

Искусство

ЮНИТ ТЕСТИРОВАНИЯ

С ПРИМЕРАМИ
НА JAVASCRIPT

ТРЕТЬЕ ИЗДАНИЕ



MANNING

Рой ОШЕРОВ
ВЛАДИМИР ХОРИКОВ

The Art of Unit Testing

THIRD EDITION
WITH EXAMPLES IN JAVASCRIPT

ROY OSHEROVE
WITH VLADIMIR KHOLOKOV



MANNING
SHELTER ISLAND

Искусство юнит-тестирования

ТРЕТЬЕ ИЗДАНИЕ
С ПРИМЕРАМИ НА JAVASCRIPT

Рой ОШЕРОВ
ВЛАДИМИР ХОРИКОВ

Выпущено
при поддержке

КРОК

 ПИТЕР®

Санкт-Петербург · Москва · Минск

2025

ББК 32.973.2-018-07
УДК 004.415.53
О-96

Ошеров Рой, Хориков Владимир

О-96 Искусство юнит-тестирования с примерами на JavaScript. 3-е межд. изд. — СПб.: Питер, 2025. — 320 с.: ил. — (Серия «Библиотека программиста»).
ISBN 978-5-4461-4232-3

Искусство юнит-тестирования не сводится к изучению правильного набора инструментов и практик. Искусство создавать классные тесты — это понимание сути, поиск верной стратегии для каждого конкретного случая и умение выйти из ситуации, когда тестирование превращается в беспорядочный процесс. Эта книга предлагает советы и рекомендации, которые полностью изменят ваш подход к тестированию ПО.

Вы научитесь создавать читабельные и простые в сопровождении тесты, изучите стратегии тестирования в масштабах организации, диагностику проблем, работу с унаследованным кодом и «бескомпромиссный» рефакторинг. Книга насыщена практическими примерами и знакомыми сценариями. Третье издание было дополнено методами, присущими объектно-ориентированному, функциональному и модульному стилю программирования. В примерах используются JavaScript, TypeScript и Node.js.

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

ББК 32.973.2-018-07
УДК 004.415.53

Права на издание получены по соглашению с Manning Publications. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1617297489 англ.

Authorized translation of the English edition © 2024 Manning Publications.
This translation is published and sold by permission of Manning Publications,
the owner of all rights to publish and sell the same.

ISBN 978-5-4461-4232-3

© Перевод на русский язык ООО «Прогресс книга», 2025
© Издание на русском языке, оформление ООО «Прогресс книга», 2025
© Серия «Библиотека программиста», 2025

Оглавление

| | |
|---|-----------|
| Отзывы о втором издании | 14 |
| Предисловие ко второму изданию | 17 |
| Предисловие к первому изданию | 19 |
| Введение..... | 21 |
| Благодарности | 23 |
| О книге | 24 |
| Что нового в третьем издании..... | 24 |
| Для кого эта книга | 25 |
| Структура книги | 25 |
| О коде в книге | 26 |
| Программные требования..... | 27 |
| Форум LiveBook..... | 27 |
| Другие проекты Роя Ошерова..... | 27 |
| Другие проекты Владимира Хорикова | 28 |
| Об авторах..... | 29 |
| Иллюстрация на обложке..... | 30 |
| От издательства..... | 31 |
| О научном редакторе русского издания | 31 |

Часть I. Первые шаги

| | |
|--|-----------|
| Глава 1. Основы юнит-тестирования | 34 |
| 1.1. Первый шаг | 36 |
| 1.2. Определение юнит-тестирования шаг за шагом..... | 37 |
| 1.3. Точки входа и точки выхода..... | 38 |
| 1.4. Типы точек выхода | 44 |
| 1.5. Разные точки выхода, разные методы | 45 |
| 1.6. Написание теста с нуля..... | 45 |

6 Оглавление

| | | |
|---------|---|-----------|
| 1.7. | Характеристики хорошего юнит-теста..... | 48 |
| 1.7.1. | Как выглядит хороший юнит-тест? | 49 |
| 1.7.2. | Чек-лист юнит-теста..... | 51 |
| 1.8. | Интеграционные тесты | 51 |
| 1.9. | Окончательное определение | 56 |
| 1.10. | Разработка через тестирование..... | 57 |
| 1.10.1. | TDD не заменит хорошие юнит-тесты | 59 |
| 1.10.2. | Три основных навыка, необходимых для успешного применения TDD..... | 61 |
| | Итоги | 62 |
| | Глава 2. Первый юнит-тест | 64 |
| 2.1. | Знакомство с Jest | 65 |
| 2.1.1. | Подготовка среды | 65 |
| 2.1.2. | Подготовка рабочего каталога..... | 65 |
| 2.1.3. | Установка Jest..... | 66 |
| 2.1.4. | Создание файла теста..... | 66 |
| 2.1.5. | Выполнение Jest..... | 68 |
| 2.2. | Библиотека для тестирования, библиотека утверждений, запуск тестов и представление отчета | 70 |
| 2.3. | Что предлагают фреймворки юнит-тестирования..... | 71 |
| 2.3.1. | Фреймворки xUnit | 73 |
| 2.3.2. | Структуры xUnit, TAP и Jest | 74 |
| 2.4. | Проект Password Verifier | 75 |
| 2.5. | Первый тест Jest для verifyPassword | 76 |
| 2.5.1. | Паттерн «подготовь – действуй – проверь» | 76 |
| 2.5.2. | Тестируем тест | 77 |
| 2.5.3. | Имена USE | 77 |
| 2.5.4. | Сравнения строк и сопровождаемость | 78 |
| 2.5.5. | Использование describe() | 79 |
| 2.5.6. | Структура, подразумевающая контекст | 80 |
| 2.5.7. | Функция it() | 81 |
| 2.5.8. | Две разновидности Jest | 81 |
| 2.5.9. | Рефакторинг рабочего кода..... | 82 |
| 2.6. | Решение с beforeEach() | 85 |
| 2.6.1. | beforeEach() и утомительная прокрутка | 86 |

| | | |
|--------|---|----|
| 2.7. | Решение с фабричным методом..... | 89 |
| 2.7.1. | Полная замена beforeEach() фабричными методами..... | 90 |
| 2.8. | Возвращение к test()..... | 92 |
| 2.9. | Рефакторинг: параметризованные тесты | 93 |
| 2.10. | Проверка ожидаемых ошибок..... | 95 |
| 2.11. | Назначение категорий тестов..... | 97 |
| | Итоги..... | 98 |

Часть 2. Приемы и методы

Глава 3. Устранение зависимостей при помощи стабов102

| | | |
|--------|---|-----|
| 3.1. | Типы зависимостей | 103 |
| 3.2. | Причины для использования стабов..... | 106 |
| 3.3. | Общепринятые подходы к созданию стабов..... | 108 |
| 3.3.1. | Создание стаба для времени с внедрением параметра | 108 |
| 3.3.2. | Зависимости, внедрения и контроль | 110 |
| 3.4. | Функциональные методы внедрения | 111 |
| 3.4.1. | Внедрение функции..... | 112 |
| 3.4.2. | Внедрение зависимости через частичное применение | 113 |
| 3.5. | Модульные методы внедрения | 113 |
| 3.6. | Переход к объектам с функциями-конструкторами | 117 |
| 3.7. | Объектно-ориентированные методы внедрения | 117 |
| 3.7.1. | Внедрение через конструктор | 118 |
| 3.7.2. | Внедрение объекта вместо функции..... | 119 |
| 3.7.3. | Выделение общего интерфейса..... | 123 |
| | Итоги..... | 126 |

Глава 4. Тестирование взаимодействий с использованием моков128

| | | |
|--------|--|-----|
| 4.1. | Тестирование взаимодействий, моки и стабы..... | 129 |
| 4.2. | Зависимость от логгера..... | 130 |
| 4.3. | Стандартный стиль: рефакторинг с введением параметра..... | 132 |
| 4.4. | Почему важно отличать моки от стабов | 134 |
| 4.5. | Моки в модульном стиле | 135 |
| 4.5.1. | Пример рабочего кода | 135 |
| 4.5.2. | Рефакторинг рабочего кода в стиле модульного внедрения | 137 |
| 4.5.3. | Пример теста с внедрением в модульном стиле | 138 |

8 Оглавление

| | | |
|--------|--|------------|
| 4.6. | Моки в функциональном стиле | 139 |
| 4.6.1. | Использование стиля каррирования | 139 |
| 4.6.2. | Работа с функциями высшего порядка без каррирования..... | 140 |
| 4.7. | Моки в объектно-ориентированном стиле | 141 |
| 4.7.1. | Рефакторинг рабочего кода для внедрения..... | 141 |
| 4.7.2. | Рефакторинг рабочего кода с внедрением интерфейсов | 143 |
| 4.8. | Сложные интерфейсы..... | 145 |
| 4.8.1. | Пример сложного интерфейса | 146 |
| 4.8.2. | Написание тестов со сложными интерфейсами | 146 |
| 4.8.3. | Недостатки прямого использования сложных интерфейсов | 148 |
| 4.8.4. | Принцип разделения интерфейсов | 148 |
| 4.9. | Частичные моки | 148 |
| 4.9.1. | Пример частичного мока..... | 149 |
| 4.9.2. | Объектно-ориентированный пример частичного мока..... | 149 |
| | Итоги..... | 151 |
| | Глава 5. Изолирующие фреймворки..... | 152 |
| 5.1. | Определение изолирующих фреймворков | 153 |
| 5.1.1. | Выбор разновидности: свободные и типизированные | 154 |
| 5.2. | Динамическое создание фейков для модулей | 154 |
| 5.2.1. | Некоторые особенности API Jest, на которые следует обратить внимание..... | 157 |
| 5.2.2. | Абстрагирование прямых зависимостей | 158 |
| 5.3. | Функциональные динамические моки и стабы | 159 |
| 5.4. | Объектно-ориентированные динамические моки и стабы | 160 |
| 5.4.1. | Использование фреймворка со свободной типизацией..... | 160 |
| 5.4.2. | Переход на фреймворк с поддержкой типов..... | 162 |
| 5.5. | Динамическое создание стабов..... | 164 |
| 5.5.1. | Объектно-ориентированный пример с моком и стабом | 165 |
| 5.5.2. | Стабы и моки с substitute.js..... | 166 |
| 5.6. | Преимущества и опасности изолирующих фреймворков | 168 |
| 5.6.1. | Моки в большинстве случаев не нужны..... | 169 |
| 5.6.2. | Нечитаемый код теста..... | 169 |
| 5.6.3. | Проверка не того, что нужно | 170 |
| 5.6.4. | Наличие более одного мока на тест..... | 170 |

| | |
|---|------------|
| 5.6.5. Излишняя детализация тестов | 170 |
| Итоги..... | 171 |
| Глава 6. Юнит-тестирование асинхронного кода | 172 |
| 6.1. Асинхронная загрузка данных | 173 |
| 6.1.1. Первая попытка написания интеграционного теста | 174 |
| 6.1.2. Ожидание действия | 175 |
| 6.1.3. Интеграционное тестирование <code>async/await</code> | 175 |
| 6.1.4. Проблемы с интеграционными тестами | 176 |
| 6.2. Создание кода, удобного для юнит-тестов..... | 177 |
| 6.2.1. Выделение точки входа | 177 |
| 6.2.2. Паттерн «выделение адаптера» | 183 |
| 6.3. Таймеры..... | 191 |
| 6.3.1. Создание стабов для таймеров | 191 |
| 6.3.2. Фейки <code>setTimeout</code> средствами Jest | 192 |
| 6.4. Распространенные события | 194 |
| 6.4.1. Генераторы событий..... | 194 |
| 6.4.2. События <code>click</code> | 195 |
| 6.5. Использование DOM Testing Library | 198 |
| Итоги..... | 199 |

Часть 3. Код тестов

| | |
|--|------------|
| Глава 7. Достоверные тесты..... | 202 |
| 7.1. Как определить, что тесту можно доверять | 203 |
| 7.2. Почему тесты не проходят..... | 204 |
| 7.2.1. В рабочем коде была обнаружена реальная ошибка | 204 |
| 7.2.2. Тест не проходит из-за ошибки в самом тесте | 204 |
| 7.2.3. Тест устарел из-за изменения в функциональности | 206 |
| 7.2.4. Тест конфликтует с другим тестом..... | 206 |
| 7.2.5. Ненадежность теста | 207 |
| 7.3. Избегание логики в юнит-тестах..... | 207 |
| 7.3.1. Логика в утверждениях: создание динамических ожидаемых значений..... | 208 |
| 7.3.2. Другие формы логики | 209 |
| 7.3.3. Еще больше логики | 211 |

10 Оглавление

| | | |
|--------|---|-----|
| 7.4. | Ложное чувство доверия к проходящим тестам | 211 |
| 7.4.1. | Тесты, которые ничего не проверяют | 212 |
| 7.4.2. | Тесты непонятны..... | 213 |
| 7.4.3. | Смешение юнит-тестов с ненадежными интеграционными тестами | 213 |
| 7.4.4. | Тестирование нескольких точек выхода..... | 214 |
| 7.4.5. | Тесты, которые продолжают изменяться | 216 |
| 7.5. | Ненадежные тесты..... | 217 |
| 7.5.1. | Что делать при обнаружении ненадежного теста? | 219 |
| 7.5.2. | Предотвращение ненадежности в высокоуровневых тестах ... | 220 |
| | Итоги..... | 221 |

Глава 8. Простота в сопровождении 222

| | | |
|--------|---|-----|
| 8.1. | Изменения из-за непроходящих тестов..... | 223 |
| 8.1.1. | Тест неактуален или конфликтует с другим тестом | 223 |
| 8.1.2. | Изменения в API рабочего кода | 224 |
| 8.1.3. | Изменения в других тестах | 227 |
| 8.2. | Рефакторинг для улучшения сопровождаемости | 232 |
| 8.2.1. | Избегайте тестирования приватных или защищенных методов | 232 |
| 8.2.2. | Соблюдение принципа DRY в тестах | 234 |
| 8.2.3. | Нежелательность подготовительных методов | 234 |
| 8.2.4. | Использование параметризованных тестов для устранения дублирования | 235 |
| 8.3. | Избегайте излишней детализации | 236 |
| 8.3.1. | Излишняя детализация внутреннего поведения с моками..... | 237 |
| 8.3.2. | Излишняя детализация выводов | 239 |
| | Итоги..... | 243 |

Часть 4. Проектирование и процесс

Глава 9. Читабельность 246

| | | |
|------|--|-----|
| 9.1. | Выбор имен юнит-тестов | 247 |
| 9.2. | Магические значения и имена переменных | 248 |
| 9.3. | Отделение проверок от действий..... | 250 |
| 9.4. | Подготовка и завершение | 250 |
| | Итоги..... | 252 |

| | |
|--|------------|
| Глава 10. Разработка стратегии тестирования..... | 253 |
| 10.1. Основные виды и уровни тестирования..... | 254 |
| 10.1.1. Критерии для выбора вида теста..... | 255 |
| 10.1.2. Юнит-тесты и компонентные тесты | 255 |
| 10.1.3. Интеграционные тесты..... | 257 |
| 10.1.4. Тесты API | 257 |
| 10.1.5. Изолированные тесты E2E/UI..... | 258 |
| 10.1.6. Системные тесты E2E/UI | 259 |
| 10.2. Антипаттерны в разработке стратегии тестирования | 260 |
| 10.2.1. Антипаттерн «только сквозные тесты» | 260 |
| 10.2.2. Антипаттерн «только низкоуровневые тесты» | 263 |
| 10.2.3. Рассоединение низкоуровневых и высокоуровневых тестов | 265 |
| 10.3. Рецепты тестирования как стратегия | 266 |
| 10.3.1. Как написать рецепт тестирования..... | 266 |
| 10.3.2. Когда писать и использовать рецепты тестирования? | 268 |
| 10.3.3. Правила составления рецептов тестирования | 269 |
| 10.4. Управление конвейером поставки ПО | 270 |
| 10.4.1. Конвейеры поставки и информационные конвейеры | 270 |
| 10.4.2. Параллелизация уровней тестирования | 272 |
| Итоги..... | 273 |
| Глава 11. Интеграция юнит-тестирования в организацию | 275 |
| 11.1. Как стать инициатором изменений | 276 |
| 11.1.1. Подготовьтесь к непростым вопросам | 276 |
| 11.1.2. Убедите окружающих: сторонников и скептиков | 276 |
| 11.1.3. Определите возможные отправные точки..... | 278 |
| 11.2. Пути к успеху | 279 |
| 11.2.1. Партизанская реализация (снизу вверх)..... | 280 |
| 11.2.2. Привлечение руководства (сверху вниз)..... | 280 |
| 11.2.3. Эксперименты как первый шаг | 281 |
| 11.2.4. Привлечение внешнего сторонника | 282 |
| 11.2.5. Демонстрация прогресса..... | 282 |
| 11.2.6. Ориентация на конкретные цели, метрики и KPI | 284 |
| 11.2.7. Понимание возможных препятствий..... | 286 |

12 Оглавление

| | |
|--|------------|
| 11.3. Возможные неудачи..... | 287 |
| 11.3.1. Недостаточный направляющий импульс | 287 |
| 11.3.2. Недостаток политической поддержки | 287 |
| 11.3.3. Ситуативные реализации и первые впечатления | 288 |
| 11.3.4. Недостаток поддержки в команде | 288 |
| 11.4. Факторы влияния..... | 289 |
| 11.5. Непростые вопросы и ответы | 291 |
| 11.5.1. Насколько юнит-тестирование удлиният текущий процесс? | 291 |
| 11.5.2. Не будет ли юнит-тестирование угрожать моей работе в отделе QA? | 293 |
| 11.5.3. Есть ли доказательства, что юнит-тестирование действительно помогает? | 294 |
| 11.5.4. Почему отдел QA все еще находит ошибки? | 294 |
| 11.5.5. У нас большой объем кода без тестов: с чего начинать?..... | 294 |
| 11.5.6. А если мы разрабатываем комбинацию программного и аппаратного обеспечения? | 295 |
| 11.5.7. Как узнать, что в наших тестах нет ошибок? | 295 |
| 11.5.8. Зачем мне нужны тесты, если отладчик показывает, что мой код работает? | 295 |
| 11.5.9. Как насчет TDD? | 296 |
| Итоги | 296 |
| Глава 12. Работа с унаследованным кодом..... | 297 |
| 12.1. С чего начинать добавление тестов? | 298 |
| 12.2. Принятие решения по стратегии выбора | 300 |
| 12.2.1. Плюсы и минусы стратегии «начать с простого»..... | 300 |
| 12.2.2. Плюсы и минусы стратегии «начать со сложного» | 301 |
| 12.3. Написание интеграционных тестов перед рефакторингом | 302 |
| 12.3.1. Прочитайте книгу Майкла Физерса об унаследованном коде | 303 |
| 12.3.2. Используйте CodeScene для анализа своего рабочего кода..... | 304 |
| Итоги | 304 |
| Приложения. Манки-патчинг функций и модулей | 305 |
| П.1. Обязательное предупреждение..... | 305 |
| П.2. Манки-патчинг функций и возможные проблемы | 306 |

| | | |
|--------|--|-----|
| П.2.1. | Манки-патчинг функции по схеме Jest | 308 |
| П.2.2. | Шпионы Jest..... | 308 |
| П.2.3. | spyOn с mockImplementation() | 309 |
| П.3. | Игнорирование целого модуля в Jest..... | 310 |
| П.4. | Моделирование поведения модуля в каждом тесте | 311 |
| П.4.1. | Создание стаба для модуля базовым вызовом require.cache | 312 |
| П.4.2. | Создание стабов для нестандартных данных из модулей в Jest затруднено | 314 |
| П.4.3. | Избегайте ручных моков Jest | 316 |
| П.4.4. | Создание стаба для модуля с Sinon.js..... | 316 |
| П.4.5. | Создание стаба для модуля с testdouble..... | 317 |

Отзывы о втором издании

Эта книга — нечто особенное. Каждая глава строится на материале предыдущих, что позволяет достичь невероятной глубины. Приготовьтесь, вас ждет сюрприз.

