как избежать появления этого дефекта, если известно или возможно.

ПРИМЕЧАНИЯ: любые другие примечания, которые могут быть полезны при исправлении или отслеживании этого дефекта, такие как конфигурация системы, дампы потоков или лог-файлы.

23.3. Красно-желто-зеленый шаблон

Систему следует разделить на разумное количество, обычно от трех до десяти, подсистем или областей функциональности. Затем каждой подсистеме или области следует присвоить "цветовую оценку": красную, желтую или зеленую, а также краткое описание (максимум несколько предложений) того, почему была дана эта оценка.

КРАСНЫЙ: этот аспект системы имеет серьезные проблемы, и не должен быть выпущен в таком состоянии. Потребуется существенная дополнительная помощь в виде ресурсов, пересмотра объема работ или увеличения графика, чтобы подготовить его к выпуску.

ЖЕЛТЫЙ: имеется несколько проблем с этим аспектом системы, некоторые из них существенные. Для получения приемлемого уровня качества может потребоваться дополнительная помощь.

ЗЕЛЕНЫЙ: с данным аспектом системы нет серьезных проблем, и для его завершения не требуется дополнительная помощь.

23.4. Ежедневный отчет о состоянии

Это шаблон для сообщения о вашем ежедневном статусе менеджеру или другому руководителю.

ДАТА

- Над чем вы работали сегодня, и соответствующий статус. Здесь могут быть несколько разных подпунктов.
- Над чем вы планируете работать завтра. Здесь также могут быть несколько подпунктов.

БЛОКЕРЫ: все, что мешает вам выполнять работу, будь то ресурсы, время или знания.

Глава 24

Использование рефлексии для тестирования *private*-методов в Java

В Java нет возможности напрямую вызывать `private`-методы из юнит-теста, хотя так можно делать в других языках (например, в Ruby метод `.send (:method_name)` полностью обходит понятие `private`). Однако, используя библиотеку рефлексии, мы можем "отразить" структуру выполняемого класса. Библиотека рефлексии встроена в язык Java, поэтому для ее использования не нужно ничего устанавливать.

Приведем пример. Если вы раньше никогда не работали с рефлексией, она может показаться немного странной. Скажем, мы хотим написать класс, который говорит пользователю, какие методы доступны в этом классе. Без рефлексии в Java это невозможно; как вам узнать, какие методы существуют без жесткой записи их в `String` или реализации чего-то подобного? На самом деле, это сравнительно просто сделать с использованием рефлексии:

```
import java.lang.reflect.Method;
public class ReflectionFun {
    public void printQuack() {
        System.out.println("Quack");
    }

    public static void main(String[] args) {
        Metod[] methods =ReflectionFun.class.getMethods ();
        // Получить все методы из класса и из суперклассов,
        // вызываемых в этом методе
        System.out.println("All methods:");
        for (Method method : methods){
            System.out.println(method.getName());
        }
    }
}
```

Когда мы запустим эту программу, то увидим следующее:

```
All methods:
main
printQuack
```

```
wait
wait
wait
equals
toString
hashCode
getClass
notify
notifyAll
```

Затем мы можем решать такие задачи, как проверка существования метода перед его вызовом, или дать знать программисту, какие методы существуют. Если вы когда-либо использовали такой язык, как Ruby, в котором можно быстро проверить, какие методы доступны для объекта, вы можете понять, насколько это полезно. Если вы новичок в написании кода и хотите сделать что-то, связанное с кряканьем, но не уверены, называется ли тот метод, который вы хотите вызвать, `displayQuack()`, `quackify()`, `quackAlot()` или как-то еще, вы можете быстро просмотреть список методов и увидеть, что метод, который вы ищете, — это `printQuack()`.

Возможно, вы заметили, что мы получили гораздо больше методов, чем указано в классе `ReflectionFun`. Это связано с тем, что метод `getMethods()` возвращает список *всех* методов, вызываемых для объекта (т. е. общедоступных `public`-методов, мы скоро узнаем, как получить `private`-методы). Так как все объекты в Java наследуются от класса `Object`, в выдачу попали общедоступные методы класса `Object`.

Вы также заметили, что в списке перечислены три различных метода `wait`. Причина этого в том, что Java рассматривает одноименные методы с разными списками аргументов как разные методы. Изучая Java API, мы увидим, что существуют три следующих метода:

```
public void wait();
public void wait(long timeout);
public void wait(long timeout, int nanos);
```

Изучая приведенный выше код, вы можете увидеть, что массив `methods[]` на самом деле содержит методы как объекты! Здесь нет ничего необычного для функционального языка, но если вы привыкли к прямому программированию Java, это может показаться немного странным. Если концепция метода или функции, существующей как объект, кажется вам подозрительной, моя первая рекомендация — изучать Haskell или другой функциональный язык программирования, пока это не станет для вас привычным. Если у вас нет на это времени, просто думайте о них как о функциях, которые вы можете взять и носить с собой, чтобы сделать что-нибудь с ними позже, вместо того, чтобы держать их в одном месте.

Теперь, когда у нас есть этот список методов, мы можем вызывать их по имени, передав имя метода, для которого мы хотели иметь ссылку, на метод `getMethod()`:

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
```

```
public class ReflectionFun {
  public void printQuack() {
      System.out.println("Quack!");
  }

  public static void main(String[] args) {
    try {
        System.out.println("Call public method (printQuack):");
        Method method = ReflectionFun.class.getMethod("printQuack");
        ReflectionFun rf = new ReflectionFun();
        Object returnValue = method.invoke(rf);
    } catch (NoSuchMethodException|IllegalAccessException|InvocationTargetException ex) {
        System.err.println("Failure!");
    }
  }
}
```

После выполнения этого кода получим:

```
Call public method (printQuack):
Quack!
```

Используя это, вы можете создать программу для ручного тестирования и вызова методов, предлагая пользователю ввести строку и попытаться вызвать метод с этим именем для объекта. Теперь у нас есть контроль во время выполнения над тем, какие методы вызывать. Это очень полезно для метапрограммирования и таких интерфейсов программистов, как REPL (системы read-eval-print-loop, которые позволяют вам ввести некоторый код, посмотреть результаты и повторить). Теперь, когда вы понимаете рефлексию, достаточно внести незначительные изменения в наш существующий код, чтобы мы могли легко получить доступ к `private`-методам и протестировать их.

Вы не можете использовать методы `getMethod()` или `getMethods`, поскольку они возвращают только общедоступные методы. Вместо этого вам нужно применять методы `getDeclaredMethod()` или `getDeclaredMethods()`. У них имеются два ключевых отличия от рассмотренных выше `getMethod()` или `getMethods()`:

1. Они возвращают только методы, объявленные в этом конкретном классе. Они не будут возвращать методы, определенные в суперклассах.
2. Они возвращают методы `public`, `private` и `protected`.

Следовательно, если нам нужен список *всех* методов, определенных в ReflectionFun, мы могли бы использовать метод `getDeclaredMethods()`. Забавы ради, давайте добавим `private`-метод `printQuock()` вместе с нашим `public`-методом `printQuack()` (мое определение "забавы" может немного отличаться от вашего):

```
import java.lang.reflect.Method;
public class ReflectionFun {
    public void printQuack() {
        System.out.println("Quack!");
    }
```

```
    private void printQuock() {
        System.out.println("Quock!");
    }

    public static void main(String[] args) {
        System.out.println("Declared methods:");
        Method[] methods = ReflectionFun.class.getDeclaredMethods();
        for(Method method : methods){
             System.out.println(method.getName());
        }
    }
}
```

Наша программа выведет:

```
Declared methods:
main
printQuack
printQuock
```

У нас снова есть список объектов Method, и теперь мы можем вызывать их. Но есть одна небольшая загвоздка — нам нужно сперва сделать этот метод "доступным" перед его вызовом при помощи метода `setAccessible()`. Он принимает булев параметр, чтобы определить, может ли метод быть доступен вне класса:

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class ReflectionFun {
  public void printQuack() {
      System.out.println("Quack!");
  }
  private void printQuock() {
      System.out.println("Quock!");
  }
  public static void main(String[] args) {
    try {
        System.out.println("Call private method (printQuock):");
        ReflectionFun rf = new ReflectionFun();
        Method method2 = ReflectionFun.class.getDeclaredMethod("printQuock");
        method2.setAccessible(true);
        Object returnValue = method2.invoke(rf);
    } catch (NoSuchMethodException|IllegalAccessException|InvocationTargetException ex) {
        System.err.println("Failure!");
    }
  }
}
```

Результат работы программы:

```
Call private method (printQuock):
Quock!
```

Мы можем объединить это с другим юнит-тестированием, которое мы изучили ранее, чтобы написать юнит-тест для `private`-метода:

```
public class LaboonStuff {
    private int laboonify(int x) { return x; }
}

@Test
public void testPrivateLaboonify() {
  try {
      Method method = LaboonStuff.class.getDeclaredMethod("laboonify");
      method.setAccessible(true);
      LaboonStuff ls = new LaboonStuff();
      Object returnValue = method.invoke(ls, 4);
      int foo = ((Integer) returnValue).intValue();
      assertEquals(4, foo);
  } catch (NoSuchMethodException|IllegalAccessException|InvocationTargetException ex)   {
      // Метод не существует
      fail();
  }
}
```

Совершенно очевидно, что тестирование даже простых `private`-методов может включать довольно много шаблонного кода. Во многих случаях вы, вероятно, захотите обернуть код доступа к `private`-методу в отдельный вспомогательный метод.

Глава 25

Что еще почитать

Эта глава содержит подборку книг по различным темам, которые могут вас заинтересовать, или если вам захочется углубиться в какую-либо из тематик, затронутых в этой книге. В конце концов, это было всего лишь знакомство с миром качества программного обеспечения; предполагалось, что оно широко охватит множество областей. Если же вас интересует какая-то конкретная область, настоятельно рекомендую копнуть глубже.

Исследовательское тестирование

Hendrickson Elisabeth. Explore It! Reduce risk and increase confidence with exploratory testing. — Dallas: The Pragmatic Bookshelf, 2013.

Общее тестирование

Crispin Lisa, Gregory Janet. Agile testing: a practical guide for testers and agile teams. — Boston: Addison-Wesley, 2009.

Jorgensen Paul C. Software testing: A craftsman's approach. — Boca Raton: CRC Press, 2014.

McCaffrey James. Software testing: Fundamental principles and essential knowledge. — Charleston, 2009.

Whittaker James. How to break software. — New York: Pearson, 2002.

Whittaker James, Arbon Jason, Carollo Jeff. How Google tests software. — New York: Pearson, 2012.

Интеграционное тестирование

Duvall Paul, Matyas Stephen M. III, Glover Andrew. Continuous integration: improving software quality and reducing risk. — Boston: Addison-Wesley, 2007.

Факты и история

Kidder Tracy. The soul of a new machine. — Boston: Little, Brown and Company, 2000.

Levy Steven. Hackers: heroes of the computer revolution. — New York: Penguin Books, 2000.

Mitnick Kevin. Ghost in the wires: my adventures as the world's most wanted hacker. — Boston: Little, Brown and Company, 2011.

Stoll Clifford. The Cuckoo's egg: tracking a spy through the maze of computer espionage. — New York: Doubleday, 2012.

Zachary G. Pascal. Showstopper! The breakneck race to create Windows NT and the next generation at Microsoft. — New York: Open Road Media, 2014.

Тестирование производительности

Molyneaux Ian. The art of application performance testing: help for programmers and quality assurance. — Sebastopol, CA: O'Reilly and Associates, Inc., 2009.

Риски разработки программного обеспечения

Neumann Peter G. Computer-related risks. — New York: ACM Press, 1995.

Тестирование безопасности

Garfinkel Simson, Spafford Gene, Schwartz Alan. Practical UNIX and Internet security. — Sebastopol, CA: O'Reilly and Associates, Inc., 2003.

Mitnick Kevin, Simon William L. The art of deception: controlling the human element of security. — Hoboken, NJ: John Wiley & Sons, 2007.

Weidman Georgia. Penetration testing: A hands-on introduction to hacking. — San Francisco: No Starch Press, 2014.

Разработка программного обеспечения, ориентированного на качество

Brooks Frederick P. The mythical man-month: essays on software engineering. — 2nd ed. — Boston: Addison-Wesley, 1995.

Feathers Michael C. Working effectively with legacy code. — Upper Saddle River, NJ: Prentice Hall, 2004.

Fowler Martin. Refactoring: improving the design of existing code. — Addison-Wesley, 1999.

Hunt Andrew, Thomas David. The Pragmatic programmer: from journeyman to master. — Boston: Addison-Wesley, 1999.

Martin Robert C. Clean code: a handbook of agile software craftsmanship. — Upper Saddle River, NJ: Prentice Hall, 2008.

McConnell Steve. Code complete: a practical handbook of software construction. — Second Edition. — Seattle: Microsoft Press, 2003.

Взаимодействие с заинтересованными лицами

Stone Douglas, Patton Bruce, Heen Sheila. Difficult conversations: how to discuss what matters most. — New York: Penguin Books, 2010.

Разработка на основе тестов

Beck Kent. Test-driven development: by example. — Boston: Addison-Wesley, 2003.

Chelimsky David, Hellsoy Aslak. The RSpec book: behaviour-driven development with RSpec, cucumber, and friends. — Dallas: The Pragmatic Bookshelf, 2010.

Freeman Steve, Pryce Nat. Growing object-oriented software, guided by tests. — Boston: Addison-Wesley, 2009.

Тестирование дружественности

Krug Steve. Don’t make me think: a common sense approach to web usability. — Berkeley, CA: New Riders Publishers, 2006.

Веб-тестирование

Rappin Noel. Rails 4 test prescriptions: build a healthy codebase. — Dallas: The Pragmatic Bookshelf, 2014.

Глава 26

Словарь терминов

9 — *см.* девять девяток.

Ad-hoc тестирование (ad hoc testing) — термин (иногда считающийся уничижительным) для исследовательского тестирования.

DDos — *см.* Распределенный отказ в обслуживании.

DoS (denial of service) — отказ в обслуживании.

DRY ("Don't Repeat Youself") ("Не повторяйся") — принцип написания хорошего тестируемого кода, который гласит, что код не должен повторяться, например нельзя иметь два разных метода, которые делают одно и то же, или копировать/вставлять код из одной части кодовой базы в другую вместо того, чтобы превратить его в вызываемый метод.

expect (expect) — программа, которая позволяет автоматизировать взаимодействие с программами командной строки.

IDE (Integrated Development Environment) — интегрированная среда разработки. Единый инструмент, объединяющий большую часть функций, позволяющих разработчику, например, создавать тесты, выполнять компиляцию и настраивать зависимости.

KPI (Key Performance Indicator) — ключевой показатель производительности.

Linter — инструмент статического анализа, информирующий пользователя о потенциальных проблемах с кодом.

MTBF (mean time between failures) — среднее время между отказами.

MTTR (mean time to repair) — среднее время восстановления.

***n* девяток** (*n* nines) — способ показать, какой процент времени доступна система, основывается на количестве девяток в этом числе, и при условии, что девятки являются единственной значащей цифрой. Например, система, которая доступна 99,92% времени, имеет доступность 3 девятки, а система, которая доступна 99,999% времени, имеет доступность 5 девяток.

QA (quality assurance) — *см.* обеспечение качества.

***r*-комбинация** (r-combination) — **подмножество** (subset) *r* элементов из **набора** (set), порядок которых не имеет значения. Значение *r* можно заменить определенным значением, например 2 или 3, чтобы указать количество элементов в подмножестве. Например, [2, 3, 1] представляет собой трехкомпонентную комбинацию набора [1, 2, 3, 4] и эквивалентно [1, 2, 3].

***r*-перестановка** (r-permutation) — **подмножество** (subset) *r* элементов из **набора** (set), порядок которых имеет значение. Значение *r* можно заменить определенным значением, например 2 или 3, чтобы указать количество элементов в подмножестве. Например, [2, 3, 1] и [1, 2, 3] — это две разные трехкомпонентные перестановки множества [1, 2, 3, 4].

SLA (service level agreement) — *см.* соглашения об уровне обслуживания.

Soak-тест (soak test) — еще один термин для **теста стабильности** (stability test).

Tcl (Tool Command Language) — язык программирования, используемый для управления программой expect, которая используется для автоматизации системных тестов.

TUC (test-unfriendly construct) — *см.* недружественная к тестированию конструкция.

TUF (test-unfriendly function) — *см.* недружественная к тестированию функция.

YAGNI (You Ain't Gonna Need It) — "Тебе это не понадобится". Принцип **разработки через тестирование** (test-driven development), который гласит, что вы не должны выполнять работу или добавлять функции, которые не нужны немедленно.

Активная атака (active attack) — при тестировании безопасности — атака на систему, которая вызывает некоторые изменения в системе, такие как добавление программы или изменение данных в базе данных.

Альфа-тестирование (alpha testing) — тестирование "в реальном мире" небольшой группой инженеров-тестировщиков или другой небольшой группой технически опытного персонала. Часто предшествует бета-тестированию.

Анализ пакетов (сниффинг) (packet sniffing) — использование **анализатора пакетов** (packet analyzer) или аналогичного программного обеспечения для просмотра данных, передаваемых по сети.

Анализатор пакетов (packet analyzer) — инструмент, позволяющий просматривать отдельные пакеты, передаваемые по сети.

Атака с использованием SQL-инъекции (SQL injection attack) — особый распространенный вид **атаки с использованием инъекции** (injection attack), когда злоумышленник пытается заставить систему выполнить произвольные команды SQL.