*Из предисловия ко второму изданию
от Роберта С. Мартина, cleancoder.com*

Лучший способ освоить юнит-тестирование — прочесть книгу, которая стала классикой в своей области.

*Рафаэль Фария (Raphael Faria),
LG Electronics*

Учит вас философии, а также всем техническим тонкостям юнит-тестирования.

*Прадип Челлапан (Pradeep Chellappan),
Microsoft*

Когда участники моей команды спрашивают меня, как правильно писать юнит-тесты, я просто отвечаю: прочитайте эту книгу!

*Алессандро Кампейс (Alessandro Campeis),
Vimar SpA*

Однозначно лучший источник информации о юнит-тестировании.

*Калеб Педерсон (Kaleb Pederson),
Next IT Corporation*

Самое полезное и актуальное руководство по юнит-тестированию, которое я когда-либо читал.

*Франческо Гоги (Francesco Goggi),
FIAT*

Обязательный источник для каждого серьезного разработчика .NET, который желает изучить или отшлифовать свои знания юнит-тестирования.

*Карл Метивье (Karl Metivier),
Desjardins Security Financial*

*Посвящаю Тал, Итамар, Авиву
и Идо — моей семье.*

Рой Ошеров

Моей жене Нине и сыну Тимоти.

Владимир Хориков

Предисловие ко второму изданию

Это было в 2009 году. Я выступал с докладом на Норвежской конференции разработчиков в Осло. (Ах, Осло в июне!) Конференция проводилась на громадной спортивной арене. Организаторы разделили трибуны на секции, построили перед ними сцены и драпировали толстой черной тканью, чтобы создать восемь разных «комнат». Помню, я только что завершил свой доклад о TDD (или о SOLID, или об астрономии, или о чем-то еще), когда внезапно с соседней сцены донеслось громкое и пронзительное пение под звуки гитары.

Я заглянул за драпировку и увидел парня, который устроил этот концерт. Конечно, это был Рой Ошеров.

Те из вас, кто знаком со мной, знают, что запеть посреди технического доклада о программировании вполне *в моем духе*, если я буду в подходящем настроении. Повернувшись спиной к аудитории, я подумал, что этот Ошеров — родственная душа и с ним стоит познакомиться поближе.

Именно так я и поступил. Настолько, что он внес значительный вклад в мою последнюю книгу «The Clean Coder»¹ и три дня преподавал вместе со мной на курсе по TDD. Мое общение с Роем было исключительно позитивным, и я надеюсь, что мы еще не раз встретимся.

Предсказываю, что ваш опыт общения с Роем при чтении книги «Искусство юнит-тестирования» тоже будет очень позитивным, потому что эта книга — нечто особенное.

Вы читали романы Миченера? Я не читал; но мне говорили, что все они начинаются «с атома». Книга, которую вы держите в руках, — не роман Джеймса Миченера, но она тоже начинается с атома — атома юнит-тестирования.

Но не ошибайтесь, пролистав несколько начальных страниц. Это *не* обычное введение в юнит-тестирование. Книга начинается, действительно, с вводного

¹ Мартин Р. Идеальный программист. СПб.: Питер.

18 Предисловие ко второму изданию

материала, и опытные разработчики могут пропустить эти начальные главы. В дальнейшем каждая глава строится на материале предыдущих, что позволяет достичь невероятной глубины. В самом деле, когда я читал последнюю главу (еще не зная, что она последняя), я подумал, что следующая будет посвящена достижению мира во всем мире — потому что чем еще можно заняться после решения проблемы внедрения юнит-тестирования в упретых организациях со старыми унаследованными системами?

Перед вами техническая книга — глубоко техническая. В ней много кода. И это хорошо. Но Рой интересуется не только техническими вопросами. Время от времени он достает гитару и начинает напевать, рассказывая случаи из своего профессионального прошлого, или пускается в философские рассуждения о смысле архитектур или определении интеграции. Кажется, ему доставляет удовольствие делиться с нами историями о том, что у него очень плохо получилось в темном, мрачном 2006-м.

Да, и пусть вас не беспокоит, что код написан на C#. В конце концов, кто отличит C# от Java? Верно? Кроме того, это вообще неважно. Он может использовать C# как средство для выражения своих намерений, но уроки этой книги также применимы к Java, C, Ruby, Python, PHP или любому языку программирования (кроме, возможно, COBOL).

И новички, недавно занявшиеся юнит-тестированием и разработкой через тестирование, и опытные ветераны найдут в этой книге что-то для себя. Так что приготовьтесь, вас ждет сюрприз: Рой споет вам песню «Искусство юнит-тестирования».

И пожалуйста, Рой, настрой наконец свою гитару!

*Роберт С. Мартин (дядюшка Боб),
cleancoder.com*

Предисловие к первому изданию

Когда Рой Ошеров сказал мне, что работает над книгой по юнит-тестированию, я очень обрадовался. Разговоры о тестировании велись в отрасли годами, но толкового материала о юнит-тестировании было относительно немного. У меня есть книги, посвященные конкретно разработке через тестирование, и книги о тестировании вообще, но нет ни одного подробного руководства по юнит-тестированию — книги, которая бы представила тему и провела читателя от первых шагов к общепринятым практикам. И этот факт приводит в изумление. Юнит-тестирование — не новая практика. Как же мы дошли до такой жизни?

Говорить, что мы работаем в очень молодой отрасли, — почти что банальность, но это правда. Математики заложили основы нашей работы менее 100 лет назад, но только за последние 60 лет мы получили оборудование, достаточно быстрое для того, чтобы воспользоваться их открытиями. В нашей отрасли изначально существовал разрыв между теорией и практикой, и мы только сейчас начинаем понимать, как он повлиял на нас.

На первых порах машинное время стоило дорого. Программы приходилось запускать в пакетном режиме. Программистам выделялся заранее определенный отрезок времени, они набивали свои программы на перфокартах и относили в машинный зал. Если программа содержала ошибки, время было потеряно, так что код приходилось проверять с бумагой и карандашом, в уме прорабатывая все возможные сценарии, все граничные случаи. Сомневаюсь, что кто-нибудь мог даже представить себе концепцию автоматизированного юнит-тестирования. Зачем использовать машину для тестирования, когда на ней можно было решать задачи, для решения которых она создавалась? Нехватка ресурсов держала нас во тьме.

Позднее машины стали работать быстрее, и нас опьянили интерактивные вычисления. Вы могли просто ввести код и изменить его по своему усмотрению. Идея проверки кода путем ручной имитации работы программы (*desk checking*) испарилась, и мы отчасти утратили дисциплинированность. Мы знали, что

программировать — сложно, но это означало лишь то, что придется проводить больше времени за компьютером, изменяя строки и символы, пока не будет найдено волшебное заклинание, которое сработает.

Мы сходу перешли от нехватки к изобилию и упустили промежуточное состояние, но теперь возвращаемся к нему. Автоматизированное юнит-тестирование объединяет дисциплинированность ручной имитации работы программы с ново-приобретенным представлением о компьютере как ресурсе разработки. Мы можем писать автоматизированные тесты на языке, на котором ведется разработка, для проверки результатов нашего труда — не однократной, а настолько частой, насколько часто мы сможем запускать эти тесты. Я не думаю, что в области разработки найдется другая, столь же мощная практика.

Когда я пишу эти строки сейчас, в 2009 году, я счастлив видеть, что книга Роя отправилась в печать. Это практическое руководство, которое поможет вам начать писать тесты, а также послужит отличным справочником по этой теме. Книга «Искусство юнит-тестирования с примерами на JavaScript» не ориентируется на идеализированные сценарии. Она учит тестировать код в том виде, в котором он существует в реальности, пользоваться популярными фреймворками и, самое важное, писать код, который намного проще в тестировании.

«Искусство юнит-тестирования с примерами на JavaScript» — важная книга, которую следовало бы написать много лет назад, но мы тогда не были к этому готовы. Сейчас это время пришло. Радуйтесь.

*Майкл Физерс,
Object Mentor*

Введение

В одном из самых больших неудачных проектов, в котором я участвовал, были юнит-тесты. Вернее, я так думал. Я возглавлял группу программистов, работавших над приложением для выставления счетов, и разработка велась по методологии TDD (разработки через тестирование) — пишем сначала тест, затем код, убеждаемся, что тест не прошел, обеспечиваем его прохождение, проводим рефакторинг и повторяем все заново.

Первые несколько месяцев работы над проектом все было замечательно. Все шло по плану, и у нас были тесты, которые доказывали, что код работает. Но с течением времени требования изменялись. Нам приходилось изменять свой код в соответствии с новыми требованиями, а когда это происходило, тесты ломались и их приходилось исправлять. Код работал, но написанные нами тесты были настолько хрупкими, что ломались от малейшего изменения в программе. Перспектива изменения кода в классе или методе стала пугать, потому что нам также приходилось исправлять все сопутствующие юнит-тесты.

Что еще хуже, некоторые тесты стали непригодными, потому что написавшие их люди покинули проект и никто не знал, как сопровождать эти тесты и что они вообще тестируют. Имена, которые мы давали своим методам юнит-тестов, были недостаточно понятными, а некоторые тесты опирались на результаты других тестов. В итоге нам пришлось выкинуть большую часть этих тестов менее чем через полгода после начала проекта.

Проект потерпел неудачу, потому что от написанных нами тестов было больше вреда, чем пользы. На попытки разобраться в них и на сопровождение в итоге уходило больше времени, чем они экономили, поэтому мы перестали ими пользоваться. Я переключился на другие проекты, где у нас получалось лучше писать юнит-тесты, и те приносили огромную пользу, экономя массу времени на отладке и интеграции. После того первого неудачного проекта я стал собирать лучшие практики юнит-тестирования и использовать их в последующих проектах. С каждым новым проектом, над которым я работал, я узнавал несколько новых полезных приемов.

Эта книга должна объяснить читателю, как писать юнит-тесты (и как сделать их простыми в сопровождении, читабельными и достоверными) независимо от того, с каким языком или интегрированной средой разработки вы работаете. В книге изложены основы написания юнит-тестов, а также тестирования взаимодействий. Она представляет лучшие практики написания юнит-тестов, управления ими и их сопровождения в реальных условиях.

Рой Ошеров

Когда издательство Manning предложило мне дописать книгу по юнит-тестированию, которая уже почти была завершена, моим первым побуждением было отказаться. В конце концов, у меня уже есть своя книга по этой же теме, так зачем мне еще работать над чужим проектом? Но я изменил свое решение, когда понял, что речь идет ни много ни мало о книге Роя «Искусство юнит-тестирования с примерами на JavaScript». Первое ее издание было одной из первых книг, которые я прочитал по этой теме, и оно помогло сформировать мои взгляды на юнит-тестирование. Для меня будет честью внести свой вклад в третье издание этого выдающегося труда.

Лично я считаю эту книгу отличным введением в тему юнит-тестирования. А когда вы завершите ее и будете готовы копнуть глубже, найдите мою книгу «Unit Testing Principles, Practices, and Patterns»¹ (Manning, 2020).

Владимир Хориков

¹ Хориков В. Принципы юнит-тестирования. СПб.: Питер.

Благодарности

Мы хотим поблагодарить многих рецензентов этой рукописи. Полученная от них обратная связь помогла нам улучшить книгу. Спасибо вам всем: Абду Самаду Cap (Aboudou Samadou Sare), Адхир Рамиджиаван (Adhir Ramjiawan), Адриан Бейерц (Adriaan Beiertz), Ален Ломпо (Alain Lompo), Барнаби Норман (Barnaby Norman), Чарльз Лэм (Charles Lam), Конор Редмонд (Conor Redmond), До Морина (Daut Morina), Эсрек Дурна (Esref Durna), Фостер Хейнс (Foster Haines), Харинат Маллепалли (Harinath Mallepally), Джарен Дункан (Jared Duncan), Джейсон Хейлз (Jason Hales), Жауме Лопес (Jaume López), Джереми Чен (Jeremy Chen), Джоэл Холмс (Joel Holmes), Джон Ларсен (John Larsen), Джонатан Ривз (Jonathan Reeves), Хорхе Э. Бо (Jorge E. Bo), Кент Спиллер (Kent Spillner), Ким Габриэlsen (Kim Gabrielsen), Марсель ван ден Бринк (Marcel van den Brink), Марк Грэм (Mark Graham), Мэтт Ван Винкл (Matt Van Winkle), Маттео Баттиста (Matteo Battista), Маттео Гилдон (Matteo Gildone), Майк Холкомб (Mike Holcomb), Оливэр Кортен (Oliver Korten), Онофри Джордж (Onofrei George), Пол Робак (Paul Roebuck), Пабло Эррера Дж. (Pablo Herrera J.), Патрис Мальдаг (Patrice Maldague), Рахул Модпур (Rahul Modpur), Ранжит Сахай (Ranjit Sahai), Рич Йонц (Rich Yonts), Ричард Мейнсен (Richard Meinsen), Родриго Энсинас (Rodrigo Encinas), Рональд Борман (Ronald Borman), Сачин Сингхи (Sachin Singhi), Саманта Берк (Samantha Berk), Сандер Зегвельд (Sander Zegveld), Сатеш Кумар Сахи (Satej Kumar Sahu), Шейн Корнуэлл (Shayn Cornwell), Таня Уилк (Tanya Wilke), Том Мэдден (Tom Madden), Удит Бхардвадж (Udit Bhardwaj) и Вадим Турков (Vadim Turkov).

В создании успешной книги участвуют многие люди. Нам также хотелось бы поблагодарить шеф-редактора Manning Майкла Стивенса (Michael Stephens), ведущего редактора Коннора О'Брайена (Connor O'Brien), выпускающего редактора Майка Шепарда (Mike Shepard), корректора Жана-Франсуа Морена (Jean-François Morin) и рецензентов издательства Адриану Сабо (Adriana Sabo) и Дуню Никитович (Dunja Nikitović). Мы также благодарим всех остальных работников Manning, которые трудились над третьим изданием.

И наконец, слова благодарности обращены к читателям первой версии книги в программе раннего доступа Manning, поделившимся своими комментариями на форуме. Вы помогли привести эту книгу к окончательному виду.

О книге

Одно из самых умных высказываний относительно обучения (уже не помню, от кого я его слышал) — если ты хочешь в чем-то по-настоящему разбираться, начни преподавать это. Работа над первым изданием этой книги, которое вышло в 2009 году, стала для меня важным уроком. Изначально я написал книгу, потому что мне надоело снова и снова отвечать на одни и те же вопросы. Впрочем, были и другие причины. Я хотел опробовать что-то новое; мне хотелось поставить эксперимент; мне было интересно, что можно узнать при написании книги — любой книги. Юнит-тестирование было темой, в которой я разбираюсь... как мне казалось. Проблема в том, что чем больше у тебя опыта, тем глупее ты себя чувствуешь.

В первом издании были некоторые положения, с которыми я сегодня не могу согласиться, — например, что «юнит» соответствует методу. Это вообще не так. Юнит соответствует единице работы, как будет показано в главе 1 третьего издания. Единица работы может быть маленькой, как отдельный метод, или большой, состоящей из нескольких классов (а возможно, и сборок). Также изменились некоторые другие аспекты, как будет показано в следующем разделе.

ЧТО НОВОГО В ТРЕТЬЕМ ИЗДАНИИ

В третьем издании мы переключились с .NET на JavaScript и TypeScript. Разумеется, также изменились все соответствующие инструменты и фреймворки. Например, вместо программ для запуска тестов NUnit и NSubstitute мы использовали Jest — как фреймворк юнит-тестирования и как библиотеку моков.

В главу, посвященную реализации юнит-тестирования на уровне организации, были добавлены новые приемы.

В коде, приведенном в книге, много структурных изменений. Они в основном связаны с использованием языков с динамической типизацией (таких, как

JavaScript), но мы также уделим внимание языкам со статической типизацией при помощи TypeScript.

Обсуждение достоверности, простоты сопровождения и читабельности тестов было развернуто на три отдельные главы. Также была добавлена новая глава о стратегиях тестирования: как сделать правильный выбор между разными типами тестов и какие приемы при этом использовать.

ДЛЯ КОГО ЭТА КНИГА

Книга предназначена для всех программистов, которые пишут код и желают освоить лучшие практики юнит-тестирования. Все примеры написаны на JavaScript и TypeScript, поэтому они будут особенно полезны JavaScript-разработчикам. Тем не менее материал книги применим в равной степени ко многим (если не ко всем) объектно-ориентированным языкам со статической типизацией (C#, VB.NET, Java, C++ и т. д.). Если вы являетесь архитектором, разработчиком, тимлидом, QA-инженером (который пишет код) или начинающим программистом, книга вам также будет полезна.

СТРУКТУРА КНИГИ

Если вы еще никогда не писали юнит-тесты, лучше прочитать эту книгу от начала до конца, чтобы получить полное представление о процессе. Если у вас уже имеется опыт, можно перейти к любой главе на ваше усмотрение. Книга разделена на 4 части.

Часть 1 описывает первые шаги в написании юнит-тестов. В главах 1 и 2 представлены основы — в частности, как использовать фреймворк тестирования (Jest); кроме того, в них описаны такие понятия автоматизированного тестирования, как библиотеки для тестирования, библиотеки утверждений и программы для запуска тестов. Также в этих главах представлены концепции утверждений, игнорирования тестов, тестирования единиц работы, три типа конечных результатов юнит-тестов и три разновидности тестов, которые для них понадобятся: тесты значений, тесты состояний и тесты взаимодействий.

В части 2 обсуждаются продвинутые методы устранения зависимостей: моки, стабы, изолирующие фреймворки и зарекомендовавшие себя на практике паттерны рефакторинга кода. В главе 3 представлена концепция стабов и показано, как создавать и использовать их вручную. Глава 4 знакомит читателя с тестированием взаимодействий с помощью моков. Глава 5 объединяет эти две концепции и показывает, как изолирующие фреймворки комбинируют их

для автоматизации. В главе 6 углубленно рассматривается тема тестирования асинхронного кода.

Часть 3 посвящена организации кода тестов, паттернам запуска и рефакторинга ее структуры, а также лучшим практикам для написания тестов. В главе 7 обсуждаются методы написания достоверных тестов. В главе 8 приводятся лучшие практики юнит-тестирования для создания тестов, простых в сопровождении.

В части 4 рассматривается реализация изменений в компании и возможности работы с существующим кодом. Глава 9 посвящена читабельности тестов. Глава 10 показывает, как разработать стратегию тестирования. В главе 11 обсуждаются проблемы, возникающие при попытке внедрить юнит-тестирование в организацию, и их решения, а также даются ответы на некоторые вопросы, которые вам могут задавать в ходе такой работы. В главе 12 речь пойдет о внедрении юнит-тестирования в унаследованный код. В ней описаны несколько подходов к началу тестирования, а также рассматриваются некоторые средства для тестирования нетестируемого кода.

В приложении перечислены методы исправления ошибок во время исполнения программы (манки-патчинг), которые могут вам пригодиться в ходе тестирования.

О КОДЕ В КНИГЕ

Весь исходный код в листингах или тексте форматируется моноширинным шрифтом, в отличие от обычного текста. Иногда в листингах также применяется **полужирный шрифт**, чтобы выделить фрагменты, изменившиеся по сравнению с предыдущим примером или которые должны измениться в следующем примере. Во многих листингах код снабжается аннотациями для выделения ключевых концепций.

Исходный код книги можно загрузить с GitHub по адресу <https://github.com/royosherove/aout3-samples>, а также на сайте издателя по адресу <https://www.manning.com/books/the-art-of-unit-testing-third-edition>. Исполняемые фрагменты кода можно загрузить из версии liveBook (электронной) по адресу <https://livebook.manning.com/book/the-art-of-unit-testing-third-edition>.

ПРОГРАММНЫЕ ТРЕБОВАНИЯ

Чтобы использовать код, приведенный в книге, вам понадобится среда VS Code (распространяется бесплатно). Также вам понадобятся Jest (бесплатный фреймворк, распространяемый с открытым кодом) и другие средства, о которых будет рассказано по мере надобности. Все программные средства, упоминаемые в книге, либо распространяются бесплатно с открытым кодом, либо имеют пробные версии, которые можно свободно использовать во время чтения книги.

ФОРУМ LIVEBOOK

Приобретая книгу «Искусство юнит-тестирования с примерами на JavaScript», вы получаете бесплатный доступ к закрытому веб-форуму издательства Manning (на английском языке), на котором можно оставлять комментарии о книге, задавать технические вопросы и получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, откройте страницу <https://livebook.manning.com/book/the-art-of-unit-testing-third-edition/discussion>. Информацию о форумах Manning и правилах поведения на них см. на <https://livebook.manning.com/discussion>.

В рамках своих обязательств перед читателями издательство Manning предоставляет ресурс для содержательного общения читателей и авторов. Эти обязательства не подразумевают конкретную степень участия автора, которое остается добровольным (и неоплачиваемым). Задавайте автору хорошие вопросы, чтобы он не терял интереса к происходящему! Форум и архивы обсуждений доступны на веб-сайте издательства, пока книга продолжает издаваться.

ДРУГИЕ ПРОЕКТЫ РОЯ ОШЕРОВА

Рой также является автором книг «Elastic Leadership: Growing Self-organizing Teams», доступной на сайте www.manning.com/books/elastic-leadership, и «Notes to a Software Team Leader: Growing Self-Organizing Teams» (Team Agile Publishing, 2014).

Другие ресурсы:

- Блог для руководителей команд, относящийся к теме книги, доступен по адресу <http://5whys.com>.
- Онлайн-курс TDD от Роя: <https://courses.osherove.com/courses>.

- Многочисленные бесплатные видеоролики о юнит-тестировании: <http://ArtOfUnitTesting.com> и <http://Osherove.com/Videos>.
- Рой непрерывно проводит обучение и консультации по всему миру. Чтобы заказать учебные семинары для вашей компании, с ним можно связаться по адресу <http://contact.osherove.com>.

ДРУГИЕ ПРОЕКТЫ ВЛАДИМИРА ХОРИКОВА

Владимир также является автором книги «Unit Testing Principles, Practices, and Patterns»¹.

Другие ресурсы:

- Блог для разработчиков, желающих больше узнать о юнит-тестировании и проектировании, управляемом предметной областью (DDD): <https://enterprisecraftsmanship.com/>.
- Видеокурсы по Pluralsight: <https://bit.ly/ps-all>.

¹ Хориков В. Принципы юнит-тестирования. СПб.: Питер.

Об авторах



Рой Ошеров — один из основателей ALT.NET, ранее работавший в Турецк на должности главного архитектора ПО. Он консультирует и обучает команды по всему миру тонкому искусству юнит-тестирования и разработки через тестирование, а также учит руководителей команд, как им стать более эффективными, на сайте 5whys.com. Рой опубликовал много видеороликов о юнит-тестировании на сайте ArtOfUnitTesting.com. Его можно пригласить для консультаций и обучения на сайте Osherove.com.



Владимир Хориков — Microsoft MVP, блогер и автор Pluralsight. Профессионально занимается разработкой программного обеспечения более 10 лет, включая обучение команд всем тонкостям юнит-тестирования. Владимир написал книгу «Unit Testing, Principles, Practices, and Patterns», опубликованную издательством Manning. Кроме того, он написал ряд популярных серий постов в блогах и создал обучающий онлайн-курс по теме юнит-тестирования. Главным преимуществом его стиля обучения, который часто хвалят студенты, является приверженность Владимира сильной теоретической основе, которую он затем применяет на практических примерах. Владимир ведет блог на сайте EnterpriseCraftsmanship.com.

Иллюстрация на обложке

На обложке книги «Искусство юнит-тестирования с примерами на JavaScript» изображен японец в церемониальном костюме (Japonais en costume de cérémonie). Иллюстрация взята из книги Джеймса Причарда (James Prichard) «Natural History of Man» — альбома с раскрашенными вручную литографиями, опубликованного в Англии в 1847 году. Наш дизайнер обложек нашел ее в антикварном магазине в Сан-Франциско.

В прежние времена по одежде человека можно было легко определить, где он живет и каковы его профессия или социальное положение. Издательство Manning отдает дань изобретательности и инициативности — качествам, присущим индустрии IT, — и в знак этого размещает на обложках своих книг иллюстрации, демонстрирующие богатое многовековое разнообразие региональных культур, оживающее на страницах старинных изданий, подобных этому.

От издательства

Мы выражаем огромную благодарность компании КРОК за помощь в работе над русскоязычным изданием книги и вклад в повышение качества переводной литературы.

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

Мы будем рады узнать ваше мнение!

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

О НАУЧНОМ РЕДАКТОРЕ РУССКОГО ИЗДАНИЯ

Дмитрий Колфилд — инженер-тестировщик в компании КРОК. Принимал участие в тестировании и поддержке высоконагруженных информационных систем (десктопные и веб-приложения). Также занимался тестированием миграций в нереляционных базах данных для информационных систем, обрабатывающих большие данные.

Часть I

Первые шаги

В этой части книги представлены основы юнит-тестирования.

В главе 1 я расскажу о том, что же называется «юнитом», покажу, какими признаками обладает «хорошее» юнит-тестирование, и сравню *юнит-тестирование с интеграционным тестированием*. Затем будут рассмотрены методология разработки через тестирование (TDD) и ее роль в отношении юнит-тестирования.

В главе 2 мы напишем первый юнит-тест с использованием Jest (популярного тестового фреймворка JavaScript). Вы познакомитесь с базовым API Jest, научитесь использовать тестовые утверждения и организовывать непрерывное выполнение тестов.

1

Основы юнит-тестирования

В ЭТОЙ ГЛАВЕ

- ✓ Определение точек входа и точек выхода
- ✓ Определение юнит-тестов и единиц работы
- ✓ Различия между юнит-тестированием и интеграционным тестированием
- ✓ Простой пример юнит-тестирования
- ✓ Разработка через тестирование

Ручные тесты врагу не пожелаешь. Вы пишете свой код, запускаете его в отладчике, нажимаете нужные клавиши в своем приложении, а затем повторяете все это снова, когда напишете новый код. А еще нужно не забыть проверить весь остальной код, на который мог повлиять новый. Еще больше ручной работы. Замечательно.

Полностью ручное выполнение тестов и регрессионного тестирования, когда вам приходится тупо повторять одни действия снова и снова, ненадежно и занимает слишком много времени. Наверное, в отрасли разработки ПО ничто другое не вызывает такой ненависти. Эти проблемы отчасти компенсируются инструментарием для написания автоматизированных тестов и его грамотным

применением, что экономит наше драгоценное время и избавляет от сложностей с отладкой. Фреймворки интеграционного и юнит-тестирования помогают разработчикам быстрее создавать тесты с набором заданных API, автоматически выполнять эти тесты и легко просматривать их результаты. И никогда не забывать про это! Вероятно, вы читаете эту книгу либо потому, что понимаете важность автоматизации тестов, либо вас заставил ее прочитать тот, кто это понимает. Неважно. Если вы считаете, что однообразное ручное тестирование — приятное занятие, то вам книга не понравится. Предполагается, что вы *хотите* научиться писать хорошие юнит-тесты.

Также предполагается, что вы умеете писать код на JavaScript или TypeScript с использованием возможностей как минимум ECMAScript 6 (ES6) и уверенно работаете с менеджером пакетов npm. Еще одно предположение — вы знакомы с системой контроля версий Git. Если ранее вы уже бывали на github.com и знаете, как клонировать репозиторий, этого будет достаточно.

Хотя листинги из книги написаны на JavaScript и TypeScript, вам не обязательно быть программистом JavaScript. Примеры в предыдущих изданиях были на C#, и я обнаружил, что около 80 % паттернов было достаточно легко адаптировать. Вы должны справиться с чтением этой книги, даже если ранее программировали на Java, .NET, Python, Ruby или других языках. Паттерны — всего лишь паттерны. Язык используется для демонстрации паттернов, но они сами к языку не привязаны.

Javascript и TypeScript

В книге приводятся примеры как на базовом JavaScript, так и на TypeScript. Я принимаю на себя всю ответственность за это смешение языков, но обещаю, что для этого есть веская причина; в книге затронуты все три парадигмы программирования в JavaScript: *процедурная, функциональная и объектно-ориентированная*.

Я выбираю обычный JavaScript для примеров, в которых применяются процедурные и функциональные решения. TypeScript используется для объектно-ориентированных примеров, потому что этот язык обеспечивает структуру, необходимую для выражения таких идей.

В предыдущих изданиях книги, когда я работал на C#, это было неважно. Но при переходе на язык JavaScript, в котором поддерживаются все эти парадигмы, обращение к TypeScript выглядит разумно.

Почему бы просто не использовать TypeScript для всех парадигм, спросите вы? Чтобы показать, что TypeScript не обязателен для написания юнит-тестов, а концепции юнит-тестирования работают независимо от конкретного языка, типа компилятора или статического анализатора.

Это означает, что, если вы занимались функциональным программированием, некоторые примеры в книге вам будут понятны, тогда как другие могут показаться излишне усложненными или слишком пространными. Ничто не мешает вам ограничиться только функциональными примерами.

Если вы занимались объектно-ориентированным программированием или у вас есть опыт работы на C#/Java, вам может показаться, что некоторые не объектно-ориентированные примеры излишне упрощены и не отражают повседневные ситуации в ваших проектах. Не тревожьтесь, в книге будет много разделов, относящихся к объектно-ориентированному стилю.

1.1. ПЕРВЫЙ ШАГ

Всегда все начинается с первого шага: вы впервые пишете программу, впервые заваливаете проект, или вам впервые удается сделать то, чего вы хотели. Вы никогда не забудете свой первый раз, и я надеюсь, вы не забудете свои первые тесты.

Вероятно, вы уже сталкивались с тестами в той или иной форме. Некоторые из ваших любимых проектов с открытым кодом включают папки `test` — и они присутствуют в ваших собственных проектах, находящихся в разработке. А может быть, вы уже написали несколько тестов самостоятельно и даже помните, что они были плохими, неуклюжими, медленными или слишком сложными в сопровождении. Что еще хуже, вам могло показаться, что тесты бессмысленны и приводят лишь к напрасной трате времени (к сожалению, такое впечатление складывается у многих). А может, у вас остались наилучшие впечатления от юнит-тестов и вы читаете эту книгу, чтобы понять, что вы могли упустить.

В этой главе анализируется «классическое» определение юнит-тестов, которое сравнивается с концепцией интеграционного тестирования. Многие путают эти два вида, но различия между ними очень важно понимать, потому что, как будет показано далее в книге, отделение юнит-тестов от других видов тестов может быть критично для безусловного доверия к прохождению или непрохождению тестов.

Мы также обсудим плюсы и минусы юнит-тестирования в сравнении с интеграционным тестированием, а также придем к более точному определению того, что считать «хорошим» юнит-тестом. Глава завершается кратким описанием разработки через тестирование (TDD, Test-Driven Development), потому что она часто ассоциируется с юнит-тестированием. Однако это отдельная методология, которой я настоятельно рекомендую уделить внимание (хотя она не является основной темой данной работы). В этой главе также будут затронуты концепции, которые более подробно рассматриваются в других разделах книги.

Начнем с определения того, что же представляет собой юнит-тест.

1.2. ОПРЕДЕЛЕНИЕ ЮНИТ-ТЕСТИРОВАНИЯ ШАГ ЗА ШАГОМ

Концепция юнит-тестирования появилась в области разработки ПО достаточно давно. Она известна с ранней эпохи языка программирования Smalltalk в 1970-х годах и снова и снова подтверждала свою репутацию одного из лучших способов повышения качества кода при более глубоком понимании функциональных требований модуля, класса или функции. Кент Бек (Kent Beck) ввел концепцию юнит-тестирования в Smalltalk, и с тех пор она была принята во многих других языках программирования, в результате чего юнит-тестирование стало считаться в высшей степени полезной практикой.

Чтобы увидеть, что *не должно* считаться нашим определением юнит-тестирования, начнем с Википедии. Я буду использовать это определение с некоторыми оговорками, потому что, по моему мнению, в нем отсутствуют многие важные составляющие, но оно получило широкое распространение из-за нехватки других хороших определений. Наше определение будет постепенно дополняться в этой главе, а итоговое определение появится в разделе 1.9.

Юнит-тесты обычно представляют собой автоматизированные тесты, написанные и запускаемые разработчиками для проверки того, что часть приложения — «юнит» (программная единица) — отвечает целям проектирования, а ее поведение соответствует желаемому. В процедурном программировании юнит может быть целым программным модулем, но чаще это отдельная функция или процедура. В объектно-ориентированном программировании юнит часто совпадает с целым интерфейсом (например, класс) или отдельным методом (https://en.wikipedia.org/wiki/Unit_testing)¹.

То, для чего вы пишете тесты, называется *тестируемой системой* (SUT, Subject/ System/Suite Under Test).

ОПРЕДЕЛЕНИЕ SUT может иметь разный смысл, и некоторые разработчики любят использовать сокращение CUT (тестируемый компонент/класс/код — Component/ Class/Code Under Test). Когда вы что-то тестируете, то это «что-то» обозначается SUT.

Поговорим о слове «юнит» в юнит-тестировании. Для меня *юнит* (единица) означает «единицу работы» в системе. У единицы работы имеются начало и конец, которые я буду называть *точкой входа* и *точкой выхода*. Простой пример единицы работы — функция, которая что-то вычисляет и возвращает значение. Тем не менее в процессе вычисления функция также может использовать другие функции, другие модули и другие компоненты, что означает, что единица

¹ На момент работы над русским изданием книги определение юнит-тестирования в Википедии (доступное по приведенной ссылке) несколько изменилось. — *Примеч. ред.*

работы (от точки входа до точки выхода) может охватывать код, выходящий за пределы одной функции.

Единица работы

«Единица работы» состоит из всех действий, происходящих между вызовом точки входа и получением наблюдаемого конечного результата на одной или нескольких точках выхода. Например, для общедоступной (public) функции:

- тело функции образует всю единицу работы или ее часть;
- объявление и сигнатура функции являются точкой входа в тело;
- выходные значения или аспекты поведения функции являются ее точками выхода.

1.3. ТОЧКИ ВХОДА И ТОЧКИ ВЫХОДА

Единица работы всегда имеет одну точку входа и одну или несколько точек выхода. На рис. 1.1 изображена простая диаграмма с единицей работы.

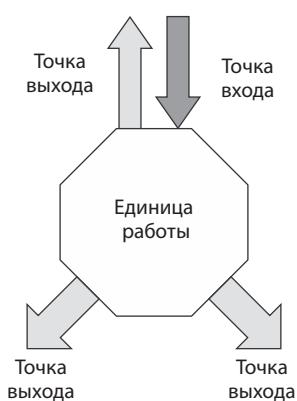
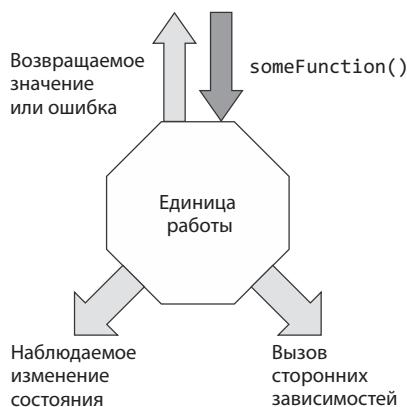


Рис. 1.1. Единица работы имеет точку входа и точки выхода

Единица работы может быть отдельной функцией, несколькими функциями или даже несколькими модулями либо компонентами. Но она всегда имеет точку входа, которая может активироваться извне (через тесты или другой рабочий код) и всегда в конечном счете делает нечто полезное. Если бы она не делала ничего полезного, то ее можно было бы просто удалить из кодовой базы.

Что считается *полезным*? Нечто происходящее в коде, что может наблюдаться другими сторонами: возвращаемое значение, изменение состояния или внешний вызов, как показано на рис. 1.2. Такое наблюдаемое поведение я называю *точками выхода*.

**Рис. 1.2.** Типы точек выхода

Почему «точка выхода»?

Почему я использую термин «точка выхода», а не что-нибудь вроде «поведения»? В моем представлении поведение может быть чисто внутренним, тогда как нас интересуют аспекты поведения, видимые извне. Различия может быть трудно понять сразу. Кроме того, «точка выхода» предполагает выход из контекста единицы работы и возврат в контекст теста, хотя поведения могут быть более гибкими. Обширное обсуждение разновидностей поведения, включая наблюдаемое поведение, приведено в книге Владимира Хорикова «Unit Testing Principles, Practices, and Patterns»¹ (Manning, 2020). Обращайтесь к ней за дополнительной информацией по этой теме.

В листинге 1.1 приведен краткий пример простой единицы работы.

Листинг 1.1. Простая функция, которую нужно протестировать

```
const sum = (numbers) => {
  const [a, b] = numbers.split(',');
  const result = parseInt(a) + parseInt(b);
  return result;
};
```

¹ Хориков В. Принципы юнит-тестирования. СПб.: Питер.

О версии Javascript, используемой в книге

Я решил использовать Node.js 12.8 с простым ES6 JavaScript и комментариями в стиле JSDoc. Для простоты я буду применять систему модулей CommonJS. Возможно, в будущем издании я обращусь к модулям ES (файлы `.mjs`), но пока (и для большей части книги) CommonJS будет достаточно. Для паттернов, представленных в книге, это на самом деле не так уж важно.

Вы сможете легко экстраполировать описанные приемы для любого стека JavaScript, с которым вы в настоящее время работаете, будь то TypeScript, базовый JS, модули ES, бэкенд или фронтенд, Angular или React. Это не должно ни на что влиять.

Где загрузить код для этой главы

Все примеры кода, приведенные в книге, можно загрузить с GitHub. Репозиторий находится по адресу <https://github.com/royosherove/aout3-samples>. Убедитесь в том, что у вас установлена версия Node 12.8 и выше, и выполните команду `npm install`, за которой следует `npm run ch[номер главы]`. Например, для данной главы это будет команда `npm run ch1`. Команда выполняет все тесты для этой главы, чтобы вы смогли просмотреть их результаты.

Эта единица работы полностью ограничена одной функцией. Функция является точкой входа, а поскольку ее конечный результат возвращает значение, она также служит точкой выхода. Мы получаем конечный результат в том же месте, где

активируется единица работы, так что точка входа совпадает с точкой выхода.