Атрибут качества (quality attribute) — еще один (возможно, лучший) термин **нефункционального требования** (non-functional requirement).

Баг (bug) — еще один термин для обозначения дефекта.

Базовый случай (base case) — тест-кейс для базовой ожидаемой функциональности системы или внутреннего значения в классе эквивалентности. Например, при тестировании калькулятора базовым случаем может быть сложение пользователем 2 и 2.

Базовый тест (baseline test) — вид нагрузочного теста, при котором обрабатывается минимальное количество событий, а возможно, даже ни одного, чтобы обеспечить "базовый уровень", показывающий, как выглядит минимальная нагрузка на систему.

Бактерия (bacteria) — вид вредоносного ПО, которое потребляет избыточное количество системных ресурсов, возможно, занимая все файловые дескрипторы или дисковое пространство.

Бета-тестирование (beta testing) — "реальное" тестирование подмножеством пользовательской базы до выпуска системы. Часто ему предшествует альфа-тестирование.

Биномиальный коэффициент (binomial coefficient) — количество r комбинаций данного набора размера n. Он определяется формулой $C(n, r) = n!/(r! \cdot (n - r)!)$.

Блокер (blocker) — дефект высшей степени серьезности, при котором система не может быть выпущена без исправления или использования обходного решения.

Браунфилд-разработка (brownfield development) — написание программного обеспечения, которое должно взаимодействовать с уже существующим в производстве ПО, что ограничивает потенциальный дизайн, решения, архитектуру и т. п.

Валидация (validation) — обеспечение соответствия системы потребностям клиента. Проверка того, что вы построили правильную *систему*.

Верификация (в юнит-тестировании) (verification (in unit testing)) — проверка того, что конкретный метод был вызван для **мок-объекта** (mock object).

Верификация (категория тестирования) (verification (category of testing)) — обеспечение того, что система работает правильно, в ней не возникают сбои, она дает правильные ответы и т. д. Проверка того, что вы построили систему *правильно*.

Взломщик (cracker) — неавторизованное лицо, пытающееся получить доступ к системе либо данным и/или изменить их, используя злоумышленные методы.

Вирус (virus) — разновидность **вредоносного ПО** (malware), часто небольшого размера, которое воспроизводится при вмешательстве человека. Этим вмешательством может быть что-то вроде нажатия на ссылку или запуска программы, отправленной вам в виде вложения.

Влияние (impact) — используется при сообщении о дефектах, т. е. как дефект повлияет на пользователей системы.

Внедрение зависимости (dependency injection) — передача зависимостей метода в качестве параметров вместо того, чтобы они были жестко прописаны в коде.

Это помогает при тестировании, т. к. их можно легко заменить тестовыми двойниками или фейками.

Внешне непротиворечивые (требования) (externally consistent (requirements)) — свойство соответствия системы требованиям других систем или Вселенной. Например, наличие требования к системе, чтобы она могла мгновенно связываться с базой на Плутоне, потребовало бы связи со скоростью, превышающей скорость света, и, таким образом, было бы несовместимо с законами этой вселенной.

Внешнее качество (external quality) — качество системы с точки зрения пользователя, т. е. соответствует ли она требованиям, работает ли должным образом, генерирует ли правильные результаты и т. д.

Внутренне непротиворечивые (требования) (internally consistent (requirements)) — свойство отсутствия требований, противоречащих друг другу. Например, спецификация требований, содержащая требование, гласящее: "Система всегда должна оставлять красный свет включенным", и другое требование, которое гласит, что "Система должна выключать красный свет, если включен переключатель SWITCH1", не будет внутренне непротиворечивой.

Внутреннее значение (interior value) — значение, которое не является граничным значением в своем классе эквивалентности.

Внутреннее качество (internal quality) — качество кодовой базы с точки зрения разработчиков, т. е. читаемость, понятность, поддерживаемость, расширяемость и т. д.

Вредоносное ПО (malware) — программное обеспечение, которое оказывает вредное и преднамеренное воздействие на пользователя программного обеспечения (например, компьютерный вирус или кейлоггер).

Время настенных часов (wall clock time) — еще один термин для обозначения **реального времени** (real time).

Время отклика (response time) — при **тестировании производительности** (performance testing) показывает, насколько быстро отвечает система после ввода данных пользователем или другого события.

Входное значение (input value) — конкретное значение, которое будет передано в тест-кейс. Различие между ним и предусловием может быть нечетким при ручном или другом тестировании методом черного ящика; при тестировании методом белого ящика значения, которые вы передаете тестируемому методу (например, в качестве аргументов или параметров), являются входными значениями.

Выбирать (choose) — при использовании в комбинаторике способ выражения биномиального коэффициента набора размера n и r комбинаций размера r. Например, "5 выбирает 3" означает количество возможных 3 комбинаций в наборе из 5 элементов.

Выполнимо (feasible) — термин используется в связке с требованиями; возможность протестировать с реалистичными временными рамками и распределением ресурсов.

Выходное значение (output value) — конкретное значение, которое будет получено после завершения тест-кейса. Различие между ним и постусловием при ручном или другом тестировании методом черного ящика может быть нечетким; при тестировании методом белого ящика значения, которые напрямую возвращаются из метода, являются выходными значениями.

Граничное значение (boundary value) — значение, которое находится "на границе" между классами эквивалентности. Например, система, которая имеет два класса эквивалентности — от 0 до 19 и 20 или выше, будет иметь граничные значения 19 и 20.

Граничный случай (edge case) — тест-кейс функциональности системы, которая может произойти, но будет редкой и может потребовать специальной работы для надлежащей обработки с точки зрения программирования. Например, при тестировании калькулятора граничный случай может потребовать гарантирования, что попытка деления на ноль дает правильное сообщение об ошибке.

Гринфилд-разработка (Greenfield development) — написание программного обеспечения с нуля и, таким образом, способность проектировать всю систему, не беспокоясь о предыдущих решениях.

Девятки (nines) — *см. п* девяток.

Детерминированный (deterministic) — нечто, для чего причинно-следственные связи полностью известны и воспроизводимы.

Дефект (defect) — недостаток в системе, который становится причиной ее неожиданного или некорректного поведения или несоответствия требованиям к системе. Большая часть тестирования программного обеспечения связана с обнаружением дефектов в системе.

Динамическое тестирование (dynamic testing) — тестирование системы путем ее выполнения. Примерами могут быть юнит-тестирование или тестирование методом черного ящика.

Доклад (отчет) (report) — акт информирования о дефекте по согласованной в проекте схеме.

Доступность (availability) — при тестировании производительности показывает, какой процент времени система доступна пользователю (не находится в режиме отказа, отвечает на запросы и т. п.). При тестировании безопасности один из элементов триады InfoSec, атрибут, который относится к возможности авторизованных пользователей получать доступ к системе.

Дымовой тест (smoke test) — небольшой набор тестов, который используется в качестве шлюза для дальнейшего тестирования.

Заблокировано (blocked) — статус тест-кейса в тестовом прогоне. Это означает, что в настоящее время тест-кейс не может быть выполнен по причинам, не зави-

сящим от тестировщика. Например, тестируемую функциональность еще нельзя проверить потому, что, возможно, она еще не разработана.

Завершенность (требования) (complete (requirements)) — свойство требований, определяющих целостность системы.

Заглушка (стаб) (stub) — "фейковый метод", который можно использовать в юнит-тестировании для ограничения зависимостей от других методов и сосредоточения на тестируемом методе.

Заинтересованное лицо (stakeholder) — любой человек, который напрямую заинтересован в успешном завершении, выполнении или выпуске системы — например, клиенты, разработчики и менеджеры проектов.

Запущен (running) — статус тест-кейса в тестовом прогоне. Указывает на то, что тест в настоящее время выполняется, но еще не завершен.

Злая обезьяна (evil monkey) — метод стохастического тестирования, при котором вредоносный код или данные отправляются в систему. Имитируется попытка злоумышленника получить доступ или нанести ущерб системе.

Значительный (major) — дефект третьего по значимости уровня серьезности, который указывает на серьезную проблему, но все же позволяет пользователю использовать систему.

Зомби (zombie) — компьютер с установленным программным обеспечением, которое позволяет неавторизованным пользователям получать к нему доступ для выполнения несанкционированных функций. Например, в систему может быть встроена почтовая программа, которая позволит другим пользователям отправлять спам с вашего компьютера таким образом, что истинных отправителей нельзя будет отследить.

Идемпотентность (idempotent) — качество функции или запроса, при котором один и тот же результат будет возвращен независимо от того, сколько раз функция (запрос) вызывается. Например, умножение числа на единицу является идемпотентной операцией ($5 \cdot 1 = 5$, $5 \cdot 1 \cdot 1 = 5$, $5 \cdot 1 \cdot 1 \cdot 1 = 5$ и т. д.). Но сложение единицы с числом является **неидемпотентной** (non-idempotent) операцией ($5 + 1 = 6$, $5 + 1 + 1 = 7$, $5 + 1 + 1 + 1 = 8$ и т. д.).