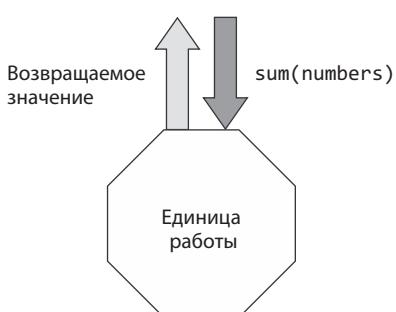


Рис. 1.3. Функция, у которой точка входа совпадает с точкой выхода

Если изобразить эту функцию как единицу работы, она будет выглядеть примерно так, как показано на рис. 1.3. Я использовал в качестве точки входа `sum(numbers)`, а не `numbers`, потому что точкой входа является сигнатура функции. Параметры образуют контекст (или ввод), передаваемый через точку входа.

В листинге 1.2 представлена вариация на эту тему.

Листинг 1.2. Единица работы с точками входа и выхода

```
let total = 0;

const totalSoFar = () => {
  return total;
};

const sum = (numbers) => {
  const [a, b] = numbers.split(',');
  const result = parseInt(a) + parseInt(b);
  total += result; ← Новая функциональность:
  return result;    | вычисление накапливаемой суммы
};
```

Новая версия `sum` имеет *две* точки выхода. Она решает две задачи:

- возвращает значение;
- вводит новую функциональность: накапливаемую сумму всех частичных сумм. Состояние модуля задается таким образом, что оно становится наблюдаемым (через `totalSoFar`) для стороны, вызывающей точку входа.

На рис. 1.4 показано, как бы я изобразил эту единицу работы. Можно рассматривать две точки выхода как два разных пути (требования) из одной единицы работы, потому что в действительности они представляют две разные полезные операции, которые должны выполняться кодом. Это также означает, что я с большой вероятностью написал бы здесь два разных юнит-теста: по одному для каждой точки выхода. Вскоре именно это мы и сделаем.

Как насчет переменной `totalSoFar`? Является ли она отдельной точкой входа? Да, может быть, в *отдельном тесте*. Я могу написать тест, который докажет, что вызов `totalSoFar` без активации до этого вызова вернет 0. Тогда она станет отдельной маленькой единицей работы, что будет вполне нормально. Часто одна единица работы (такая, как `sum`) может состоять из меньших единиц.

Как видите, область видимости тестов может изменяться, но мы все равно можем определять их точками входа и точками выхода. Точки входа всегда находятся там, где тест активирует единицу работы. Единица работы может иметь несколько точек входа, каждая из которых используется отдельным набором тестов.

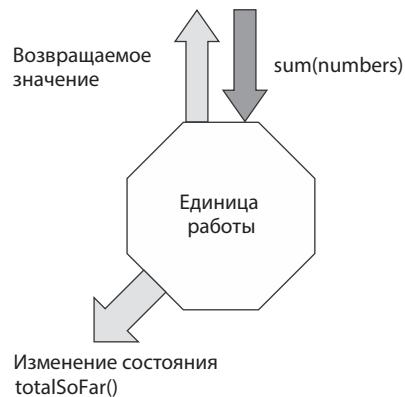


Рис. 1.4. Единица работы с двумя точками выхода

О проектировании

Существуют два основных типа действий: запросы и команды. Действия-запросы ничего не изменяют; они только возвращают значения. Действия-команды что-то изменяют, но не возвращают значения.

Часто эти два типа объединяются, но во многих случаях их разделение может быть более эффективным проектировочным выбором. Проектирование не является основной темой этой книги, но я рекомендую дополнительно ознакомиться с концепцией *разделения команд и запросов* на веб-сайте Мартина Фаулера (Martin Fowler): <https://martinfowler.com/bliki/CommandQuerySeparation.html>.

Точки выхода как обозначения требований и новые тесты

Точки выхода являются конечными результатами единицы работы. Для юнит-тестов я обычно пишу один тест со своим читабельным именем для каждой точки выхода. Позднее я могу добавить дополнительные тесты с различными вариациями ввода, использующими одну и ту же точку входа, для большей уверенности.

Интеграционные тесты, которые будут рассмотрены далее в этой главе и в книге, обычно включают несколько конечных результатов, потому что разделить кодовые пути на этих уровнях не всегда возможно. Кроме того, это одна из причин, по которым интеграционные тесты сложнее в отладке, запуске и сопровождении. Как вы вскоре увидите, они делают намного больше, чем юнит-тесты.

Третья версия функции из нашего примера приведена в листинге 1.3.

Листинг 1.3. Добавление вызова logger в функцию

```
let total = 0;

const totalSoFar = () => {
    return total;
};

const logger = makeLogger();

const sum = (numbers) => {
    const [a, b] = numbers.split(',');
    logger(`Adding ${a} + ${b}`);
    return a + b;
};
```

```

logger.info(
    'this is a very important log output',
    { firstNumWas: a, secondNumWas: b });

const result = parseInt(a) + parseInt(b);
total += result;
return result;
};

```

Новая точка выхода

Как видите, в функции появилась новая точка выхода (или требование, или конечный результат). Она выводит что-то во внешнюю сущность — возможно, в файл, или на консоль, или в базу данных. Мы точно не знаем, и нас это не интересует. Это третий тип точки выхода: *вызов третьей стороны*. Я обычно называю это *вызовом зависимости*.

ОПРЕДЕЛЕНИЕ Зависимостью называется то, что не находится под вашим полным контролем во время юнит-тестирования (или, например, попытки контролировать это в ходе теста слишком сильно усложняют жизнь). Примеры такого рода — логгеры, осуществляющие запись в файлы, подсистемы взаимодействия с сетью, код под контролем других команд, компоненты, выполнение которых занимает много времени (вычисления, потоки, обращения к базам данных) и т. д. На практике можно руководствоваться следующим критерием: если вы можете легко и в полной мере контролировать то, что делает сущность, если она выполняется в памяти и это происходит быстро, то это не зависимость. У правил всегда находятся исключения, но это правило работает по крайней мере в 80 % случаев.

На рис. 1.5 показано, как бы я изобразил эту единицу работы со всеми тремя точками выхода. На данный момент мы все еще обсуждаем единицу работы масштаба функции. Точной входа является вызов функции, но теперь у нас имеются три возможных пути (точки выхода), которые делают что-то полезное и могут быть открыто проверены стороной вызова.

И здесь все становится действительно интересным: *желательно иметь отдельный тест для каждой точки выхода*. Тесты становятся более читабельными и простыми в отладке или изменении, и при этом они никак не влияют на другие результаты.



Рис. 1.5. Представление трех точек выхода из функции

1.4. ТИПЫ ТОЧЕК ВЫХОДА

Ранее были представлены три разных типа конечных результатов.

- В вызванной функции возвращается полезное значение (не `undefined`). Если бы мы использовали язык со статической типизацией — такой, как Java или C#, — мы бы сказали, что это общедоступная функция с возвращаемым значением, отличным от `void`.
- Существуют *наблюдаемые* изменения состояния или поведения системы до и после вызова, которые могут быть определены без обращения к приватному состоянию.
- Присутствует вызов сторонней системы, которая не находится под контролем теста. Сторонняя система не возвращает никакого значения, или это значение игнорируется. (Пример: код вызывает стороннюю систему регистрации данных, которая не была написана вами, и вы не контролируете ее исходный код.).

Определение точек входа и выхода, взятое из книги «XUnit Test Patterns»

В книге Джерарда Месароша (Gerard Meszaros) «XUnit Test Patterns»¹ (Addison-Wesley Professional, 2007) обсуждается понятие *прямых и непрямых вводов и выводов*. *Прямой ввод* — то, что я предполагаю называть «точками входа». Месарош называет их «использованием парадного входа» компонента. К *непрямым выводам* в этой книге относятся два других типа точек выхода, упоминавшихся ранее (изменение состояния и вызов третьей стороны).

Обе версии этих идей развивались параллельно, но идея «единицы работы» встречается только в этой книге. Единица работы в сочетании с точками входа и выхода кажется мне гораздо более логичной, чем прямые и непрямые вводы и выводы; впрочем, это можно считать стилистическим решением относительно того, как следует преподавать концепции областей действия тестов. За дополнительной информацией о «XUnit Test Patterns» обращайтесь по адресу xunitpatterns.com.

Как же идея точек входа и выхода влияет на определение юнит-теста? *Юнит-тест* — фрагмент кода, который активирует единицу работы и проверяет одну конкретную точку выхода как конечный результат этой единицы работы. Если предположения относительно конечного результата окажутся неправильными, значит, юнит-тест не прошел. Область действия юнит-теста

¹ Месарош Д. Шаблоны тестирования xUnit: рефакторинг кода тестов.

может простираться от одной функции до нескольких модулей или компонентов в зависимости от того, сколько функций и модулей используется между точкой входа и точкой выхода.

1.5. РАЗНЫЕ ТОЧКИ ВЫХОДА, РАЗНЫЕ МЕТОДЫ

Почему я трачу столько времени на обсуждение разновидностей точек выхода? Не только из-за того, что для каждой точки выхода желательно создать отдельный тест, но и из-за того, что разным типам точек выхода для успешного тестирования могут потребоваться разные методы.

- Точки выхода на основе возвращаемых значений («прямые выводы» в книге Месароша «JUnit Test Patterns») тестируются проще других точек выхода. Вы активизируете точку входа, получаете что-то обратно и проверяете полученное значение.
- Тесты на основе состояния («непрямые выводы») обычно получаются чуть более хитроумными. Вы что-то вызываете, а затем выдаете другой вызов для проверки чего-то другого (или снова вызываете предыдущее «что-то»), чтобы узнать, все ли прошло по плану.

В ситуации со сторонними вызовами (непрямые выводы) приходится идти на еще большие ухищрения. Эта тема пока не обсуждалась, но именно там приходится использовать *моки* (mock objects) для замены внешней системы чем-то таким, что мы можем контролировать и анализировать в своих тестах. Эта идея будет более подробно рассмотрена далее в книге.

Какие точки выхода создают больше всего проблем?

Как правило, я стараюсь использовать тесты на основе возвращаемых значений и состояния. Если это возможно, я стараюсь избегать тестов на основе моков, и обычно мне это удается. В результате обычно не более 5 % моих тестов использует моки для проверки. Эти разновидности тестов усложняют ситуацию и затрудняют сопровождение кода. Впрочем, иногда выбирать не приходится, и мы обсудим их по мере изложения материала в следующих главах.

1.6. НАПИСАНИЕ ТЕСТА С НУЛЯ

Вернемся к первой, самой простой версии кода (листинг 1.1) и попробуем протестировать ее. Если бы вам предложили написать для нее тест, как бы он выглядел?

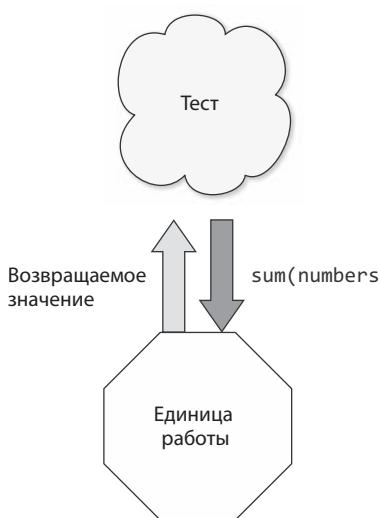


Рис. 1.6. Наглядное представление нашего теста

Начнем с наглядного представления на рис. 1.6. Точкой входа является функция `sum` со входной строкой `numbers`; `sum` также является точкой выхода, так как мы получаем от нее возвращаемое значение и проверяем его.

Автоматизированный юнит-тест можно написать без использования тестового фреймворка. На самом деле все больше разработчиков используют автоматизацию своих тестов, и я видел, как многие из них делали это еще до знакомства с тестовыми фреймворками. В этом разделе мы напишем такой тест без фреймворка, чтобы его можно было сравнить с решением, использующим фреймворк, в главе 2.

Итак, допустим, что тестовых фреймворков нет (или мы не знаем об их существовании).

Мы решили написать собственный маленький автоматизированный тест с нуля. В листинге 1.4 приведен очень наивный пример тестирования нашего кода на базовом JavaScript.

Листинг 1.4. Очень наивный тест для `sum()`

```
const parserTest = () => {
  try {
    const result = sum('1,2');
    if (result === 3) {
      console.log(`parserTest example 1 PASSED`);
    } else {
      throw new Error(`parserTest: expected 3 but was ${result}`);
    }
  } catch (e) {
    console.error(e.stack);
  }
};
```

Нет, красивым этот код никак не назовешь. Но он достаточно хорош для пояснения того, как работают тесты. Чтобы выполнить этот код, необходимо сделать следующее.

1. Откройте командную строку и введите пустую строку.
2. Добавьте в раздел "scripts" в файле `package.json` запись с именем "test", чтобы выполнить "`node mytest.js`", а затем выполните команду `npm test` в командной строке.

В листинге 1.5 показано, как это выглядит.

Листинг 1.5. Начало файла package.json

```
{
  "name": "aout3-samples",
  "version": "1.0.0",
  "description": "Code Samples for Art of Unit Testing 3rd Edition",
  "main": "index.js",
  "scripts": {
    "test": "node ./ch1-basics/custom-test-phase1.js",
  }
}
```

Тестовый метод активирует *тестируемую систему* (SUT), после чего проверяет возвращенное значение. Если оно отличается от ожидаемого, тестовый метод выдает на консоль ошибку и трассировку стека. Тестовый метод также перехватывает любые возникающие исключения и выводит их на консоль, чтобы они не мешали выполнению последующих методов. При использовании тестового фреймворка это обычно делается за нас автоматически.

Очевидно, такой подход к написанию тестов довольно бессистемный. Если вам потребуется написать несколько таких тестов, возможно, стоит создать обобщенный метод тестирования или проверки, который будет использоваться всеми тестами и будет последовательно форматировать выводимые ошибки. Также можно добавить специальные вспомогательные методы для проверки таких условий, как `null`-объекты, пустые строки и т. д., чтобы вам не приходилось включать одни и те же длинные строки кода в разные тесты.

В листинге 1.6 показано, как будет выглядеть этот тест с чуть более общими функциями `check` и `assertEquals`.

Листинг 1.6. Использование более общей реализации метода check

```
const assertEquals = (expected, actual) => {
  if (actual !== expected) {
    throw new Error(`Expected ${expected} but was ${actual}`);
  }
};

const check = (name, implementation) => {
  try {
    implementation();
    console.log(`${name} passed`);
  } catch (e) {
    console.error(`${name} FAILED`, e.stack);
  }
};

check('sum with 2 numbers should sum them up', () => {
```

```
const result = sum('1,2');
assertEquals(3, result);
});

check('sum with multiple digit numbers should sum them up', () => {
  const result = sum('10,20');
  assertEquals(30, result);
});
```

Мы создали два вспомогательных метода: `assertEquals`, который устраниет шаблонный код для вывода на консоль и выдачи ошибок, и `check`, который получает строку с именем теста и обратным вызовом реализации. Затем он перехватывает любые ошибки в teste, выводит их на консоль и сообщает о статусе teste.

Встроенные утверждения

Важно заметить, что писать собственные утверждения (`asserts`) необязательно. С таким же успехом можно воспользоваться встроенными проверочными функциями Node.js, которые изначально были созданы для внутреннего использования при тестировании самого Node.js. Для этого следует импортировать функции командой

```
const assert = require('assert');
```

Тем не менее я пытаюсь продемонстрировать простоту концепции, поэтому мы обойдемся без этих функций. Дополнительную информацию о модуле `assert` в Node.js можно найти по адресу <https://nodejs.org/api/assert.html>.

Обратите внимание, насколько проще читаются и быстрее пишутся teste с добавлением пары вспомогательных методов. Фреймворки юнит-тестирования (такие, как Jest) могут предоставить еще более универсальные методы для помощи в написании testeов. Мы вернемся к этой теме в главе 2, но сначала немного поговорим о главной теме этой книги: хороших юнит-тестах.

1.7. ХАРАКТЕРИСТИКИ ХОРОШЕГО ЮНИТ-ТЕСТА

Какой бы язык программирования вы ни использовали, один из самых сложных вопросов — это определение того, какой же юнит-тест следует считать *хорошим*. Конечно, само понятие «хороший» относительно, и оно может изменяться каждый раз, когда вы узнаете что-то новое о программировании. Это может показаться очевидным, но на самом деле это не так. Я должен объяснить, *почему* нужно писать хорошие testeы, — понимать, что такое «единица работы», недостаточно.

По моему многолетнему опыту работы в разных компаниях и командах, большинство людей, пытающихся организовать юнит-тестирование своего кода, либо сдаются в какой-то момент, либо просто не выполняют юнит-тесты. Эти люди либо тратят массу времени на написание проблемных тестов, а потом бросают это дело, когда приходится уделять еще больше времени на их сопровождение, либо, что еще хуже, не доверяют их результатам.

Нет смысла писать плохие юнит-тесты, если только это не часть процесса по обучению того, как писать хорошие. От написания плохих тестов больше вреда, чем пользы: вам придется тратить время на отладку ошибочных тестов, написание тестов, от которых нет никакой пользы, на попытки разобраться в непонятных тестах — и все это только для того, чтобы удалить их через несколько месяцев. Также возникают огромные проблемы с сопровождением плохих тестов и их влиянием на сопровождение рабочего кода. Плохие тесты могут замедлить разработку при написании не только кода тестов, но и рабочего кода. Все эти темы будут затронуты далее в книге.

Если вы знаете, какой юнит-тест можно считать хорошим, вы не встанете на путь, на котором потом будет трудно что-то исправить, когда код превратится в настоящий кошмар. Также в следующих главах будут определены другие разновидности тестов (компонентные, сквозные и другие).

1.7.1. Как выглядит хороший юнит-тест?

Каждый хороший автоматизированный тест (не только юнит-тесты) должен обладать следующими свойствами.

- Намерения автора теста должны быть понятными.
- Тест должен быть простым в чтении и написании.
- Он должен быть автоматизированным.
- Его результаты должны быть стабильными (он всегда должен возвращать один и тот же результат, если между выполнениями ничего не изменилось).
- Он должен приносить пользу и предоставлять результаты, имеющие практическую ценность.
- Он должен выполняться простым нажатием кнопки.
- Если тест не прошел, читатель кода должен легко определить, какой результат ожидался и как найти причину проблемы.

Хороший юнит-тест дополнительно должен обладать следующими свойствами.

- Он должен быстро выполняться.
- Он должен *полностью контролировать* тестируемый код (подробнее об этом в главе 3).

- Он должен быть полностью изолированным (то есть выполняться независимо от других тестов).
- Он должен выполняться в памяти, не требуя обращений к системным файлам, сетям или базам данных.
- Он должен быть настолько синхронным и линейным, насколько это возможно, когда это имеет смысл (без параллельных программных потоков, если это возможно).

Все тесты не могут обладать характеристиками хороших юнит-тестов, и это нормально. Такие тесты просто переходят в категорию интеграционного тестирования (тема раздела 1.8). Тем не менее некоторые тесты могут быть доработаны и приобретут требуемые свойства.

Замена базы данных (или другой зависимости) стабом

Стабы (stubs) будут рассмотрены далее, но сейчас вкратце скажем, что они представляют собой фиктивные зависимости, которые эмулируют реальные. Они создаются для упрощения процесса тестирования, потому что они проще в настройке и сопровождении.