Идентификатор (identifier) — число или строка, которые однозначно идентифицируют тест-кейс, дефект или что-либо еще.

Импровизационное тестирование (unscripted testing) — тестирование без жесткого сценария, при котором тестировщик имеет широкие возможности использования своих собственных желаний и знаний для определения качества системы или обнаружения дефектов. Противоположно **сценарному тестированию** (scripted testing).

Инструменты DoS (DoS tools) — инструменты, которые позволяют осуществить атаку отказа в обслуживании.

Интеграция (integration) — соединение нескольких систем или подсистем для совместной работы.

Интуитивное тестирование ("Seat of your pants" testing) — тестирование, при котором ожидаемое поведение известно тестировщику из опыта работы с программным обеспечением или в данной области, а не формально.

Информационная безопасность (information security) — область обеспечения того, что компьютерные системы успешно соблюдают все три аспекта **триады InfoSec** (InfoSec triad).

Инъекционная атака (injection attack) — разновидность атаки, при которой злоумышленник пытается заставить компьютер жертвы выполнить произвольный код.

Исследовательское тестирование (exploratory testing) — неформальный стиль тестирования, цель которого часто состоит в том, чтобы узнать систему путем ее тестирования, а также найти дефекты.

Исчерпывающее тестирование (exhaustive testing) — проверка каждого возможного входного значения, состояния среды и других соответствующих факторов для данной тестируемой системы. Например, при тестировании Java-метода `int max(int a, int b)`, который возвращает большее из двух значений `a` и `b`, гарантирование, что он работает для каждой отдельной комбинации значений `a` (`MININT..MAXINT`) и `b` (`MININT..MAXINT`), даст 18 446 744 073 709 551 616 тестовых случаев[1]. Исчерпывающее тестирование на практике зачастую невозможно.

Качественный (qualitative) — характеристика, которая не может быть выражена численно, например "система должна быть *очень* крутой". Противоположность **количественному** (quantitative).

Кейлоггер (keylogger) — программное обеспечение, которое сохраняет все клавиши, которые были нажаты пользователем, обычно для передачи или извлечения злоумышленником.

Класс эквивалентности (equivalence class) — группа входных значений, которые обеспечивают одинаковый или подобный тип выходных данных.

Ключевой показатель производительности (key performance indicator) — показатель производительности, который считается важнейшим при разработке системы.

Количественный (quantitative) — характеристика, которая может быть выражена численно, например "система должна ответить в течение 500 миллисекунд". Противоположность **качественному** (qualitative).

Комбинаторное тестирование (combinatorial testing) — тестирование таким образом, чтобы гарантировать, что различные комбинации переменных будут работать должным образом.

Комбинация (combination) — подборка элементов из набора, порядок которых не имеет значения.

---

[1] 2^{64} — наибольшее значение unsigned long long. — *Прим. пер.*

Конфиденциальность (confidentiality) — атрибут системы, при котором только авторизованные пользователи могут читать данные. Элемент триады InfoSec.

Кормление собак (dogfooding) — использование собственного программного обеспечения при его разработке, например запуск операционной системы, которую вы разрабатываете, на вашем собственном компьютере.

Кривая ванны (bathtub curve) — обобщенная функция частоты отказов, значения которой вначале большие (поскольку неисправные компоненты выходят из строя вскоре после внедрения), остаются низкими на протяжении большей части срока службы системы, а затем начинают увеличиваться, когда система приближается к концу срока службы. Названа так потому, что ее график выглядит как очертание ванны при виде сбоку.

Критический (critical) — дефект второго по значимости уровня серьезности, который значительно влияет на то, как пользователь может использовать систему.

Ловушка (trapdoor) — программа или часть программы, которая обеспечивает секретный доступ к системе или приложению.

Логическая бомба (logic bomb) — код в программе, который выполняет несанкционированную функцию, например удаляет все данные в первый день месяца.

Логическая ошибка (logic error) — ошибка в программе из-за использования неправильной логики.

Матрица трассируемости (матрица соответствия требований) (traceability matrix) — двумерная матрица, отображающая тест-кейсы и требования и указывающая, какие тест-кейсы проверяют какие требования.

Модификация (modification) — в **тестировании безопасности** (security testing) — атака на **целостность** (integrity), которая намеренно изменяет данные, например атака, позволяющая пользователю произвольно изменять баланс своего банковского счета.

Мок (mock) — особый вид **тестового двойника** (test double), который отслеживает, какие методы были вызваны.

Мутационное тестирование (mutation testing) — средство "тестирования тестов" путем **посева** (seeding) тестируемой системы дефектами в виде случайного изменения кода.

Наблюдаемое поведение (observed behavior) — то, что система на самом деле делает при определенных обстоятельствах. Например, если я ввожу "2 + 2 =" на калькуляторе, **ожидаемое поведение** (expected behavior) может заключаться в том, что я увижу "4", но если вместо этого я вижу "WALLA WALLA", то "WALLA WALLA" является наблюдаемым поведением.

Набор (set) — группа уникальных элементов. Например, [1, 2, 3] — это набор, а [1, 1, 2, 3] — нет, потому что существует несколько экземпляров элемента 1.

Нагрузочное тестирование (load testing) — запуск всей системы с заданным объемом запросов (например, определенное количество пользователей или событий) для определения, как система работает в реальных условиях.

Не определено (undefined) — в том, что касается спецификации системы, — любая область, в которой поведение не указано. Например, если вся моя системная спецификация выглядит так: "Первый индикатор должен загореться, если входное значение меньше 4", то поведение системы, если входное значение равно 7, не определено.

Недетерминированный (non-deterministic) — сбой теста, который происходит не каждый раз, а только при определенных прогонах и по неизвестным причинам.

Недружественная к тестированию конструкция (test-unfriendly construct) — часть структуры кода, которую трудно протестировать, например конструктор, финализатор или закрытый метод.

Недружественная к тестированию функция (test-unfriendly function) — функциональный аспект программы, который трудно протестировать, — например, обмен данными по сети или запись на диск.

Незначительный (minor) — дефект более низкого уровня серьезности, чем **обычный** (normal), и он является причиной лишь очень небольшой проблемы при использовании системы.

Неидемпотентность (non-idempotent) — качество функции или запроса, при котором один и тот же результат может или не может быть возвращен в зависимости от того, сколько раз эта функция вызывается. Например, умножение числа на единицу является **идемпотентной** (idempotent) операцией ($5 \cdot 1 = 5$, $5 \cdot 1 \cdot 1 = 5$, $5 \cdot 1 \cdot 1 \cdot 1 = 5$ и т. д.). Но добавление единицы к числу является неидемпотентной операцией ($5 + 1 = 6$, $5 + 1 + 1 = 7$, $5 + 1 + 1 + 1 = 8$ и т. д.).

Неизменяемый (invariant) — в **тестировании на основе свойств** (property-based testing) — свойство, которое всегда должно сохраняться для функции или метода. Например, метод сортировки, который принимает несортированный массив, всегда должен возвращать массив с тем же количеством элементов, что и в исходном несортированном массиве.

Неудавшийся (failed) — статус тест-кейса в тестовом прогоне. Указывает на то, что хотя сам тест был выполнен без ошибок, оказалось не выполнено хотя бы одно постусловие или не было получено ожидаемое выходное значение, либо имело место другое неожиданное поведение. Другими словами, наблюдаемое поведение не было ожидаемым.

Неуспешный случай (failure case) — разновидность теста, при котором ожидаемое поведение системы — отказ определенным образом. Например, можно ожидать, что отправка отрицательного числа в функцию извлечения квадратного корня, которая не поддерживает комплексные числа, вызовет исключение. Также существует **успешный случай** (success case).

Нефункциональное требование (non-functional requirement) — требование, определяющее, как система должна работать, без указания конкретного поведения. Например, "система должна быть расширяемой и позволяющей добавлять плагины" или "система должна использоваться персоналом, прошедшим менее од-

ного часа обучения". Является противоположным **функциональным требованиям** (functional requirement).

Нечеткое тестирование (fuzz testing) — форма **стохастического тестирования** (stochastic testing), при котором случайные, но возможные данные передаются в систему, чтобы увидеть, как она реагирует.

Нечистый (impure) — противоположность **чистому** (pure) — метод или функция, которые вызывают по крайней мере один **побочный эффект** (side effect).

Неявное граничное значение (implicit boundary value) — **граничное значение** (boundary value), которое явно не вызывается требованиями системы, но может влиять на работу системы. Например, код, написанный с использованием 32-битных целых чисел со знаком (таких как `int` в Java), имеет неявную границу в 2 147 483 647, максимальный размер целого числа. Добавление единицы к этому значению приведет к ошибке переполнения.