Следует избегать использования баз данных в памяти при тестировании. Они могут способствовать изоляции тестов друг от друга (при условии, что экземпляры БД не используются совместно разными тестами), что соответствует свойствам хороших юнит-тестов, но такие базы данных приводят к созданию неуклюжих промежуточных решений. Базы данных в памяти не так просто настраиваются, как стабы. В то же время они не предоставляют таких сильных гарантий, как реальные БД. С точки зрения функциональности БД в памяти может кардинально отличаться от рабочей, так что тесты, проходящие с БД в памяти, могут не проходить с реальной, и наоборот. Часто приходится повторно выполнять те же тесты вручную с рабочей БД, чтобы лишний раз удостовериться в работоспособности кода. Если только вы не используете небольшой стандартизированный набор средств SQL, я рекомендую придерживаться либо стабов (для юнит-тестов), либо реальных баз данных (для интеграционного тестирования).

Сказанное относится и к таким средствам, как `jsdom`. Вы можете использовать их для замены реальной DOM, но убедитесь в том, что они поддерживают ваши конкретные сценарии использования. Не пишите тесты, которые приходится перепроверять вручную.

Эмуляция асинхронной обработки в линейных синхронных тестах

С пришествием промисов (`promises`) и `async/await` асинхронное программирование стало стандартным явлением в JavaScript. Тем не менее наши тесты могут синхронно проверять асинхронный код. Обычно это подразумевает активацию

обратных вызовов прямо из теста или явное ожидание завершения асинхронной операции.

1.7.2. Чек-лист юнит-теста

Многие люди путают акт тестирования своего кода с концепцией юнит-тестов. Для начала задайте себе ряд вопросов о тестах, которые вы написали и выполнили до настоящего момента.

- Смогу ли я запустить тест, написанный две недели (или два месяца, или два года) назад, и получить результаты?
- Сможет ли любой участник моей команды запустить и получить результаты тестов, написанных мной два месяца назад?
- Смогу ли я выполнить все тесты, написанные мной, не более чем за несколько минут?
- Смогу ли я запустить все тесты, написанные мной, нажатием одной кнопки?
- Смогу ли я написать базовый тест за несколько минут?
- Проходят ли мои тесты, когда в коде другой команды присутствуют баги?
- Показывают ли мои тесты те же результаты при запуске на других машинах или в других средах?
- Перестают ли мои тесты работать при недоступности базы данных, сети или среды развертывания?
- Если я удаляю, перемещаю или изменяю один тест, это никак не влияет на другие тесты?

Если вы ответили отрицательно на какие-либо из этих вопросов, существует высокая вероятность, что реализуемое вами решение либо не полностью автоматизировано, либо не является юнит-тестом. Это определенно некоторая разновидность теста, и она может быть не менее важной, чем юнит-тест, но у нее есть недостатки по сравнению с тестами, которые позволяют ответить «да» на все эти вопросы.

«Тогда чем же я занимался до сих пор?» — спросите вы. Вы занимались интеграционным тестированием.

1.8. ИНТЕГРАЦИОННЫЕ ТЕСТЫ

Я отношу к категории *интеграционных тестов* любые тесты, не соответствующие одному или нескольким критериям хороших юнит-тестов, упоминавшимся выше. Так, если тест использует реальную сеть, реальные REST API, реальное системное время, реальную файловую систему или реальную базу данных, он заходит в область интеграционного тестирования.

Например, если тест не контролирует системное время и в коде теста используется текущее значение `new Date()`, то разные выполнения теста фактически будут разными тестами, потому что в них используется разное время. Тест перестает быть стабильным. Это неплохо само по себе. Я считаю интеграционные тесты важными дополнениями юнит-тестов, но они должны быть отделены от последних для формирования ощущения «безопасной зеленой зоны», обсуждаемой далее в этой книге.

Если тест использует реальную базу данных, то он уже не выполняется только в памяти — его действия труднее удалить, чем при использовании только фиктивных данных в памяти. Тест также будет дольше выполняться, и мы не сможем управлять тем, сколько времени займет обращение к данным. Юнит-тесты должны быть *быстрыми*. Когда количество тестов достигает нескольких сотен, каждая секунда имеет значение.

Интеграционные тесты повышают риск возникновения другой проблемы: тестирования многих аспектов одновременно. Допустим, у вас сломалась машина. Как вы узнаете, в чем причина проблем, не говоря уже об исправлении? Двигатель состоит из множества подсистем, работающих совместно, и каждая из них зависит от других для достижения конечного результата: движущейся машины. Если она не едет, виновником может быть любая из подсистем — и даже не одна. Именно интеграция этих подсистем (или уровней) заставляет машину перемещаться. Можно сказать, что езда автомобиля по дороге является самым важным интеграционным тестом этих частей. Если тест не проходит, значит, все части могут оказаться неисправными; если он проходит, то и все части работают успешно.

То же самое происходит с программным кодом. Большинство разработчиков тестируют свою функциональность через итоговую функциональность приложения, REST API или UI. Щелчок на кнопке запускает серию событий: функции, модули и компоненты работают вместе для получения конечного результата. Если тест не проходит, то все эти компоненты не проходят как единое целое, и может быть достаточно трудно определить, что именно привело к сбою всей операции (рис. 1.7).

В соответствии с определением из книги «The Complete Guide to Software Testing» Билла Хетцеля (Bill Hetzel) (Wiley, 1988), интеграционное тестирование представляет собой «организованную процедуру тестирования, при которой программные и/или аппаратные элементы объединяются и тестируются вплоть до интеграции всей системы». Приведу мою собственную формулировку определения интеграционного тестирования.

Интеграционное тестирование — это тестирование единицы работы без полного контроля над всеми ее реальными зависимостями: компонентами других команд, другими сервисами, временем, сетью, базами данных, программными потоками, генераторами случайных чисел и т. д.

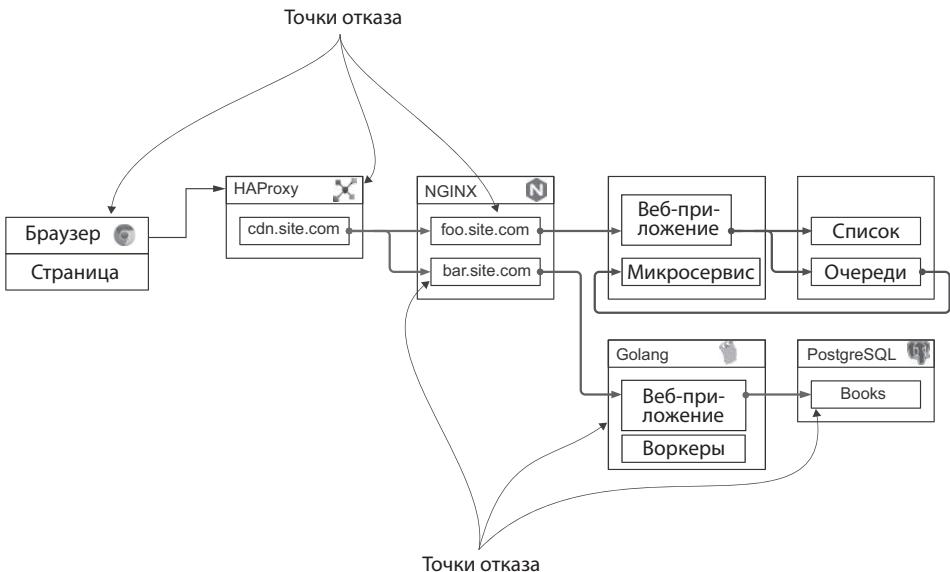


Рис. 1.7. Интеграционный тест может содержать много точек отказа. Все юниты должны работать совместно, и в каждом из них может произойти сбой, что усложняет поиск источника ошибки

Подведем итог: в интеграционном тестировании используются реальные зависимости; юнит-тесты изолируют единицу работы от ее зависимостей, чтобы обеспечить простую стабильность их результатов, а также чтобы легко контролировать и моделировать любой аспект поведения этой единицы.

Давайте вспомним вопросы из раздела 1.7.2 и подумаем, чего мы хотим добиться с реальными юнит-тестами.

- *Смогу ли я запустить тест, написанный две недели (два месяца, два года) назад, и получить результаты?*

Если не сможете, то как вы узнаете, не нарушили ли вы работоспособность некоторой функции, реализованной ранее? Общие данные и код регулярно изменяются на протяжении жизни приложения, если вы не можете (или не будете) запускать тесты для всех ранее работавших функциональностей после изменения кода, то можете нарушить их работоспособность, не подозревая об этом, — это называется *регрессией*. Регрессии часто встречаются в конце спринта или накануне выпуска, когда разработчики вынуждены исправлять существующие ошибки. Иногда они ненамеренно вносят новые ошибки при исправлении старых. Разве не замечательно было бы узнать о том, что вы что-то сломали, в течение минуты после поломки? Позднее в этой книге вы увидите, как это делается.

ОПРЕДЕЛЕНИЕ Регрессия представляет собой нарушенную функциональность — код, который ранее работал. Ее также можно рассматривать как одну или несколько единиц работы, которые ранее работали, а потом перестали.

- *Сможет ли любой участник моей команды запустить и получить результаты тестов, написанных мной два месяца назад?*

Это отчасти перекликается с предыдущим пунктом, но поднимает его на новый уровень. Когда вы что-то изменяете, необходимо убедиться в том, что вы не нарушили работоспособность чужого кода. Многие разработчики боятся изменять унаследованный код в старых системах, опасаясь за другой код, который может зависеть от того, что они изменяют. По сути, они рисуют перевести систему в состояние с неизвестной стабильностью.

Мало что вызывает столько опасений, как отсутствие уверенности в том, что приложение продолжает работать должным образом, особенно если код написан не вами. Если у вас имеется «подушка безопасности» в виде юнит-тестов и вы знаете, что ничего не сломали, вам уже будет не так страшно браться за код, с которым вы не очень хорошо знакомы.

Хорошие тесты могут быть запущены кем угодно.

ОПРЕДЕЛЕНИЕ Унаследованный код (legacy code) определяется в Википедии как «старый исходный код, который перестал поддерживаться стандартным оборудованием и средами» (https://en.wikipedia.org/wiki/Legacy_system), но во многих организациях «унаследованным кодом» называется любая старая версия приложения, которая продолжает поддерживаться в настоящее время. Часто так называют код, с которым трудно работать, который трудно тестировать и обычно даже трудно читать. Один клиент однажды определил унаследованный код с практической точки зрения: «код, который работает». Многие люди предпочитают определять унаследованный код как «код без тестов». В книге «Working Effectively with Legacy Code»¹ Майкла Физерса (Michael Feathers) (Pearson, 2004) фраза «код без тестов» используется как официальное определение унаследованного кода, и этим определением нужно руководствоваться во время чтения книги.

- *Смогу ли я выполнить все тесты, написанные мной, не более чем за несколько минут?*

Если вы не можете быстро запустить тесты (лучше за секунды, чем за минуты), вы будете реже запускать их: раз в день и даже раз в неделю, а то и в месяц. Проблема в том, что при изменении кода необходимо как можно раньше получить обратную связь, чтобы узнать, не было ли что-то сломано. Чем больше времени проходит между запусками тестов, тем больше изменений вы вносите в систему и тем больше мест вам придется просмотреть в поисках ошибок, когда вы обнаружите, что что-то сломалось.

¹ Физерс Майкл К. Эффективная работа с унаследованным кодом.

Хорошие тесты должны выполняться *быстро*.

- *Смогу ли я запустить все тесты, написанные мной, нажатием одной кнопки?*

Если не можете, вероятно, это означает, что вам нужно настроить машину, на которой выполняются тесты, чтобы они выполнялись правильно (например, настроить среду Docker или задать строки подключения к базе данных), или же ваши юнит-тесты не были полностью автоматизированы. Если вы не можете полностью автоматизировать свои юнит-тесты, вероятно, вы будете избегать их повторного выполнения, как и все остальные участники вашей команды.

Никому не захочется мучиться с настройкой всех мелочей для выполнения тестов, просто чтобы убедиться в том, что система все еще работает. У разработчиков есть более важные дела, например реализация новой функциональности в системе. Но они не смогут заняться этим, если не знают состояния системы.

Хорошие тесты должны легко выполняться в своей исходной форме, а не вручную.

- *Смогу ли я написать базовый тест за несколько минут?*

Один из самых простых признаков интеграционных тестов — затраты времени на их подготовку и реализацию, а не только на выполнение. Разработчику требуется время, чтобы понять, как писать эти тесты, из-за множества внутренних (а иногда и внешних) зависимостей. (База данных может считаться внешней зависимостью.) Если не автоматизировать тест, зависимости создают меньше проблем, но вы лишаетесь всех преимуществ автотеста. Чем сложнее писать тесты, тем вероятнее, что вы напишете их меньше или сосредоточитесь на «больших проблемах», которые касаются всех. Одна из сильных сторон юнит-тестов заключается в том, что они часто находят разные мелочи, а не только «большие проблемы». Люди часто удивляются, сколько багов им удается обнаружить в коде, который казался таким простым и безошибочным.

Когда ваше внимание сосредоточено только на больших тестах, общая надежность вашего кода оставляет желать лучшего. Многие части базовой логики кода не тестируются (даже притом что ваши тесты покрывают большее количество компонентов), и в них могут присутствовать баги, которые вы не учли и которые могут породить «неофициальные» проблемы.

Хорошие тесты должны писаться легко и быстро, потому что вы уже вычислили паттерны, которые должны использоваться для тестирования конкретного набора объектов, функций и зависимостей (*модель предметной области*).

- *Проходят ли мои тесты, когда в коде другой команды присутствуют баги? Показывают ли мои тесты те же результаты при запуске на других машинах или в других средах? Перестают ли мои тесты работать при недоступности базы данных, сети или среды развертывания?*

Эти три пункта связаны с идеей о том, что тестовый код изолируется от различных зависимостей. Результаты тестов стабильны, потому что мы полностью контролируем непрямой ввод, предоставляемый нашей системе. Можно создавать фиктивные базы данных, фиктивные сети, фиктивные показания времени и фиктивную машинную культуру. В последующих главах я буду называть такие точки *стабами* (stubs).

- *Если я удаляю, перемещаю или изменяю один тест, это никак не влияет на другие тесты?*

Юнит-тестам обычно не нужно иметь никакое общее или разделяемое (shared) состояние, но в интеграционных тестах оно часто используется (например, внешние базы данных и сервисы). Общее состояние может создавать зависимости между тестами. Например, запуск тестов в неправильном порядке может повредить состояние для будущих тестов.

ВНИМАНИЕ! Даже опытному юнит-тестировщику может потребоваться более 30 минут для написания *самого первого* теста для модели предметной области, в которой он еще никогда не запускал тесты. Это часть работы, к которой следует относиться как к неизбежности. Второй и последующие тесты для этой предметной области уже будут создаваться очень легко, когда вы определите точки входа и выхода единицы работы.

В этих вопросах и ответах можно выделить три главных критерия.

- *Читабельность.* Если тест невозможно прочитать, его будет трудно сопровождать, трудно отлаживать и трудно понять, что же пошло не так.
- *Простота сопровождения.* Если сопровождение кода теста или рабочего кода усложняется из-за наших тестов, работа превратится в настоящий кошмар.
- *Достоверность.* Если вы не доверяете результатам тестов, которые не прошли, вы повторяете тесты вручную. При этом теряется весь выигрыш по времени, который должны обеспечить тесты. Если вы не доверяете *прошедшим* тестам, вы начинаете отладку заново, снова теряя выигрыши по времени.

Итак, я объяснил, чем не является юнит-тест и какими характеристиками он должен обладать, чтобы тестирование приносило практическую пользу. Теперь можно переходить к ответу на основной вопрос этой главы: что представляет собой хороший юнит-тест?

1.9. ОКОНЧАТЕЛЬНОЕ ОПРЕДЕЛЕНИЕ

После описания важных свойств, которыми должен обладать юнит-тест, я приведу окончательное определение:

Юнит-тест представляет собой автоматизированный фрагмент кода, который активирует единицу работы через точку входа, а затем про-

веряет одну из ее точек выхода. Юнит-тесты почти всегда создаются с использованием фреймворка юнит-тестирования. Они легко пишутся и быстро запускаются. Они достоверны, хорошо читаются и просты в сопровождении. Их результаты стабильны при условии, что рабочий код, который находится под нашим контролем, не изменился.

Конечно, такое определение выглядит весьма амбициозно, особенно если учесть, сколько разработчиков плохо справляются с реализацией юнит-тестов. Оно заставляет нас критически взглянуть на то, как мы, разработчики, реализовывали тестирование до настоящего момента по сравнению с тем, как нам хотелось бы его реализовать. (Достоверность, читабельность и сопровождаемость тестов подробно рассматриваются в главах с 7-й по 9-ю.)

В первом издании книги мое определение юнит-теста выглядело несколько иначе. Я определял юнит-тест как «выполняемый только в соответствии с кодом потока управления», но сейчас я уже не считаю это определение точным. Код без логики обычно используется как часть единицы работы. Даже свойства без логики будут использоваться как единица работы, так что они не должны становиться отдельной целью для тестов.

ОПРЕДЕЛЕНИЕ *Потоком управления* называется любой фрагмент кода, содержащий какую-либо логику. Это может быть инструкция `if`, цикл, вычисления, любой код с принятием решений или их произвольная комбинация.

Get- и set-методы (геттеры и сеттеры) — хороший пример кода, который обычно не содержит логики и, следовательно, не становится конкретной целью тестов. Это код, который может использоваться тестируемой единицей работы, но тестиировать его напрямую нет необходимости. Однако будьте внимательны: как только вы добавите в геттер или сеттер какую-либо логику, необходимо позаботиться о том, чтобы она была протестирована.

В следующем разделе мы завершим обсуждение того, что такое хороший тест, и поговорим о том, когда пишутся тесты. Речь пойдет о методологии разработки через тестирование, потому что ее часто ставят рядом с юнит-тестированием. Я хочу сразу расставить все точки над «*i*».

1.10. РАЗРАБОТКА ЧЕРЕЗ ТЕСТИРОВАНИЕ

Когда вы научитесь писать понятные, простые в сопровождении и достоверные тесты с фреймворком юнит-тестирования, необходимо ответить на следующий вопрос: *когда* писать тесты? Многие люди считают, что писать юнит-тесты лучше всего после того, как они создадут некоторую функциональность, и непосредственно перед тем, как их код будет сохранен в системе управления исходным кодом.

Кроме того, немало людей вообще считает, что писать тесты — пустая трата времени, но они методом проб и ошибок выяснили, что при код-ревью выдвигаются строгие требования к тестированию. По этой причине они *вынуждены* писать тесты, чтобы умилостивить богов рецензирования и добиться того, чтобы их код был включен в главную ветвь. (Подобная динамика — частый источник плохих тестов, и она будет рассмотрена в третьей части книги.)

Однако все больше разработчиков предпочитают писать юнит-тесты поэтапно, во время написания кода и до реализации каждой части очень небольшой функциональности. Такой подход называется *разработкой через тестирование* (TDD, Test-Driven Development).

ПРИМЕЧАНИЕ Существует много разных точек зрения на то, что же именно означает термин «разработка через тестирование». Одни считают, что это практика написания тестов до кода, а другие — что это практика написания многочисленных тестов. Третьи считают, что это методология проектирования, а четвертые — что это способ управления поведением кода с незначительной долей проектирования. В этой книге под TDD понимается практика опережающего написания тестов, при которой проектирование играет постепенно возрастающую роль (кроме этого раздела, TDD в книге обсуждаться не будет).

На рис. 1.8 и 1.9 представлены различия между традиционным программированием и TDD. TDD отличается от традиционной разработки, как видно из рис. 1.9. Вы начинаете с написания теста, который не проходит; затем вы пишете рабочий код, убеждаетесь в том, что тесты проходят, и либо продолжаете проводить рефакторинг вашего кода, либо создаете новый тест, который не проходит.

В этой книге центральное место занимает написание хороших юнит-тестов, а не TDD, но я большой поклонник TDD. Я написал несколько серьезных приложений и фреймворков с применением TDD, управлял командами, в которых эта практика применялась, и провел сотни учебных курсов и семинаров по TDD и средствам юнит-тестирования. За свою карьеру я пришел к выводу, что TDD помогает создавать качественный код, качественные тесты и более эффективные дизайны для моего кода. Я убежден, что разработка через тестирование может приносить пользу, но за нее приходится платить (время на обучение, время на реализацию и т. д.). Тем не менее «плата за вход» окупается, если вы не побоитесь трудностей при изучении TDD.

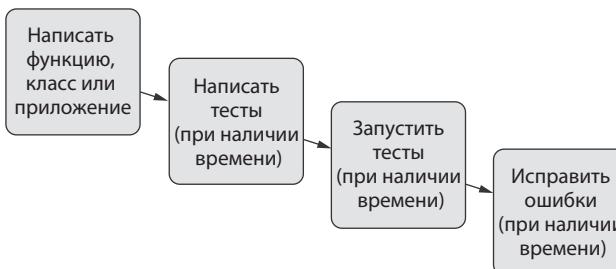


Рис. 1.8. Традиционный способ написания юнит-тестов

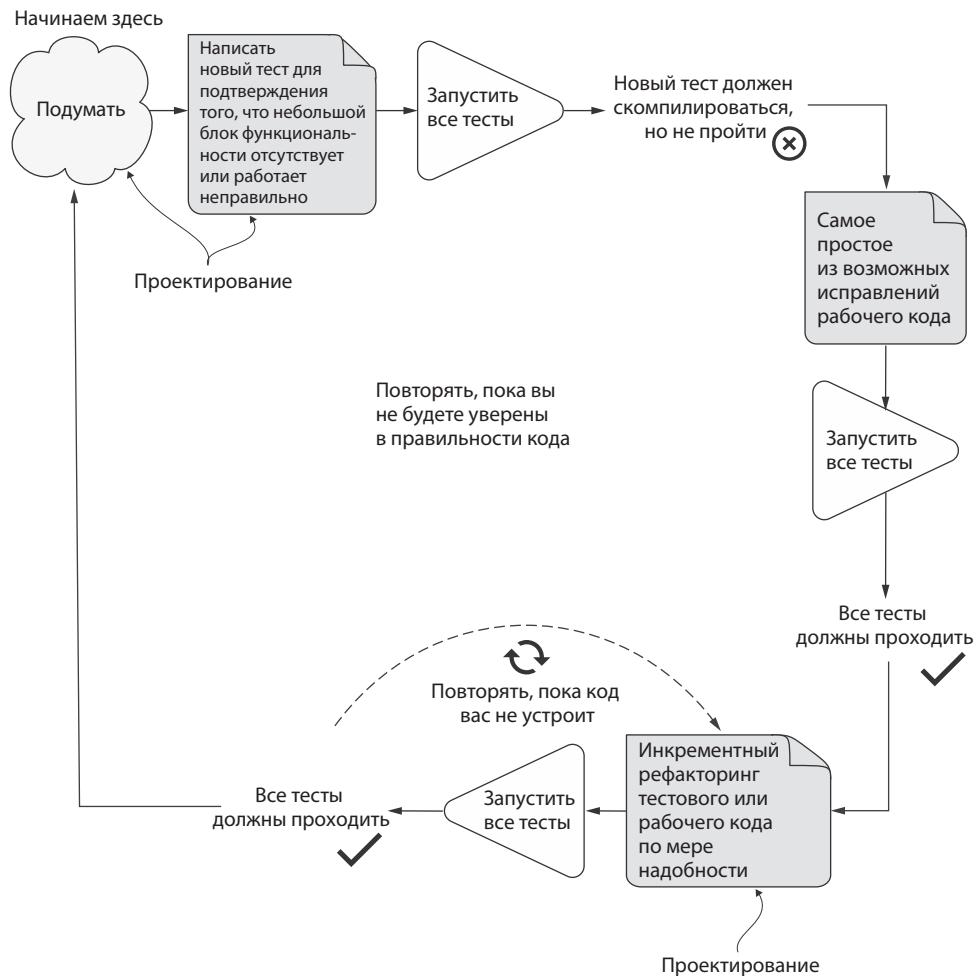


Рис. 1.9. Разработка через тестирование. Обратите внимание на циклическую природу процесса: написание теста, написание кода, рефакторинг, написание следующего теста. Она демонстрирует инкрементную природу TDD: небольшие шаги надежно приводят к качественному конечному результату

1.10.1. TDD не заменит хорошие юнит-тесты

Важно понимать, что TDD не гарантирует успеха проекта или надежности либо простоты в сопровождении тестов. Довольно легко увлечься методологией TDD и не обращать внимания на то, как пишутся юнит-тесты: на выбор имен, сопровождаемость или читабельность тестов, правильный выбор целей для тестирования или наличие багов в самих тестах. Именно поэтому я пишу эту книгу; написание хороших тестов — отдельный от TDD навык.

Методология TDD достаточно проста.

- Напишите непроходящий (провальный) тест, доказывающий отсутствие кода или функциональности в конечном продукте.* Тест пишется так, словно рабочий код уже работает, так что непройденный тест означает наличие бага в рабочем коде. Откуда я об этом знаю? Тест пишется так, что он должен проходить, если в рабочем коде нет багов.

В некоторых языках, кроме JavaScript, тест поначалу даже может не компилироваться, потому что код еще не существует. Когда тест заработает, он не должен проходить, потому что рабочий код еще не работает. Именно на эту стадию приходится большая часть проектирования в методологии TDD.

- Обеспечьте прохождение теста, добавляя в рабочий код функциональность, соответствующую ожиданиям вашего теста.* Рабочий код должен быть настолько простым, насколько это возможно. Не прикасайтесь к тесту. Вы должны обеспечить его прохождение, изменяя только рабочий код.
- Проведите рефакторинг своего кода.* Когда тест пройдет, можно переходить к следующему юнит-тесту или провести рефакторинг кода (как рабочего, так и кода тестов): чтобы он лучше читался, чтобы устранить из него дублирование и т. д. Это еще одна точка, в которой происходит обозначенное на диаграмме «проектирование». Мы проводим рефакторинг и даже изменяем структуру своих компонентов при сохранении старой функциональности.

Шаги рефакторинга должны быть очень маленькими и постепенными, и после каждого такого шага необходимо запускать все тесты, чтобы убедиться, что новые изменения ничего не нарушили. Рефакторинг может выполняться после написания нескольких тестов или после написания каждого. Эта практика важна, потому что она гарантирует, что ваш код будет проще читаться и сопровождаться, при этом проходя все ранее написанные тесты. Далее в книге рефакторингу будет посвящен целый раздел (8.3).

ОПРЕДЕЛЕНИЕ *Рефакторингом* называется изменение фрагмента кода без изменения его функциональности. Если вы когда-либо переименовывали метод, это было рефакторингом. Если вы когда-либо разбивали большой метод на вызовы нескольких меньших, это было рефакторингом. Код делает то же самое, но его становится проще сопровождать, читать, отлаживать и изменять.

Может показаться, что описанные действия имеют чисто техническую природу, но при правильном проведении TDD может резко поднять качество и улучшить структуру кода, уменьшить количество багов, повысить вашу уверенность в коде, сократить время поиска ошибок и обеспечить хорошее настроение начальства. При неправильном проведении TDD может привести к нарушению графика проекта, неэффективному расходованию времени, снижению мотивации и ухудшению качества кода. Это палка о двух концах, и многие разработчики узнают об этом на собственном горьком опыте.

Одно из величайших преимуществ TDD, о которых обычно никто не говорит, заключается в том, что, когда вы сначала видите непроходящий тест, а потом то, как он проходит без изменения теста, вы фактически тестируете сам тест. Если вы ожидаете, что тест не пройдет, а он проходит, в нем может присутствовать ошибка или вы тестируете что-то не то. Если тест не проходил, вы его исправили и теперь ожидаете, что он пройдет, а он не проходит — возможно, в тесте допущена ошибка или его ожидания неверны.

В книге рассматриваются читабельные, легко сопровождаемые и достоверные тесты. Но при добавлении TDD ваша уверенность в собственных тестах повысится: тесты изначально проваливаются, вы исправляете код и потом видите, что тесты проваливаются там, где должны проваливаться, и проходят там, где должны проходить. В стиле с написанием тестов после кода вы обычно видите только то, что тесты проходят там, где должны проходить, и проваливаются, где не должны проваливаться (так как тестируемый код уже должен работать). TDD оказывает большую помощь, и это одна из причин, по которой разработчикам, практикующим TDD, приходится тратить меньше времени на отладку, чем при простом юнит-тестировании «задним числом». Если разработчики доверяют своим тестам, они не чувствуют необходимости проводить отладку «на всякий случай». Этот уровень доверия достигается только при виде обеих сторон теста: когда он проходит там, где должен проходить, и проваливается там, где должен проваливаться.

1.10.2. Три основных навыка, необходимых для успешного применения TDD

Чтобы с успехом применять разработку через тестирование, вам понадобятся три разных набора навыков: умение писать хорошие тесты, написание тестов до рабочего кода (*test-first*) и качественное проектирование тестов и рабочего кода. На рис. 1.10 они обозначены более наглядно.

- *Даже если вы пишете свои тесты раньше, чем код, это не означает, что они будут простыми в сопровождении, читабельными или достоверными.* Хорошие навыки юнит-тестирования — основная тема нашей книги.
- *Даже если вы пишете читабельные и простые в сопровождении тесты, это не означает, что вы получите те же преимущества, как при их написании до рабочего кода.* Большинство книг по TDD учит читателя навыкам первоочередного написания тестов, не обучая их навыкам хорошего тестирования. Я бы особо порекомендовал книгу Кента Бека «Test-Driven Development: By Example»¹ (Addison-Wesley Professional, 2002).

¹ Бек К. Экстремальное программирование. Разработка через тестирование. СПб.: Питер.

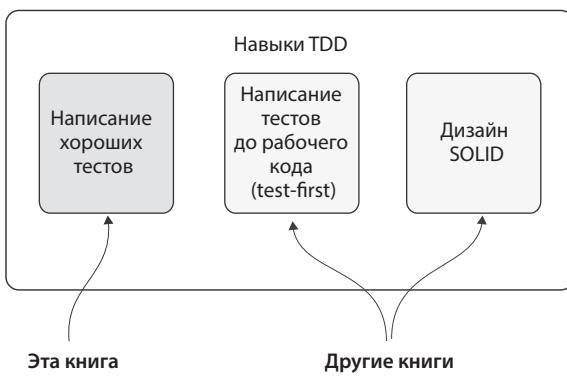


Рис. 1.10. Три основных навыка разработки через тестирование

- *Даже если вы пишете тесты раньше, чем код, и они читабельные и простые в сопровождении, это не означает, что у вас получится хорошо спроектированная система.* Мастерство проектирования — то, что делает ваш код красивым и простым в сопровождении. Если вам понадобятся хорошие книги по теме, я рекомендую «Growing Object-Oriented Software, Guided by Tests» Стива Фримена (Steve Freeman) и Нэта Прайса (Nat Pryce) (Addison-Wesley Professional, 2009) и «Clean Code»¹ Роберта К. Мартина (Robert C. Martin) (Pearson, 2008).

Прагматичный подход к TDD заключается в том, чтобы изучать каждый из трех аспектов по отдельности, то есть в любой момент времени сосредоточиться на одном навыке, временно забыв о других. Я рекомендую этот подход, потому что часто вижу, как люди пытаются освоить все три навыка одновременно, им приходится нелегко, и в итоге они сдаются, потому что преграда оказывается слишком высокой. Действуя пошагово, вы избавите себя от постоянных опасений, что делаете что-то не так в других областях, отличных от той, на которой сосредоточены в настоящий момент.

В следующей главе вы начнете писать собственные юнит-тесты с использованием Jest — одного из самых популярных тестовых фреймворков для JavaScript.

ИТОГИ

- Хороший юнит-тест обладает следующими качествами.
 - Он должен выполняться быстро.
 - Он должен полностью контролировать тестируемый код.

¹ Мартин Р. Чистый код. СПб.: Питер.

- Он должен быть полностью изолированным (то есть выполняться независимо от других тестов).
- Он должен выполняться в памяти, не требуя обращений к файлам из файловой системы, сетям или базам данных.
- Он должен быть настолько синхронным и линейным, насколько это возможно.
- Точки входа — общедоступные (public) функции, которые предоставляют доступ к единицам работы и активируют нижележащую логику. Точки выхода — места, в которых можно проверить результат теста. Они представляют собой результаты выполнения единицы работы.
- Точка выхода может быть возвращаемым значением, изменением состояния или вызовом сторонней зависимости. Каждая точка выхода обычно требует отдельного теста, и каждая разновидность точки выхода требует отдельной методологии тестирования.
- Единица работы представляет собой совокупность действий, выполняемых между активацией точки входа до наблюдаемого конечного результата через одну или несколько точек выхода. Единица работы может охватывать отдельную функцию, модуль или несколько модулей.
- Интеграционное тестирование представляет собой юнит-тестирование, у которого некоторые (или все) зависимости реальны и существуют за пределами текущего процесса выполнения. И наоборот, юнит-тестирование напоминает интеграционное тестирование, у которого все зависимости находятся в памяти (как реальные, так и фиктивные) и их поведение может контролироваться в teste.
- Самые важные атрибуты любого теста — читабельность, простота в сопровождении и достоверность. Читабельность показывает, насколько легко прочитать и понять тест. Простота в сопровождении является оценкой того, насколько сложно проходит сопровождение тестового кода. Без достоверности в кодовую базу будет трудно вносить важные изменения (например, рефакторинг), что приведет к постепенному снижению качества кода.
- Разработка через тестирование (TDD) — методология, при которой тесты пишутся до написания рабочего кода. Этот метод также называется test-first (в отличие от code-first).
- Главное преимущество TDD — проверка правильности ваших тестов. Если ваши тесты не проходят до написания рабочего кода, это означает, что те же тесты не будут проходить при неправильной работе покрываемой ими функциональности.

2

Первый юнит-тест

В ЭТОЙ ГЛАВЕ

- ✓ Написание первого юнит-теста с использованием Jest
- ✓ Структура теста и соглашения об именах
- ✓ Работа с библиотеками утверждений
- ✓ Рефакторинг тестов и сокращение повторяющегося кода

Когда я только начинал писать юнит-тесты с реальным фреймворком юнит-тестирования, документации было очень мало и у фреймворков, с которыми я работал, не было нормальных примеров. (В то время я в основном программировал на VB 5 и 6.) Научиться работать с ними было непросто, и тесты у меня получались плохие. К счастью, ситуация изменилась. В JavaScript (и практически в любом существующем языке) имеются широкий выбор вариантов, обилие документации и поддержка от сообщества для тех, кто пытается освоить эти полезные ресурсы.

В предыдущей главе мы написали очень простой самодельный тестовый фреймворк. В этой главе рассматривается Jest – фреймворк, который был выбран для примеров этой книги.

2.1. ЗНАКОМСТВО С JEST

Jest – тестовый фреймворк с открытым кодом, созданный компанией Facebook. Он прост в использовании, его компоненты легко запоминаются, и он обладает выдающимися возможностями. В наше время он широко применяется во многих областях отрасли для тестирования как бэкенд-, так и фронтенд-проектов. Jest поддерживает две основные разновидности синтаксиса тестов (в одной используется слово `test`, а другая базируется на синтаксисе `Jasmine` – фреймворке, который стал прообразом для многих функций Jest). Мы опробуем обе версии, чтобы вы выяснили, какая вам нравится больше.

Кроме Jest, в JavaScript существует много других тестовых фреймворков, и практически все они распространяются с открытым кодом. Между ними существуют определенные различия как в стиле, так и в API, но для целей этой книги это несущественно.

2.1.1. Подготовка среды

Убедитесь в том, что у вас локально установлена среда Node.js. Инструкции по ее установке и запуску на вашей машине доступны по адресу <https://nodejs.org/en/download/>. Сайт предлагает на выбор либо версию с долгосрочной поддержкой (LTS, Long-Term Support), либо текущую версию. LTS-версия ориентирована на корпоративные среды, тогда как текущая версия чаще обновляется. Для целей этой книги подойдет любой вариант.

Убедитесь в том, что на вашей машине установлен менеджер пакетов `npm` (Node Package Manager). Он включен в поставку Node.js, так что выполните в командной строке команду `npm -v`, и, если вы увидите версию 6.10.2 и выше, все должно быть нормально. Если нет, убедитесь в том, что среда Node.js установлена на вашей машине.

2.1.2. Подготовка рабочего каталога

Чтобы начать работу с Jest, создайте новый пустой каталог с именем `ch2` и инициализируйте его пакетным менеджером по своему выбору. Я буду использовать `npm` (нужно было что-то выбрать). Yarn – еще один альтернативный менеджер пакетов. Для целей этой книги неважно, какой менеджер пакетов вы будете применять.

Для работы Jest нужен файл `jest.config.js` или `package.json`. Мы будем использовать второй, `npm` сгенерирует его за вас.

```
mkdir ch2
cd ch2
npm init --yes
//или
```

```
yarn init -yes
git init
```

Я также инициализирую Git в этом каталоге. Это рекомендуется сделать в любом случае для отслеживания изменений, а Jest использует этот файл в своей внутренней работе для отслеживания изменений в файлах и запуска конкретных тестов. Это упрощает работу Jest.

По умолчанию Jest ищет данные конфигурации в файле `package.json`, который создается командой, либо в специальном файле `jest.config.js`. Пока нам будет достаточно файла `package.json`. Если вам захочется больше узнать о параметрах конфигурации Jest, обращайтесь по адресу <https://jestjs.io/docs/en/configuration>.

2.1.3. Установка Jest

Затем необходимо установить Jest. Чтобы установить его как зависимость разработки (что означает, что он не будет распространяться в продакшен), можно ввести следующую команду:

```
npm install --save-dev jest
//или
yarn add jest -dev
```

Команда создает новый файл `jest.js` в каталоге `[корневой каталог]/node_modules/bin`. Затем можно запустить Jest командой `npx jest`.

Также можно установить Jest *глобально* на локальной машине (я рекомендую сделать это поверх установки `save-dev`):

```
npm install -g jest
```

Это позволит выполнить команду `jest` прямо из командной строки в любом каталоге с тестами, при этом вам не придется пользоваться `npm` для ее выполнения.

В реальных проектах для запуска тестов обычно используются команды `npm` вместо глобальной установки. На нескольких ближайших страницах я покажу, как это делается.

2.1.4. Создание файла теста

В Jest предусмотрены два стандартных способа поиска файлов с тестами.

- Если существует каталог `__tests__`, то все содержащиеся в нем файлы загружаются как файлы тестов независимо от их имен.
- Jest пытается найти все файлы, имена которых заканчиваются на `*.spec.js` или `*.test.js`, во всех подкаталогах в корневом каталоге вашего проекта (рекурсивный поиск).

Мы будем использовать первый вариант, но файлам также будут присваиваться имена с `*test.js` или `*.spec.js` на случай, если позднее мы захотим переместить их (и отказаться от использования каталога `__tests__`).

Jest можно настроить так, как вы считаете нужным, и указать местонахождение файлов в файле `jest.config.js` или `package.json`. С техническими подробностями можно ознакомиться по адресу <https://jestjs.io/docs/en/configuration>.