Нормальный (normal) — дефект той степени серьезности, которая заметна, но не сильно мешает пользователю использовать систему.

Обезьянье тестирование (monkey testing) — еще один термин для **стохастического тестирования** (stochastic testing).

Обеспечение качества (quality assurance) — гарантирование качества программного обеспечения различными методами. Тестирование программного обеспечения — важная, но не единственная часть обеспечения качества.

Общее время (total time) — общее количество времени, в течение которого выполняется код (пользовательский или ядра), без учета других факторов (таких как время, затраченное на ожидание ввода). *См.* также реальное время.

Однозначность (требования) (unambiguous (requirements)) — свойство требований, которые можно прочитать и понять одним и только одним способом.

Ожидаемое поведение (expected behavior) — то, что система должна делать при определенных обстоятельствах. Например, после ввода "2 + 2 =" в калькуляторе ожидается, что система отобразит на экране "4".

Остановлен (paused) — статус тест-кейса в тестовом прогоне. Указывает на то, что тестировщик начал выполнение теста, но он временно приостановил его по сторонним причинам (например, тестировщик ушел на обед).

Отказ в обслуживании (denial of service) — метод атаки на доступность путем отправки такого количества неавторизованных пакетов или других событий на вычислительный ресурс, что ни один авторизованный пользователь не сможет получить к нему доступа.

Отрицательный тест-кейс (negative test case) — *см.* неуспешный случай.

Очистка (sanitization) — "очистка" пользовательского ввода, чтобы он не содержал кода, который мог бы выполняться запущенной программой, или другого содержимого, которое могло бы нанести вред системе.

Ошибка (error) — статус тест-кейса в тестовом прогоне. Указывает на то, что существует проблема с самим тест-кейсом и тест не может быть запущен. Например, в предусловиях указано невозможное состояние.

Ошибка ввода-вывода диска (disk I/O error) — ошибка, возникающая из-за ошибки чтения или записи в долговременное локальное хранилище (обычно, но не всегда, диск).

Ошибка внедрения (инъекции) (injection error) — ошибка, при которой система случайно разрешает выполнение произвольного кода. Оставляет систему уязвимой для **инъекционной атаки** (injection attack).

Ошибка доступности (accessibility error) — ошибка, возникающая из-за того, что пользователь применяет нестандартное устройство ввода или вывода, когда система не может использоваться теми, кто не имеет доступа к системе с помощью "стандартных" устройств.

Ошибка интеграции (integration error) — тип ошибки, возникающий из-за несовместимости или других проблем на границе между различными системами или подсистемами.

Ошибка интерфейса (interface error) — ошибка, из-за которой интерфейс системы определен неправильно или система, обращающаяся к нему, осуществляет это неправильно.

Ошибка конфигурации (configuration error) — ошибка, возникающая в результате неправильной конфигурации системы, отличается от ошибки в коде, из которого состоит сама система.

Ошибка на единицу (off-by-one error) — особый вид **логической ошибки** (logic error), при которой программа делает что-то неправильно, потому что некое значение больше или меньше на единицу.

Ошибка неверных данных (bad data error) — ошибка, возникающая в результате получения системой искаженных, поврежденных или недействительных данных.

Ошибка нулевого указателя (null pointer error) — ошибка, возникающая в результате попытки разыменовать в коде нулевой указатель или получить доступ к нулевому объекту.

Ошибка округления (rounding error) — ошибка в программе, вызванная округлением числа системой.

Ошибка отображения (display error) — ошибка, при которой было вычислено правильное значение, но оно не было отображено правильно.

Ошибка отсутствия данных (missing data error) — ошибка, возникающая из-за того, что система не получает необходимые данные.

Ошибка предположения (error of assumption) — ошибка, возникающая в результате неправильного предположения разработчика или другого человека о том, как должна работать система.

Ошибка распределенной системы (distributed system error) — ошибка, возникающая как следствие того, что система распределена, а не работает полностью на одном компьютере.

Ошибка регрессии (regression failure) — сбой ранее работавшей функциональности, вызванный (по-видимому) несвязанной дополнительной функциональностью или исправлениями дефектов.

Ошибка с плавающей запятой (floating-point error) — ошибка, вызванная округлением или неточностью числа с плавающей запятой из-за неполного сопоставления фактических десятичных чисел и значений с плавающей запятой.

Парное программирование (pair programming) — два человека, одновременно работающие над задачей на одном компьютере. Могут быть тестировщиком белого ящика и разработчиком, вместе изучающими код.

Пассивная атака (passive attack) — при **тестировании безопасности** (security testing) — атака на систему, не вызывающая изменений в системе, например перехват сетевого трафика.

Патологический случай (pathological case) — другой термин для **углового случая** (corner case).

Переполнение буфера (buffer overrun) — уязвимость, при которой можно записать больше данных, чем было разрешено. Это может вызвать сбои системы или несанкционированный доступ.

Перестановка (permutation) — расположение **набора** (set), в котором порядок имеет значение. Например, возможные перестановки набора [1, 2] — это [1, 2] и [2, 1]. Возможные перестановки набора [1, 2, 3]: [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2] и [3, 2, 1].

Перехват (interception) — при **тестировании безопасности** (security testing) — атака на **конфиденциальность** (confidentialit), например перехват данных в сети с помощью **анализатора пакетов** (packet analyzer) или на компьютере с помощью **кейлоггера** (keylogger).

Периодический сбой (intermittent failure) — сбой теста, который происходит не все время, а только при определенных запусках. Часто причина периодического сбоя неизвестна или уже могла быть устранена.

Побочный эффект (side effect) — все, что не является строго возвращаемым результатом вычисления. Например, отображение сообщения или установка значения переменной.

Позитивный тест-кейс (positive test case) — *см.* успешный случай.

Показатель производительности (performance indicator) — количественный показатель, указывающий на уровень производительности системы. Например, время отклика или использование памяти.

Показатель, ориентированный на сервис (service-oriented indicator), — показатель производительности, связанный с тем, как пользователь будет взаимодействовать с системой.

Показатель, ориентированный на эффективность (efficiency-oriented indicator), — показатель производительности, связанный с эффективностью использования доступных системе вычислительных ресурсов.

Покрывающий массив (covering array) — массив, охватывающий все возможные комбинации значений переменных.

Покрытие ветвей (branch coverage) — какой процент ветвей в коде тестируется, обычно юнит-тестами.

Покрытие кода (code coverage) — часть кодовой базы, которая реально тестируется обычно с помощью юнит-тестов. Хотя существуют разные виды покрытия кода, чаще всего при использовании этого термина неспециалистами подразумевается покрытие операторов.

Покрытие операторов (statement coverage) — показывает, какой процент операторов в коде тестируется, обычно с помощью юнит-тестов.

Полевое тестирование (field testing) — проверка того, что система работает, когда с ней взаимодействуют реальные пользователи.

Пользовательское время (user time) — количество времени, за которое выполняется пользовательский код, пока система выполняет задачу.

Пользовательское приемочное тестирование (user acceptance testing) — особый вид **приемочного тестирования** (acceptance testing), при котором тестировщик является пользователем программного обеспечения. Это гарантирует, что тестируемая система приемлема для пользователя, удовлетворяя его потребности.

Пользовательское тестирование (user testing) — фактический пользователь системы пытается выполнить задачи, часто без инструкций, чтобы определить, как пользователи взаимодействуют с системой.

Попарное тестирование (pairwise testing) — особая форма комбинаторного тестирования, при котором вы проверяете все двусторонние взаимодействия.

Порог (threshold) — в **тестировании производительности** (performance testing) еще один термин для обозначения **порога производительности** (performance threshold).

Порог производительности (performance threshold) — при **тестировании производительности** (performance indicators) абсолютно минимальное значение заданного показателя производительности, при котором система считается готовой к выпуску. *См.* также цель производительности.

Посев (seeding) — преднамеренное добавление дефектов в программу для определения, найдет ли их стратегия тестирования.

Постусловие (postcondition) — все условия, которые должны быть реализованы после завершения шагов выполнения для того, чтобы тест считался пройденным. Например, постусловие для редактирования вашего имени пользователя может заключаться в том, что на странице "Информация об учетной записи" будет отображаться новое имя пользователя.

Предусловие (precondition) — все условия, которые должны выполняться до того, как начнутся шаги выполнения тест-кейса. Например, при тестировании информации об учетной записи веб-сайта предусловием может быть то, что пользователь уже вошел в систему.

Прерывание (interruption) — при **тестировании безопасности** (security testing) атака на **доступность** (availability), например **DDoS-атака** (DDoS attack) или отключение сетевого коммутатора.

Приемочное тестирование (acceptance testing) — тестирование конечным пользователем, заказчиком или другим независимым персоналом для подтверждения того, что система может быть принята к использованию.

Принцип наименьших привилегий (principle of least privilege) — принцип, согласно которому пользователи должны иметь тот минимальный доступ к системе, который необходим для выполнения их работы. Например, разработчик не должен (как правило) иметь доступ к данным о заработной плате, а персонал отдела кадров не должен иметь доступа к исходному коду.

Проверка границ (bounds checking) — проверка во время выполнения, что данные не записываются за пределы правильно выделенного массива.

Программа-вымогатель (ransomware) — разновидность **вредоносного ПО** (malware), которое выполняет нежелательное действие (например, шифрует ваш жесткий диск) и требует деньги или другую компенсацию для его отмены.

Пройден (passed) — статус тест-кейса в тестовом прогоне. Указывает на то, что тест был выполнен без ошибок и система выполнила все постусловия и/или получены ожидаемые выходные значения, — наблюдаемое поведение соответствует ожидаемому.

Производительная (performant) — программа, обеспечивающая высокий уровень производительности, который обычно измеряется соответствием KPI или его превышением.

Пропускная способность (throughput) — при **тестировании производительности** (performance testing) количество событий или задач, которые система может обработать за определенный период времени.

Профилировщик (profiler) — инструмент, который позволяет вам измерять использование ресурсов и внутренние события (такие как вызовы методов или создание экземпляров объектов) запущенной программы.

Разбиение (partitioning) — *см.* разбиение классов эквивалентности.

Разбиение классов эквивалентности (equivalence class partitioning) — разделение определенного функционала на отдельные классы эквивалентности на основе входных значений.

Разработка Test-First ("сперва тесты") (test-first development) — любая методология разработки программного обеспечения, при которой тесты пишутся до кода, который должен с ними работать.

Разработка через тестирование (test-driven development, TDD) — особая методология разработки программного обеспечения, которая включает в себя **разработку, ориентированную на тестирование ("сперва тесты")** (test-first development), наряду с другими принципами, такими как непрерывный рефакторинг и ожидание изменений.

Распределенный отказ в обслуживании (distributed denial of service) — атака отказа в обслуживании, в которой участвует множество различных источников неавторизованных пакетов с целью увеличения количества событий, которые система должна обрабатывать, а также для маскировки конечного источника. Часто сокращается как DDos.

Реальное время (real time) — фактическое количество времени (то же самое время, которое измеряется часами), необходимое процессу для выполнения некоторой задачи. Также называется **временем настенных часов** (wall clock time). Не путать с системой реального времени.

Рефакторинг (refactoring) — изменение кода без изменения его функциональности для улучшения внутреннего качества программы (например, чтобы сделать код более легким для чтения, более понятным или более удобным для сопровождения).

Рефлексия (reflection) — способ определения структуры классов и методов во время выполнения.

Решение (обходное решение) (workaround) — метод, позволяющий избежать известный **дефект** (defect), сохраняя при этом возможность использовать систему.

Рыбалка (fishing) — ловля морских животных для еды или развлечения. Не имеет ничего общего с **тестированием безопасности** (security testing). А если вам так показалось, вероятно, думаете о **фишинге** (phishing).

Сводка (отчет о дефекте) (summary (defect reporting)) — краткое описание **дефекта** (defect) при заполнении отчета о дефекте.

Серьезность (severity) — степень, в которой конкретный **дефект** (defect) вызывает беспокойство у проектировщиков системы, разработчиков и других заинтересованных сторон.

Сетевая ошибка (network error) — ошибка, которая возникает, когда сетевое подключение неоптимально или отсутствует вообще. Пример — приложение, которое зависает, если подключение к Интернету потеряно в середине транзакции.

Сеть ботов (ботнет) (bot network) — коллекция зомби, управляемая мастером.

Системное время (system time) — количество времени, в течение которого работает код ядра при выполнении системой задачи.

Собачья еда, съешь сам свою собачью еду (dogfood, eating your own) — еще один термин для обозначения кормления собак.

Согласованность (требований) (consistent (requirements)) — свойство требований, которым можно следовать без парадокса (например, "система должна отобра-

жать сообщение „Привет“ при запуске" и "система не должна отображать никаких сообщений при запуске" несовместимы).

Соглашение об уровне обслуживания (service level agreement) — соглашение поставщика услуг, которое часто включает гарантию **доступности** (availability).

Сортировка (triage) — встреча или другой способ (например, обсуждение по электронной почте) определения приоритетности дефектов.

Социальная инженерия (social engineering) — манипулирование людьми, чтобы со злым умыслом заставить их выполнять действия, которые ставят под угрозу безопасность системы. Например, злоумышленник, притворяясь сотрудником ИТ-отдела, звонит помощнику администратора и требует пароль пользователя.

Спецификация требований (requirements specification) — список требований для данной системы. Ожидается, что система будет соответствовать всем требованиям спецификации.

Среднее время восстановления (mean time to repair) — при **тестировании доступности** (availability testing) — среднее время, необходимое для устранения отказа. Часто сокращенно обозначается как MTTR (mean time to repair).

Среднее время между отказами (mean time between failures) — при **тестировании доступности** (availability testing) — среднее время наработки на отказ в системе. Часто сокращенно обозначается как MTBF (mean time between failures).

Статическое тестирование (static testing) — тестирование системы без выполнения какого-либо ее кода. Примерами могут служить инструменты анализа кода или моделирования.

Стохастическое тестирование (stochastic testing) — тестирование системы с использованием случайных входных данных. Эти входные данные не обязательно должны быть полностью случайными; например, это может быть распределение вероятностей значений или сгенерированные строки. Также называется **обезьяньим тестированием** (monkey testing).

Стресс-тест (stress test) — вид **нагрузочного теста** (load test), при котором очень большое количество событий обрабатывается за небольшой промежуток времени для определения, как система ведет себя в те периоды, когда она "испытывает стресс".

Строгое разбиение (strict partitioning) — разбиение классов эквивалентности таким образом, чтобы входные значения любого из них не перекрывались.

Сценарное тестирование (scripted testing) — тестирование с помощью жесткого сценария, такого как тест-план, где шаги четко определены и упорядочены. Противоположное к **импровизационному тестированию** (unscripted testing).

Счастливый путь (happy path) — самый простой путь, по которому пользователь может пройти при работе с системой, если система работает правильно, а пользователь не пытается выполнить что-то, что является **граничным случаем** (edge case) или **угловым случаем** (corner case).

Таблица истинности (truth table) — таблица, в которой показаны все возможные значения группы логических (истина/ложь) переменных.

Тавтологический тест-кейс (tautological test case) — тест-кейс, который написан так, что всегда будет проходить, например `boolean foo = true; assertTrue (foo);`. Обычно это происходит непреднамеренно.

Телефонный фрикер (или просто фрикер) (phone phreak) — человек, исследующий телефонную систему, обычно без разрешения соответствующих органов. Среди известных телефонных фрикеров — Джон Дрейпер (John Draper) и Джо Энгрессиа (Joe Engressia).

Тест стабильности (stability test) — вид **нагрузочного теста** (load test), при котором обрабатывается небольшое количество событий, но в течение длительного периода времени, это необходимо для определения, насколько стабильна система в нетривиальные периоды времени.

Тест фиксации (pinning test) — автоматический тест (обычно юнит-тест), который проверяет текущий ответ системы для изменения кода без изменения существующего поведения. Обратите внимание, что тесты фиксации проверяют текущий ответ, а *не* правильный. Часто используется при рефакторинге или работе с **устаревшим кодом** (legacy code).

Тестирование безопасности (security testing) — проверка того, что система соответствует критериям триады InfoSec, что система защищена от несанкционированного вмешательства и/или доступа.

Тестирование всех пар (all-pairs testing) — еще один термин для попарного тестирования.

Тестирование методом белого ящика (white-box testing) — тестирование кода напрямую и с полным знанием тестируемого кода. **Юнит-тестирование** (unit testing) — пример тестирования методом белого ящика.

Тестирование на основе свойств (property-based testing) — метод тестирования, обычно автоматизированный, при котором многие значения, часто псевдослучайно генерируемые, задаются в качестве входных данных и тестируются свойства выходных данных, а не проверяются их конкретные значения или поведение.

Тестирование на проникновение (penetration testing) — тестирование безопасности системы путем попытки взломать ее так, как это мог бы сделать неавторизованный пользователь.

Тестирование на соответствие нормативным требованиям (regulation acceptance testing) — проверка соответствия системы законодательным или другим нормативным требованиям.

Тестирование носителей (media testing) — проверка того, что носитель, на котором хранится система (например, CD-ROM или жесткий диск сервера), работает правильно и все данные находятся в нужном месте.

Тестирование производительности (performance testing) — тестирование соответствия системы указанным для нее **показателям производительности** (performance threshold).

Тестирование серого ящика (grey-box testing) — тестирование кода в качестве пользователя, но со знанием кодовой базы для понимания того, где могут скрываться ошибки. Смесь **тестирования белого ящика** (white-box testing) и **тестирования черного ящика** (black-box testing).