Следующим шагом станет создание в `ch2` специального каталога с именем `__tests__`. Создайте в этом каталоге файл, имя которого завершается на `test.js` или `spec.js` — например, `my-component.test.js`. Выберите один вариант, который посчитаете нужным, — это вопрос стиля. В книге будут встречаться оба варианта, хотя `test` будет чаще использоваться в очень простых примерах.

Чтобы начать пользоваться Jest, не нужно включать `require()` в начало файла. Глобальные функции импортируются автоматически. Среди функций, которые представляют для нас интерес, стоит особо выделить `test`, `describe`, `it` и `expect`. В листинге 2.1 показано, как может выглядеть простой тест.

Местоположение файлов с тестами

Существуют два основных варианта размещения таких файлов. Кто-то предпочитает располагать их сразу после тестируемых файлов и модулей. Другие же помещают все файлы с тестами в отдельный каталог. Неважно, какой подход вы выберете: главное придерживаться своего выбора на протяжении всего проекта, чтобы было легко находить файлы тестов для конкретного элемента.

Я думаю, что размещение файлов с тестами в отдельном каталоге предпочтительнее, потому что позволяет располагать вспомогательные файлы под тестовым каталогом рядом с самими тестами. Что касается удобного перехода между тестами и тестируемым кодом, то для большинства сред программирования существуют специальные плагины, которые позволяют перемещаться между кодом и тестами с помощью «горячих клавиш».

Листинг 2.1. Hello Jest

```
test('hello jest', () => {
  expect('hello').toEqual('goodbye');
});
```

Функции `describe` и `it` еще не использовались, но скоро мы до них доберемся.

2.1.5. Выполнение Jest

Чтобы запустить этот тест, необходимо иметь возможность выполнить Jest. Чтобы фреймворк Jest распознавался в командной строке, следует сделать одно из двух:

- установить Jest глобально командой `npm install jest -g`;
- воспользоваться `npm` для выполнения Jest из каталога `node_modules`, вводя команду `jest` в каталоге `ch2`.

Если все было сделано правильно, вы увидите результаты запуска теста Jest и сообщение о сбое. Ваш первый тест, который не прошел. Ура! На рис. 2.1 приведен вывод в моем терминале при выполнении этой команды. Приятно видеть такой красивый, многоцветный (если вы читаете электронную версию книги) и полезный вывод инструмента тестирования. Если в вашем терминале включен темный режим, выглядит еще круче.

```
[aout3-samples/ch2 [ jest
FAIL  __tests__/hellojest.test.js
  X hello jest (14ms)

  • hello jest

    expect(received).toEqual(expected) // deep equality

      Expected: "goodbye"
      Received: "hello"

        1 | test('hello jest', () => {
        > 2 |   expect('hello').toEqual('goodbye');
           |
           3 | });
           4 |

      at Object.toEqual (__tests__/hellojest.test.js:2:21)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:  0 total
Time:        1.145s
Ran all test suites.
aout3-samples/ch2 [ ]
```

Рис. 2.1. Вывод Jest на терминал

А теперь присмотримся к подробностям. На рис. 2.2 приведен тот же вывод, но с числовыми метками для отслеживания. Посмотрим, сколько информации здесь представлено.

- ❶ Краткий список всех непрошедших тестов (с именами), помеченных красным символом «X».

- ❷ Подробное описание условия, которое не было выполнено.
- ❸ Различия между фактическим и ожидаемым значением.
- ❹ Тип выполненной проверки.
- ❺ Код теста.
- ❻ Конкретная строка, в которой тест не прошел (в наглядном представлении).
- ❼ Отчет о количестве запущенных, непрошедших и прошедших тестов.
- ❽ Затраченное время.
- ❾ Количество скриншотов (не относится к нашему обсуждению).

Представьте, что вам пришлось бы писать всю эту функциональность самостоятельно. Это можно сделать, но у кого будут время и желание? Вдобавок вам придется заняться обработкой ошибок в механизме выдачи отчета.

```
FAIL __tests__/hellojest.test.js
  × hello jest (6ms) ❶
    • hello jest
      ❷ expect(received).toEqual(expected) // deep equality ❸
        Expected: "goodbye"
        Received: "hello" ❹
          1 |   test('hello jest', () => { ❺
❻ > 2 |     expect('hello').toEqual('goodbye');
          |
          3 |   });
          4 |
          at Object.toEqual (__tests__/hellojest.test.js:2:21)
  ❼
Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.743s, estimated 1s ❻
Ran all test suites matching /__tests__/\hellojest.test.js/i.
```

Рис. 2.2. Вывод Jest на терминал с пометками

Если в тесте заменить `goodbye` на `hello`, вы увидите, что произойдет при прохождении теста (рис. 2.3). Все красивое и зеленое, как и должно быть (опять же в электронной версии; в печатном издании все красивое и серое).

Возможно, вы заметили, что выполнение простого теста Hello World заняло 1,5 секунды. Если бы мы воспользовались командой `jest --watch`,

можно было бы приказать Jest отслеживать операции файловой системы с каталогом:

```
[aout3-samples/ch2 [ jest
  PASS  __tests__/hellojest.test.js
    ✓ hello jest (4ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.487s
Ran all test suites.
aout3-samples/ch2 [ ]
```

Рис. 2.3. Вывод Jest на терминал для прошедшего теста

и автоматически запускать тесты для изменившихся файлов без повторной инициализации. Это сэкономит значительное время и сочетается с идеей *непрерывного тестирования*. Откройте терминал в другом окне своей рабочей станции и выполните в нем команду `jest --watch`, и вы сможете продолжать кодинг и быстро получать обратную связь о проблемах, которые могли возникнуть. Так вы сумеете быстро составить представление о происходящем.

Jest также поддерживает тестирование в стиле `async` и обратные вызовы. Об этом вы прочтете позднее, когда в книге будут рассматриваться соответствующие темы, но, если вы хотите больше узнать об этом стиле работы, обратитесь к документации Jest: <https://jestjs.io/docs/en/asynchronous>.

2.2. БИБЛИОТЕКА ДЛЯ ТЕСТИРОВАНИЯ, БИБЛИОТЕКА УТВЕРЖДЕНИЙ, ЗАПУСК ТЕСТОВ И ПРЕДСТАВЛЕНИЕ ОТЧЕТА

Jest выполнил за нас целый ряд функций.

- Jest работает как *библиотека для тестирования*, используемая при написании тестов.
- Jest работает как *библиотека утверждений* для выполнения проверок в teste (`expect`).
- Jest обеспечивает *запуск тестов*.
- Jest выводит *информацию о результатах тестирования*.

Jest также предоставляет средства *изоляции* для создания моков, стабов и шпионов (которые еще не рассматривались). Эти понятия будут раскрыты в дальнейших главах.

В других языках тестовые фреймворки, помимо предоставления средств изоляции, очень часто выполняют и другие упомянутые роли — библиотека для тестирования, библиотека утверждений, среда запуска тестов и системы формирования отчетов, — но JavaScript выглядит чуть более фрагментированным. Многие другие тестовые фреймворки предоставляют только часть этих возможностей. Иногда это связано с мантрой «делать что-то одно, зато хорошо», воспринятой слишком буквально, а иногда объясняется другими причинами. В любом случае Jest выделяется на общем фоне своей универсальностью. Это свидетельство моих разработки с открытым кодом в JavaScript: для каждой из этих ролей существуют разные инструменты, которые можно комбинировать для создания собственного инструментария.

Одна из причин, по которым я выбрал Jest для своей книги, заключается в том, что с Jest вам не надо отвлекаться на инструменты или решать проблемы с недостающей функциональностью — можно просто сосредоточиться на содержательной части. Так что нам не придется использовать несколько фреймворков в книге, которая в основном посвящена паттернам и антипаттернам тестирования.

2.3. ЧТО ПРЕДЛАГАЮТ ФРЕЙМВОРКИ ЮНИТ-ТЕСТИРОВАНИЯ

Задержимся ненадолго на этом вопросе и посмотрим, в какой ситуации мы находимся. Что нам предлагают такие фреймворки, как Jest, по сравнению с созданием собственного фреймворка (работу над которым мы начали в предыдущей главе) или ручным тестированием?

- *Структура* — вам не приходится изобретать велосипед каждый раз, когда потребуется что-нибудь протестировать. При использовании фреймворка тестирования все всегда начинается одинаково — с написания теста, имеющего четко определенную структуру, которую каждый легко узнает, прочитает и поймет.
- *Повторяемость* — при использовании фреймворка тестирования акт написания нового теста легко повторяется. Так же легко повторяется выполнение теста с помощью программы для запуска тестов, это можно быстро и просто делать по многу раз в день. Так же просто разобраться в причинах, почему тест не прошел. Кто-то уже проделал всю тяжелую работу за нас, и нам не приходится программировать всю эту функциональность в самодельном фреймворке.
- *Уверенность в результатах и экономия времени* — когда вы создаете собственный фреймворк тестирования, он с большой вероятностью будет содержать ошибки, потому что не прошел такой проверки, как существующий, зрелый и широко применяемый на практике фреймворк. С другой стороны, ручное тестирование обычно занимает очень много времени. Когда времени не хватает, вы скорее всего сосредоточитесь на тестировании того, что воспринимается как критически важное, и пропустите то, что кажется второстепенным.

При этом можно упустить мелкие, но значимые ошибки. Когда написание новых тестов упрощается, повышается вероятность того, что вы напишете тесты для аспектов, которые кажутся менее существенными, потому что вам не приходится тратить слишком много времени на тесты для главных модулей и компонентов.

- *Общее понимание* — стандартизация отчетов фреймворка также упрощает управление задачами на уровне команд (если тест проходит, значит, задача выполнена). Многие разработчики считают, что это полезно.

Короче говоря, фреймворки для написания, запуска и анализа результатов юнит-тестов могут оказать огромное влияние на повседневную жизнь разработчиков, готовых выделить время на то, чтобы научиться ими правильно пользоваться. На рис. 2.4 показаны области разработки, на которые могут повлиять фреймворки юнит-тестирования и их сопутствующие инструменты, а в табл. 2.1 перечислены основные типы действий, которые обычно выполняются с фреймворками тестирования.

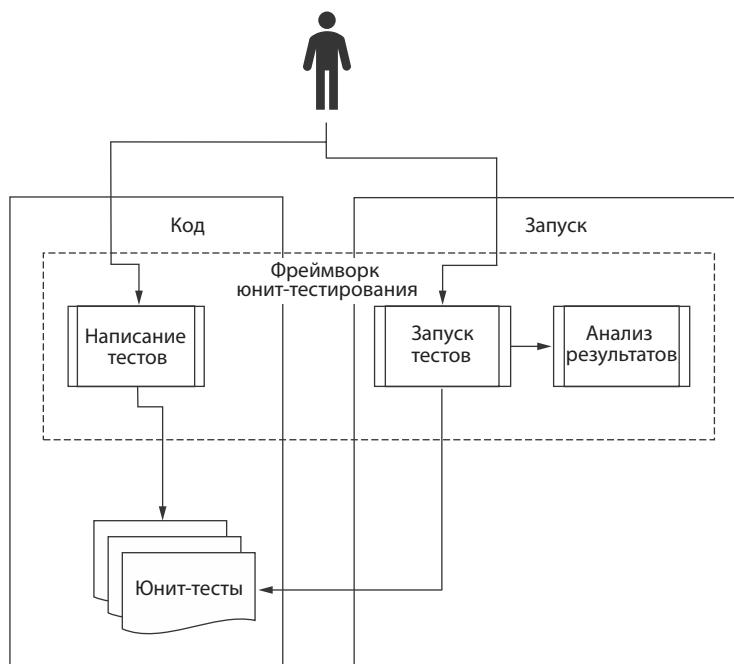


Рис. 2.4. Юнит-тесты пишутся в программном коде, при этом используются библиотеки из фреймворка юнит-тестирования. Тесты запускаются из подсистемы выполнения тестов внутри IDE или из командной строки, а результаты просматриваются (как текстовый вывод или в IDE) разработчиком либо автоматизированным процессом сборки

Таблица 2.1. Как фреймворки тестирования помогают разработчикам писать/выполнять тесты и просматривать результаты

| Практика юнит-тестирования | Как помогает фреймворк |
|--|--|
| Простое написание структурированных тестов | Фреймворк предоставляет разработчику вспомогательные функции, функции проверки утверждений и функции, относящиеся к структуре тестирования |
| Выполнение одного или нескольких тестов | Фреймворк предоставляет средства для запуска тестов (обычно из командной строки), которые: <ul style="list-style-type: none"> находят тесты в вашем коде; автоматически запускают тесты; отображают статус теста во время выполнения |
| Просмотр результатов запуска тестов | Подсистема выполнения тестов обычно выдает следующую информацию: <ul style="list-style-type: none"> сколько тестов запущено; сколько тестов не запущено; сколько тестов не прошло; какие тесты не прошли; по каким причинам тесты не прошли; позиция в коде, в которой произошел сбой; возможны полная трассировка стека для любых исключений, которые привели к непрохождению теста, и переход к различным вызовам методов в стеке вызовов |

На момент написания книги существовало более 900 фреймворков юнит-тестирования, и для каждого широко используемого языка программирования (и нескольких мертвых) существовала хотя бы пара таких фреймворков. Хороший список имеется в Википедии: https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks.

ПРИМЕЧАНИЕ Использование фреймворка юнит-тестирования не гарантирует, что написанные вами тесты будут *читабельными, простыми в сопровождении* или *достоверными* или что они покрывают всю логику, которую вам хотелось бы протестировать. Я покажу, как наделить ваши юнит-тесты этими свойствами, в главах с 7-й по 9-ю (а также в нескольких других местах книги).

2.3.1. Фреймворки xUnit

Когда я начал писать тесты (во времена Visual Basic), стандарт, по которому оценивалось большинство фреймворков юнит-тестирования, обозначался общим термином xUnit. Предком идеи фреймворков xUnit был SUnit – фреймворк юнит-тестирования для Smalltalk.

Имена фреймворков юнит-тестирования обычно начинаются с первых букв языка программирования, для которого они строятся; например, для C++ может использоваться имя CppUnit, JUnit – для Java, NUnit и xUnit – для .NET, – HUnit для языка программирования Haskell. Не все фреймворки следуют этим соглашениям, но большинство им подчиняется.

2.3.2. Структуры xUnit, TAP и Jest

Не только имена оставались последовательными до разумной степени. При использовании фреймворка xUnit также можно было ожидать конкретную структуру, по которой строились тесты. При запуске такие фреймворки выводили результаты в единый файл, обычно XML-файл с конкретной схемой.

Такая разновидность отчетов xUnit в формате XML все еще доминирует и широко используется во многих средствах сборки (например, Jenkins), которые поддерживают этот формат при помощи встроенных плагинов и используют его для вывода результатов запуска. Многие фреймворки юнит-тестирования для статических языков продолжают применять модель xUnit для структурирования; это означает, что когда вы изучили один фреймворк, то сможете легко работать с любым из них (конечно, если вы знаете соответствующие языки программирования).

Другой интересный стандарт для структуры отчетов о результатах тестов (и не только) называется TAP (Test Anything Protocol). TAP начал свое существование как часть тестовой оснастки для Perl, но сейчас его реализации существуют в C, C++, Python, PHP, Perl, Java, JavaScript и других языках. TAP – нечто гораздо большее, чем спецификация отчетов. В мире JavaScript фреймворк TAP – самый известный тестовый фреймворк со встроенной поддержкой протокола TAP.

Формально Jest не является фреймворком из семейства xUnit или TAP. Его вывод несовместим с xUnit или TAP по умолчанию. Тем не менее, так как отчеты в стиле xUnit все еще доминируют в мире средств сборки, обычно этот протокол желательно использовать для выдачи отчетов на сервере сборки. Чтобы получить результаты тестов Jest в формате, который легко распознается многими средствами сборки, можно установить такие модули npm, как `jest-xunit` (если вам нужен вывод в формате TAP, используйте `jest-tap-reporter`), а затем воспользоваться в проекте специальным файлом `jest.config.js` для изменения формата отчетов Jest.

А теперь сделаем следующий шаг и напишем что-то похожее на реальный тест на базе Jest.

2.4. ПРОЕКТ PASSWORD VERIFIER

Проект, который мы будем использовать во многих примерах тестов этой книги, изначально сделаем очень простым: в нем будет всего одна функция. Далее мы будем дополнять его новой функциональностью, модулями и классами для демонстрации различных аспектов юнит-тестирования. Проект будет называться Password Verifier.

Первый сценарий также очень прост. Мы построим библиотеку проверки паролей, и сначала она будет состоять всего из одной функции. Эта функция `verifyPassword(rules)` позволяет добавлять нестандартные функции проверки (`rules`) и выводит список ошибок в соответствии с введенными правилами проверки. Каждая функция-правило выводит два поля:

```
{
  passed: (boolean),
  reason: (string)
}
```

В этой книге я научу вас писать тесты, проверяющие функциональность `verifyPassword` несколькими способами при добавлении новых возможностей.

В листинге 2.2 приведена версия 0 этой функции с очень наивной реализацией.

Листинг 2.2. Password Verifier версия 0

```
const verifyPassword = (input, rules) => {
  const errors = [];
  rules.forEach(rule => {
    const result = rule(input);
    if (!result.passed) {
      errors.push(`error ${result.reason}`);
    }
  });
  return errors;
};
```

Честно говоря, этот код трудно отнести к функциональному стилю, и позднее мы можем его немного переработать, но я хотел сделать код предельно простым, чтобы вы могли сосредоточиться на тестах.

Функция делает не так много. Она перебирает заданные правила и запускает их все с заданным вводом `input`. Если результат применения правила отличается от `passed` (пройдено), то в массив ошибок добавляется новая ошибка и этот массив возвращается в качестве окончательного результата.

2.5. ПЕРВЫЙ ТЕСТ JEST ДЛЯ VERIFYPASSWORD

Будем считать, что вы установили Jest в своей системе. Создайте новый файл с именем `password-verifier0.spec.js` в каталоге `__tests__`.

Использование каталога `__tests__` — всего лишь одна из возможных схем организации ваших тестов, которая является частью конфигурации Jest по умолчанию. Многие разработчики предпочитают размещать файлы тестов вместе с тестируемым кодом. У каждого метода есть свои плюсы и минусы, которые будут рассмотрены в других частях книги. А пока ограничимся конфигурацией по умолчанию.

В листинге 2.3 приведена первая версия теста нашей новой функции.

Листинг 2.3. Первый тест для `verifyPassword()`

```
test('badly named test', () => {
  const fakeRule = input =>
    ({ passed: false, reason: 'fake reason' });
  const errors = verifyPassword('any value', [fakeRule]);
  expect(errors[0]).toMatch('fake reason');
});
```

2.5.1. Паттерн «подготовь — действуй — проверь»

Структура теста в листинге 2.3 часто называется паттерном AAA (Arrange – Act – Assert, «подготовь — действуй — проверь»). И это очень удобно! Мне кажется, очень просто описывать части теста, говоря что-то вроде «часть подготовки получилась слишком сложной» или «а где часть действия?»

В части подготовки (Arrange) создается фиктивное правило, которое всегда возвращает `false`; проверка в конце теста позволит доказать, что правило действительно используется. Затем это правило передается `verifyPassword` с простым вводом. В разделе проверки (Assert) мы убеждаемся в том, что первая полученная ошибка соответствует фиктивной причине, переданной в части подготовки. Конструкция `.toMatch(/string/)` использует регулярное выражение для поиска части строки. Она эквивалентна использованию `.toContain('fake reason')`.