Тестирование системы (system testing) — тестирование системы в целом так, как пользователь (в отличие от разработчика) будет с ней взаимодействовать. Обычно осуществляется в виде черного или серого ящика.

Тестирование черного ящика (black-box testing) — тестирование кода так, как это сделал бы пользователь без знания кодовой базы. Большинство ручных тестов — это тесты черного ящика.

Тестируемая система (system under test) — система, которая тестируется фактически.

Тестируемое приложение (application under test) — система, которую тестирует тестировщик.

Тестируемость (testability) — качество системы, которое определяет, насколько легко ее тестировать, этому способствует наличие хорошо спроектированных и согласованных методов, использование, где это возможно, чистых функций, разрешение внедрения зависимостей и т. д.

Тестируемый код (testable code) — код, который можно легко протестировать в автоматическом режиме на нескольких уровнях абстракции.

Тест-кейс (test case) — наименьшая единица тест-плана, отдельный тест, который описывает, что нужно тестировать, и проверку ожидаемого поведения системы.

Тестовая фикстура (test fixture) — процедура или программа, которая переводит систему в состояние готовности к тестированию.

Тестовое покрытие (test coverage) — общий термин обозначения того, как аспекты или части системы охватываются тестами.

Тестовый артефакт (test artifact) — документ или другой побочный продукт процесса тестирования, например тест-планы или результаты тестирования.

Тестовый двойник (test double) — "фейковый объект", используемый в юнит-тестировании для ограничения зависимостей от других классов, которые могут иметь собственные проблемы, быть еще не разработаны или просто требуют время или ресурсы для фактического создания экземпляра.

Тестовый набор (test suite) — группа связанных тест-планов.

Тестовый прогон (test run) — фактическая итерация (прогон) тест-плана.

Тест-план (test plan) — список связанных тест-кейсов, которые выполняются вместе.

Тест-раннер (test runner) — программа, которая автоматически выполняет набор тестов.

Тест-хук (test hook) — "скрытый" метод, который позволяет вводить или выводить данные из системы для упрощения тестирования.

Требование "-сть" (-ility requirement) — еще один термин для **нефункционального требования** (non-functional requirement), названный так потому, что многие из этих требований используют слова, оканчивающиеся на "-сть", для их описания (например, масштабируемость, обслуживаемость).

Требование (requirement) — изложение того, что должна выполнять разрабатываемая система, чтобы ее можно было считать полной и правильной.

Триада CIA (CIA triad) — еще один термин для триады InfoSec.

Триада InfoSec (InfoSec triad) — три критерия, указывающие на безопасность системы: **конфиденциальность** (confidentiality), **целостность** (integrity) и **доступность** (availability).

Тривиальный (trivial) — дефект, степень серьезности которого такова, что он незаметен или малозаметен, и вызывает лишь самые незначительные проблемы для пользователя системы.

Троянский конь (trojan horse) — разновидность **вредоносного ПО** (malware), которое выдает себя за другой вид программы, чтобы обманом заставить пользователей установить и запустить ее.

Тупая обезьяна (dumb monkey) — метод стохастического тестирования, при котором в систему отправляются случайные данные.

Угловой случай (corner case) — тест-кейс того функционала системы, который маловероятно произойдет или выходит за пределы области, в которой пользователь, вероятно, сможет воспроизвести его. Назван по аналогии с граничным случаем (угол — это место пересечения нескольких границ).

Улучшение (enhancement) — запрошенная модификация или дополнительный функционал, которые изначально не были указаны в требованиях.

Умная обезьяна (smart monkey) — метод **стохастического тестирования** (stochastic testing), при котором передаваемые данные имитируют то, как реальный пользователь будет использовать систему. Например, тест умной обезьяны для системы обработки текста может напечатать несколько букв, отформатировать их и сохранить в файл (как это может сделать реальный пользователь) вместо того, чтобы нажимать кнопки наугад.

Успешный случай (success case) — вид тест-кейса, в котором ожидаемое поведение системы состоит в том, чтобы вернуть правильный результат или выполнить правильное действие. *См.* также неуспешный случай.

Устаревший код (legacy code) — код, который выполняется в производственной среде и был написан без использования современных методов разработки программного обеспечения и/или имеет некачественное автоматизированное покрытие для тестирования.

Утверждение (assertion) — в юнит-тесте утверждение, в котором устанавливается, что должно выполняться определенное условие. Если условие не выполняется, тест считается неудачным. Например, `assertEquals(2, Math.sqrt (4));`.

Утилизация (utilization) — в **тестировании производительности** (performance testing) относительный или абсолютный объем определенного вычислительного ресурса (например, ОЗУ, инструкций процессора, дискового пространства), который используется при определенных обстоятельствах.

Уязвимость (vulnerability) — потенциальный дефект, который позволяет пользователю скомпрометировать систему или неким образом получить несанкционированный доступ к ней.

Фабрикация (fabrication) — при **тестировании безопасности** (security testing) атака на **целостность** (integrity), при которой намеренно добавляются данные, например атака, позволяющая пользователю создать совершенно новый банковский счет.

Факториал (factorial) — математическая функция, которая вычисляет результат целого числа n, умноженного на $n-1$, $n-2$ и т. д., вплоть до 1. Например, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.

Фальсификация инварианта (falsifying the invariant) — демонстрация примера, в котором **инвариант** (invariant) не содержится. Например, инвариант для арифметического метода, в котором складываются два положительных целых числа, заключается в том, что сумма всегда будет превышать одно из чисел. Фальсификация инварианта показывает, что $1 + 1 = 0$.

Фейк (fake) — разновидность **тестового двойника** (test double), в котором поведение двойника обрабатывается самим объектом, а не методами-заглушками. Фейки действуют как более простые и быстрые версии реального объекта с целью более быстрого выполнения тестов или уменьшения зависимостей.

Фишинг (phishing) — распространенная **атака** (attack), направленная на получение личной или другой конфиденциальной информации по электронной почте или другим средствам связи.

Форк-бомба (fork bomb) — особый вид **бактерий** (bacteria), которые постоянно разветвляются, в результате чего все ресурсы процессора расходуются на создание новых копий форк-бомбы.

Функциональное требование (functional requirement) — требование, которое точно определяет, что система должна делать при определенных обстоятельствах. Например, "система должна отображать ERROR в консоли, если какой-либо параметр отрицательный". Также существует **нефункциональное требование** (non-functional requirement).

Хакер (hacker) — согласно Jargon File[1] "человек, которому нравится изучать особенности программируемых систем, и то, как растянуть их возможности". В наше время часто используется для обозначения **взломщика** (cracker).

---

[1] Сетевой энциклопедический словарь хакерского сленга. — *Прим. пер.*

Хаотическая обезьяна (chaos monkey) — инструмент стохастического тестирования, разработанный Netflix, который тестирует распределенные системы, отключая случайные серверы в системе.

Хрупкий (fragile) — описание тест-кейса или набора тестов, которые легко сломать небольшими изменениями в кодовой базе или среде.

Целевой фишинг (spear phishing) — особый вид **фишинга** (phishing), который нацелен на конкретных лиц и предназначен только для них. Например, обычное фишинговое письмо может выглядеть так: "Уважаемый пользователь, пожалуйста, сбросьте свой пароль электронной почты, нажав на эту ссылку", тогда как целевое фишинговое письмо будет выглядеть так: "Уважаемый мистер Джонс, пожалуйста, сбросьте свой пароль вашей почты SuperDuperEmail, нажав на эту ссылку. С уважением, Джейн Смит, вице-президент по безопасности, SuperDuperCo".

Целостность (integrity) — атрибут системы, показывающий, что только авторизованные пользователи могут записывать данные. Элемент **триады InfoSec** (InfoSec triad).

Цель (target) — в **тестировании производительности** (performance testing) — это еще один термин для обозначения **цели производительности** (performance target).

Цель производительности (performance target) — целевое значение **показателя производительности** (performance testing) при **тестировании производительности** (performance indicator). Если значение показателя соответствует цели или превышает ее, значит, система достигла цели. *См.* также порог производительности.

Червь (worm) — разновидность **вредоносного ПО** (malware), часто небольшого размера, которое воспроизводится без вмешательства человека.

Чистый (pure) — метод или функция, которые не вызывают **побочных эффектов** (side effects), а просто возвращают результат вычисления.

Шаги воспроизведения (reproduction steps) — шаги, необходимые для воспроизведения дефекта. Часто включаются в отчеты о дефектах, чтобы читающие поняли, что вызывает дефект и как его воспроизвести.

Шаги выполнения (execution steps) — фактические шаги, которые выполнит тест после проверки выполнения всех предусловий.

Шов (seam) — место в кодовой базе, где вы можете изменять поведение, не изменяя сам код.

Шпионское ПО (spyware) — разновидность **вредоносного ПО** (malware), которое тайно отслеживает действия пользователя системы.

Эксплойт (exploit) — программа или фрагмент данных, использующие уязвимость. Уязвимость является "строго теоретической" до тех пор, пока кто-нибудь не разработает способ ее использования.

Эксплуатационное тестирование (operational testing) — тестирование, что система работает в реальных условиях.

Юнит-тестирование (unit testing) — тестирование наименьших отдельных единиц кода, таких как методы или функции, методом белого ящика.

Явное граничное значение (explicit boundary value) — **граничное значение**, явно вызываемое системными требованиями. Например, требования к автоматическому термометру могут указывать на то, что система включит световой индикатор ОПАСНО, когда зарегистрированная температура составляет 102 градуса или выше. Явные граничные значения будут 101 и 102. Являются противоположными **неявным граничным значениям**.

Глава 27

Благодарности

Спасибо всем, кто помог воплотить эту книгу в жизнь или сделать ее лучше, особенно Россу Ачесону (Ross Acheson), Натаниэлю Блейку (Nathaniel Blake), Уиллу Энглеру (Will Engler), Джейку Голдингу (Jake Goulding), Брэндону Хангу (Brandon Hang), Джоэлу МакКракену (Joel McCracken), Робби Маккинстри (Robbie McKinstry), Райану Рахуба (Ryan Rahuba), Стиву Роббибаро (Steve Robbibaro), Нику Треу (Nick Treu), Эду Вьянко (Ed Wiancko), Шеридан Зиванович (Sheridan Zivanovich).

Особая благодарность:

Патрику Кину (Patrick Keane), который проделал чрезвычайно тщательную работу по редактированию и предоставил ценный взгляд на область QA в целом;

Кэрол Николс (Carol Nichols), которая не только отлично справлялась с поиском дефектов в моем коде, но и постоянно забавляла меня своими сообщениями о правках;

Тиму Паренти (Tim Parenti), который обнаружил больше опечаток и ошибок, чем я когда-либо мог предположить, и даже больше — проделал отличную работу, исправив многие из них.

Предметный указатель

A

Availability 263
Avoid Slow Running Tests 158

B

Base case 31
Bathtub curve 213
Boundary values 28

C

Corner case 31

D

Denial of service 224
Dependency injection 174, 261
Dogfooding 103
Don’t Repeat Youself (DRY) 172
Dynamic testing 263

E

Edge case 31

F

Fake 139
Fake It ‘Til You Make It 158
Fork bomb 225

I

Input value 262
Interior values 28

J

JUnit 121

K

Keep It Simple, Stupid (KISS) 158
Key performance indicators (KPI) 205

L

Legacy code 176

M

Malware 225, 262
Mean time between failures (MTBF) 212
Mean time to repair (MTTR) 212
Mock 137
Monkey testing 191

P

Pinning test 176
Property-based testing 192

R

Real time 207
Remember That These Are Principles, Not Laws 158
Response time 262

S

Seam 177
Service level agreement (SLA) 210

Smoke test 99
Soak test 214
SQL-инъекция 226
Stability test 214
Stub 135
System time 207

T

Test doubles 132
Test hooks 171
Test-driven development (TDD) 153
Test-unfriendly constructs (TUC) 175
Test-unfriendly features (TUF) 175
Total time 208

U

User time 207

W

Wall clock time 262

Y

You Ain't Gonna Need It (YAGNI) 158, 260

А

Альфа-тестирование 102
Анализатор пакетов 218
Атака:
◊ активная 224
◊ инъекция 226
◊ модификации 224
◊ отказа в обслуживании 224
◊ пассивная 224
◊ перехвата 224
◊ прерывания 224
◊ фабрикации 224

Б

Бактерия 225
Бета-тестирование 102
Блокер 93
Бот-сеть 226

В

Валидация 16
Верификация 16
Вирус 225
◊ вымогатель 226
Влияние 92
Внедрение зависимости 174, 261
Время:
◊ восстановления среднее 212
◊ между отказами среднее 212
◊ настенных часов 207, 262
◊ общее 208
◊ отклика 262
◊ пользовательское 207
◊ реальное 207
◊ системное 207

Д

Двойник тестовый 132
Дефект 85
◊ значительный 93
◊ критический 93
◊ незначительный 93
◊ обычный 93
◊ тривиальный 93
Доступность 223, 263

Ж

Жизненный цикл дефекта 87

З

Заглушка 135
Заинтересованное лицо 234
Значение:
◊ внутреннее 28
◊ входное 48, 262
◊ выходное 51
◊ граничное 28, 30
Зомби 225

И

Идентификатор 45
Инвариант 193
Инструмент:
◊ DoS 226
◊ профилирования 218

К

Класс эквивалентности 27, 265
Код:
◊ тестируемый 167
◊ чужой унаследованный 176
Конструкция недружественная к тестированию 175
Конфиденциальность 223
Кормление собак 103
Кривая ванны 213

Л

Ловушка Trapdoor 225
Логическая бомба 225, 266

М

Матрица трассируемости 59
Медиатестирование 101
Мок 137

О

Обезьяна хаотическая 197
Очистка входных данных 226
Ошибка:
◊ ввода-вывода 69
◊ внедрения 68
◊ доступности 73
◊ интеграционная 65
◊ интерфейса 70
◊ конфигурации 72
◊ логическая 62
◊ на единицу 63
◊ нулевого указателя 71
◊ округления 63
◊ отображения 68
◊ отсутствующих данных 66
◊ плавающей запятой 63
◊ плохих данных 67
◊ предположений 65
◊ распределенных систем 71
◊ сетевая 69

П

Переполнение буфера 228
Поведение:
◊ наблюдаемое 92
◊ ожидаемое 92
Показатель производительности 202
◊ ключевой 205
◊ ориентированный на:
▫ сервис 203
▫ эффективность 203
Покрытие:
◊ кода 149
◊ методов 149
◊ операторов 149
Порог производительности 204
Постусловие 51
Предусловие 46, 122
Принцип наименьших привилегий 230
Прогон 55
Программа вредоносная 225
Программирование парное 24
Профилировщик 218
Путь счастливый 31, 107, 145

Р

Разработка через тестирование 153
Рефакторинг 154, 157
Рефлексия 251

С

Серьезность 93
Случай:
◊ базовый 31, 145, 261
◊ граничный 31
◊ неуспешный 32
◊ патологический 31
◊ угловой 31
◊ успешный 32, 267
Соглашение об уровне обслуживания 210
Социальная инженерия 230
Спецификация требований 37
Способность пропускная 204, 215

Т

Таблица истинности 181
Тест:
◊ базовый 214
◊ дымовой 263
◊ стабильности 214

Тест *(прод.)*:
◊ фиксирования 176
◊ хрупкий 47
Тестирование:
◊ автоматизированное 112
◊ белого ящика 33, 71, 83, 112
◊ всех пар 181
◊ динамическое 34, 263
◊ дымовое 99
◊ импровизационное 100
◊ интуитивное 18, 265
◊ исследовательское 104
◊ исчерпывающее 20, 265
◊ комбинаторное 181
◊ модульное 117
◊ мутационное 197
◊ на основе свойств 192
◊ на проникновение 232
◊ нагрузочное 214
◊ нечеткое 68
◊ обезьянье 191
▫ злое 196
▫ тупое 195
▫ умное 195
◊ полевое 102
◊ пользовательское 229
▫ приемочное 102
◊ попарное 181, 184
◊ приемочное 101
◊ производительности 202, 276
◊ ручное 109
◊ серого ящика 34, 276
◊ системы 276
◊ статическое 34
◊ стохастическое 191
◊ сценарное 100
◊ черного ящика 32, 71, 110, 276
◊ эксплуатационное 102
Тест-кейс 44
◊ негативный 32
◊ позитивный 32
Тест-план 44, 52
Тест-раннер 128
Тест-хуки 171
Требования:
◊ к программному обеспечению 36
◊ нефункциональные 42
◊ функциональные 42
Триада:
◊ CIA 223
◊ InfoSec 223
Троянский конь 225

У

Улучшение 93
Утверждение 123
◊ assertion 120
Утилизация 204, 217
Уязвимость 225

Ф

Фаззинг 68
Фальсификация инварианта 194
Фейк 139
Фикстура тестовая 54
Фишинг 230
◊ целевой 231
Форк-бомба 225
Функция:
◊ идемпотентная 194
◊ недружественная к тестированию 175
◊ неидемпотентная 194
◊ нечистая 126
◊ чистая 126

Ц

Цели производительности 204
Целостность 223
Цикл "красный — зеленый — рефакторинг" 156

Ч

Червь 225

Ш

Шаблон красно-желто-зеленый 239
Шаг выполнения 49
Шов 177
Шпионское программное обеспечение 226

Э

Эксплойт 225
Эффект побочный 126

Ю

Юнит-тестирование 117