Было бы слишком утомительно вручную запускать Jest после того, как мы напишем тест или что-то исправим. Давайте настроим npm для автоматического запуска Jest. Перейдите к файлу `package.json` в корневом каталоге `ch2` и добавьте следующие записи в раздел `scripts`:

```
"scripts": {
  "test": "jest",
  "testw": "jest --watch" //Если не используете git, замените на --watchAll
},
```

Если у вас нет экземпляра Git, инициализированного в этом каталоге, используйте команду `--watchAll` вместо `--watch`.

Если все прошло хорошо, вы теперь сможете ввести команду `npm test` в командной строке из каталога `ch2` и Jest выполнит тесты однократно. Если ввести команду `npm run testw`, Jest запустит бесконечный цикл и начнет ожидать изменений, пока вы не завершите процесс комбинацией клавиш `Ctrl+C`. (Присутствие `run` необходимо из-за того, что `testw` не входит в число специальных ключевых слов, распознаваемых `npm` автоматически.)

Запустив тест, вы увидите, что он проходит успешно, потому что функция работает так, как ожидалось.

2.5.2. Тестируем тест

Внесем ошибку в рабочий код и посмотрим, провалится ли тест там, где он должен провалиться.

Листинг 2.4. Внедрение ошибки

```
const verifyPassword = (input, rules) => {
  const errors = [];
  rules.forEach(rule => {
    const result = rule(input);
    if (!result.passed) {
      // errors.push(`error ${result.reason}`);
    }
  });
  return errors;
};
```

Эта строка была случайно закомментирована

Теперь вы должны увидеть, что ваш тест не проходит и на экран выводится соответствующее красивое сообщение. Раскомментируйте эту строку и убедитесь в том, что тесты снова проходят. Это отличный способ обрести некоторую уверенность в ваших тестах, если вы не практикуете разработку через тестирование и пишете тесты после кода.

2.5.3. Имена USE

Имя нашего теста выбрано крайне неудачно. Оно ничего не сообщает о том, чего вы пытаетесь добиться. Я люблю включать в имена тестов три вида информации, чтобы читатель теста мог получить ответы на большинство возникающих вопросов, просто взглянув на имя:

- тестируемая единица работы (в данном случае функция `verifyPassword`);
- сценарий или входные данные (нарушение правила);
- ожидаемое поведение или точка выхода (возвращает обоснованную ошибку).

В процессе ревью Тайлер Лемке (Tyler Lemke), рецензент книги, предложил для этой схемы удачное сокращение USE: Unit under test (тестируемая единица), Scenario (сценарий), Expectation (ожидание). Мне понравилось это сокращение, оно легко запоминается. Спасибо, Тайлер!

В листинге 2.5 приведена следующая версия теста с именем, назначенным по принципу USE.

Листинг 2.5. Назначение тесту имени по принципу USE

```
test('verifyPassword, given a failing rule, returns errors', () => {
  const fakeRule = input => ({ passed: false, reason: 'fake reason' });

  const errors = verifyPassword('any value', [fakeRule]);
  expect(errors[0]).toContain('fake reason');
});
```

Немного лучше. Если тест не проходит, особенно в процессе сборки, то вы не видите комментарии или полный код теста. Обычно вы видите только имя. Оно должно быть настолько понятным, что у вас не возникнет необходимости смотреть на код теста, чтобы понять, где могут скрываться проблемы с рабочим кодом.

2.5.4. Сравнения строк и сопровождаемость

Мы также внесли небольшое изменение в следующей строке:

```
expect(errors[0]).toContain('fake reason');
```

Вместо того чтобы проверять, что одна строка равна другой, как это очень часто происходит в тестах, мы проверяем, что строка содержится в выводе. При этом наш тест становится чуть более надежным для будущих изменений в выводе. Для этого можно воспользоваться методами `.toContain` или `.toMatch(/fake reason/)`, использующими регулярное выражение для поиска совпадения с частью строки.

Строки — разновидность пользовательского интерфейса. Они видны людям, и они могут изменяться — особенно по краям. К строкам могут добавляться пробельные символы, табуляции, звездочки или другие украшения. Для нас важно то, чтобы существовала *основная* информация, содержащаяся в строке. Мы не хотим менять наш тест каждый раз, когда кто-нибудь добавит новый символ в конец строки. Это часть мышления, которое нам хотелось бы привить в своих тестах: простота в сопровождении тестов со временем и устойчивость тестов при изменениях чрезвычайно важны.

В идеале нам хотелось бы, чтобы тесты не проходили только в том случае, если в рабочем коде что-то не так. Количество ложнопозитивных тестов должно быть

сокращено до минимума. Использование `toContain()` или `toMatch()` — отличный способ продвижения к этой цели.

На протяжении всей книги будут рассматриваться другие способы обеспечения сопровождаемости тестов, особенно много мы будем говорить об этом в части 2.

2.5.5. Использование `describe()`

При помощи функции Jest `describe()` можно еще лучше структурировать наш тест и начать отделять три части принципа USE друг от друга. И этот шаг, и другие последующие остаются полностью на ваше усмотрение: вы сами решаете, как вы хотите оформить свой тест, и выбрать его структуру для читабельности. Я показываю, как это делается, потому что многие люди либо не умеют эффективно использовать `describe()`, либо вообще забывают об этой функции. А она может быть весьма полезной.

Функции `describe()` дополняют наши тесты контекстом: как логическим для читателя, так и функциональным для самого теста. В листинге 2.6 показано, как ими можно пользоваться.

Листинг 2.6. Добавление блока `describe()`

```
describe('verifyPassword', () => {
  test('given a failing rule, returns errors', () => {
    const fakeRule = input =>
      ({ passed: false, reason: 'fake reason' });

    const errors = verifyPassword('any value', [fakeRule]);

    expect(errors[0]).toContain('fake reason');
  });
});
```

Я внес четыре изменения:

- добавил блок `describe()` с описанием тестируемой единицы работы. Мне такая структура кажется более ясной. Кроме того, создается впечатление, что после блока можно добавить больше вложенных тестов. Блок `describe()` также помогает подсистеме вывода результатов командной строки строить более удобные отчеты;
- вложил `test` под новым блоком и удалил имя тестируемой единицы работы;
- добавил `input` в строку `reason` фиктивного правила;
- добавил пустые строки между частями подготовки, действия и проверки, чтобы тест стал более читабельным (особенно для новых участников команды).

2.5.6. Структура, подразумевающая контекст

Среди плюсов `describe()` можно отметить возможность вложения функции в саму себя. Таким образом, ее можно использовать для создания нового уровня с пояснением сценария и вложить свой тест в него.

Листинг 2.7. Вложенные вызовы `describe()` для дополнительного контекста

```
describe('verifyPassword', () => {
  describe('with a failing rule', () => {
    test('returns errors', () => {
      const fakeRule = input => ({ passed: false,
                                   reason: 'fake reason' });

      const errors = verifyPassword('any value', [fakeRule]);

      expect(errors[0]).toContain('fake reason');
    });
  });
});
```

Некоторые разработчики терпеть не могут такую структуру, но, на мой взгляд, в ней есть некоторая элегантность. Вложение позволяет вынести три блока критически важной информации на отдельные уровни. Более того, при желании также можно вынести ложное правило за пределы теста, прямо под соответствующий вызов `describe()`.

Листинг 2.8. Вложенные вызовы `describe()` с выделением ввода

```
describe('verifyPassword', () => {
  describe('with a failing rule', () => {
    const fakeRule = input => ({ passed: false,
                                 reason: 'fake reason' });

    test('returns errors', () => {
      const errors = verifyPassword('any value', [fakeRule]);

      expect(errors[0]).toContain('fake reason');
    });
  });
});
```

В следующем примере я верну это правило обратно в тест (мне нравится, когда фрагменты информации находятся поблизости друг от друга — но об этом позже).

Вложенная структура также очень хорошо передает идею о том, что в конкретном сценарии может быть более одного ожидаемого поведения. В сценарии можно проверить несколько точек выхода с отдельным тестом для каждой, и это будет выглядеть логично с точки зрения читателя кода.

2.5.7. Функция `it()`

В головоломке, которую мы строили до настоящего момента, не хватает одного фрагмента. Jest также предоставляет функцию `it()`. Эта функция во всех отношениях является *синонимом* для функции `test()`, но она лучше соответствует синтаксису на базе `describe`, описанному ранее.

В листинге 2.9 показано, как будет выглядеть тест при замене `test()` на `it()`.

Листинг 2.9. Замена `test()` на `it()`

```
describe('verifyPassword', () => {
  describe('with a failing rule', () => {
    it('returns errors', () => {
      const fakeRule = input => ({ passed: false,
                                   reason: 'fake reason' });

      const errors = verifyPassword('any value', [fakeRule]);

      expect(errors[0]).toContain('fake reason');
    });
  });
});
```

Взглянув на этот тест, очень легко понять, к чему он относится. Синтаксис является естественным расширением описанных выше блоков `describe()`. И снова решайте сами, хотите ли вы использовать этот стиль. Я привожу лишь один из возможных вариантов того, как это можно делать.

2.5.8. Две разновидности Jest

Как вы уже видели, Jest поддерживает два основных способа написания тестов: компактный `test`-синтаксис и так называемый иерархический синтаксис на базе `describe`.

Синтаксис Jest на базе `describe` в значительной степени происходит от Jasmine, одного из самых старых тестовых фреймворков для JavaScript. Истоки самого стиля можно отследить до Ruby и известного тестового фреймворка RSpec Ruby. Этот иерархический стиль обычно называется стилем *BDD* (Behavior Driven Development, то есть *разработка через поведение*).

Эти стили можно смешивать так, как вы считаете нужным (я так и поступаю). Вы используете синтаксис `test`, когда можно легко понять цель теста и весь его контекст. Синтаксис `describe` может помочь, когда вы ожидаете множественных результатов с одной точки входа в одном сценарии. Я описываю оба стиля, потому что иногда использую компактный синтаксис, а иногда иерархический в зависимости от сложности и требований к выразительности.

Темное настоящее BDD

У BDD интересная история, о которой стоит поговорить. Термин BDD не связан с TDD (Test Driven Development). Дэн Норт (Dan North) — человек, оказавший наибольшее влияние на изобретение термина, — определяет BDD как практику описания желаемого поведения приложения на историях и примерах. В основном он ориентирован на взаимодействия с нетехническими стейкхолдерами — владельцами продуктов, клиентами и т. д. RSpec (создававшийся по образцу RBehave) принес в массы подход, основанный на историях, и к процессу подключились многие другие фреймворки, в том числе знаменитый Cucumber.

У этой истории также есть темная сторона: многие фреймворки были разработаны и использовались исключительно разработчиками без взаимодействия с нетехническими стейкхолдерами — в полном противоречии с главными идеями BDD.

Сегодня для меня термин «*BDD-фреймворки*» означает прежде всего «фреймворки тестирования с синтаксическим сахаром», потому что они почти никогда не используются для создания реальных взаимодействий между заинтересованными участниками и почти всегда применяются как еще один модный или предписанный инструмент для выполнения автоматизированных тестов. Я даже видел, как могущественный Cucumber относят к этой категории.

2.5.9. Рефакторинг рабочего кода

Так как в JavaScript одно и то же можно сделать многими разными способами, я решил представить пару вариаций нашей структуры и результаты ее изменений. Допустим, вы хотите реализовать средства проверки пароля в виде объекта с состоянием.

Одна из причин для перехода к дизайну с состоянием может заключаться в том, что вы хотите использовать этот объект в разных частях приложения. Одна часть будет настраивать объект и добавлять в него правила, а другая — использовать его для выполнения верификации. Другая причина — вам хочется понимать, как работать с дизайном с состоянием, знать, как он влияет на тесты и что с этим можно сделать.

Начнем с рабочего кода.

Листинг 2.10. Рефакторинг функции в класс с состоянием

```
class PasswordVerifier1 {
  constructor () {
```

```

    this.rules = [];
}

addRule (rule) {
  this.rules.push(rule);
}

verify (input) {
  const errors = [];
  this.rules.forEach(rule => {
    const result = rule(input);
    if (result.passed === false) {
      errors.push(result.reason);
    }
  });
  return errors;
}
}

```

Я выделил основные изменения по сравнению с листингом 2.9. Ничего особенного здесь не происходит, хотя вы будете чувствовать себя более уверенно, если у вас имеется опыт объектно-ориентированного программирования. Важно заметить, что это всего лишь один из возможных способов проектирования данной функциональности. Я использую решение на базе класса, чтобы показать, как этот дизайн может повлиять на тест.

Где в новом дизайне находятся точки входа и выхода для текущего сценария? Подумайте над этим вопросом. Область действия единицы работы увеличилась. Чтобы протестировать сценарий с правилом, дающим сбой, необходимо вызвать две функции, влияющие на состояние тестируемого юнита: `addRule` и `verify`.

Посмотрим, как может выглядеть тест (как обычно, изменения выделены жирным шрифтом).

Листинг 2.11. Тестирование единицы работы с состоянием

```

describe('PasswordVerifier', () => {
  describe('with a failing rule', () => {
    it('has an error message based on the rule.reason', () => {
      const verifier = new PasswordVerifier1();
      const fakeRule = input => ({ passed: false,
                                    reason: 'fake reason'});

      verifier.addRule(fakeRule);
      const errors = verifier.verify('any value');

      expect(errors[0]).toContain('fake reason');
    });
  });
});

```

Пока все идет нормально; ничего необычного нет. Обратите внимание на увеличение единицы работы. Теперь она охватывает две взаимосвязанные функции, которые должны работать вместе (`addRule` и `verify`). Появляется *связь* (*coupling*), обусловленная природой дизайна с состоянием. Для эффективного тестирования приходится использовать две функции без раскрытия какого-либо внутреннего состояния объекта.

Сам тест выглядит безобидно. Но что произойдет, если вы захотите написать несколько тестов для того же сценария? Такая ситуация может возникнуть при наличии нескольких точек выхода или при необходимости протестировать несколько результатов из одной точки выхода. Допустим, вы хотите убедиться в том, что произошла только одна ошибка. Можно просто добавить в тест новую строку:

```
verifier.addRule(fakeRule);
const errors = verifier.verify('any value');
expect(errors.length).toBe(1); ← Новое утверждение
expect(errors[0]).toContain('fake reason');
```

Что произойдет, если новое утверждение не пройдет? Второе утверждение никогда не будет выполняться, потому что подсистема исполнения тестов получит ошибку и перейдет к следующему тестовому случаю.

Но вам все равно хочется знать, прошло ли второе утверждение, верно? Кто-то попробует закомментировать первое и запустить тест заново. Применять такие способы выполнения тестов не рекомендуется. В книге Джерарда Месароша «xUnit Test Patterns» это человеческое поведение с закрытием комментариями одних частей для тестирования других называется *рулеткой утверждений* (*assertion roulette*). Оно может породить значительную путаницу и ложные позитивные тесты при запусках (вы думаете, что проверка прошла или не прошла, хотя на самом деле это не так).

Я бы предпочел выделить дополнительную проверку в отдельный тестовый случай с хорошо выбранным именем, как показано в листинге 2.12.

Листинг 2.12. Проверка дополнительного результата от той же точки выхода

```
describe('PasswordVerifier', () => {
  describe('with a failing rule', () => {
    it('has an error message based on the rule.reason', () => {
      const verifier = new PasswordVerifier1();
      const fakeRule = input => ({ passed: false,
        reason: 'fake reason'});

      verifier.addRule(fakeRule);
      const errors = verifier.verify('any value');

      expect(errors[0]).toContain('fake reason');
    });
  });
});
```

```

it('has exactly one error', () => {
  const verifier = new PasswordVerifier1();
  const fakeRule = input => ({ passed: false,
    reason: 'fake reason'});

  verifier.addRule(fakeRule);
  const errors = verifier.verify('any value');

  expect(errors.length).toBe(1);
});
});
});
});

```

Все это начинает выглядеть плохо. Да, мы разобрались с проблемой «рулетки проверок». Каждый вызов `it()` проводит проверку независимо от других вызовов и не вмешиваясь в результаты других тестовых случаев. Но чего это стоило? Всего. Взгляните, до какой степени дублируется код. На этой стадии каждый читатель с опытом юнит-тестирования должен закричать: «Используйте `setup/beforeEach!`»

Отлично!

2.6. РЕШЕНИЕ С BEFOREEACH()

Я еще не упоминал `beforeEach()`. Функция и ее родственник `afterEach()` используются для подготовки и уничтожения конкретного состояния, необходимого для тестовых случаев. Также существуют функции `beforeAll()` и `afterAll()`, которых я стараюсь любой ценой избегать для сценариев юнит-тестирования. Родственные функции будут рассмотрены далее в книге.

`beforeEach()` поможет устраниить дублирование в тестах, потому что эта функция выполняется однократно перед каждым тестом в блоке `describe`, в который она вкладывается. Также возможно многократное вложение функции, как показано в листинге 2.13.

Листинг 2.13. Использование `beforeEach()` на двух уровнях

```

describe('PasswordVerifier', () => {
  let verifier;
  beforeEach(() => verifier = new PasswordVerifier1());
  describe('with a failing rule', () => {
    let fakeRule, errors;
    beforeEach(() => {
      fakeRule = input => ({passed: false, reason: 'fake reason'});
      verifier.addRule(fakeRule);
    });
    it('has an error message based on the rule.reason', () => {
      const errors = verifier.verify('any value');
      // Создание нового экземпляра verifier, который будет использоваться в каждом teste
      // Создание фиктивного правила, которое будет использоваться в этом методе describe()
    });
  });
}

```

```

    expect(errors[0]).toContain('fake reason');
  });
it('has exactly one error', () => {
  const errors = verifier.verify('any value');
  expect(errors.length).toBe(1);
});
});
});

```

Присмотритесь к выделенному коду.

В первом вызове `beforeEach()` создается объект `PasswordVerifier1`, который будет создаваться для каждого тестового случая. В следующем вызове `beforeEach()` создается фиктивное правило, которое добавляется к новому экземпляру `verifier` для каждого тестового случая в этом конкретном сценарии. Если бы у нас были другие сценарии, второй вызов `beforeEach()` в строке 6 не выполнялся бы, но первый был бы выполнен.

Тесты стали короче; в идеале именно это нам и нужно, чтобы тесты проще читались и создавали меньше проблем с сопровождением. Стока создания была удалена из каждого теста, а во всех случаях повторно использовалась одна высокоуровневая переменная `verifier`.

Необходимо сделать два замечания.

- Мы забыли очистить массив `errors` в `beforeEach()` в строке 6. Это может преподнести неприятные сюрпризы позднее.
- По умолчанию Jest запускает юнит-тесты параллельно. Это означает, что перемещение `verifier` в строку 2 может создать проблемы с параллельными тестами, где `verifier` может быть перезаписан другим параллельно выполняемым тестом, что приведет к нарушению состояния выполняемого теста. Jest сильно отличается от фреймворков юнит-тестирования в большинстве других известных мне языков, которые выполняют тесты в одном потоке, а не параллельно (по крайней мере по умолчанию) для предотвращения подобных проблем. С Jest приходится помнить, что параллельные тесты — это реальность, так что тесты с общим состоянием, как в строке 2 нашего примера, потенциально могут создать проблемы: тесты станут нестабильными, и будет непонятно, почему они не проходят.

Скоро мы исправим оба недостатка.

2.6.1. `beforeEach()` и утомительная прокрутка

В процессе рефакторинга с `beforeEach()` тесты лишились двух полезных свойств.

- Если я пытаюсь читать только части с `it()`, я не могу сказать, где создается и объявляется `verifier`. Чтобы понять это, придется прокручивать код к началу.

- То же происходит, когда я пытаюсь понять, какое правило было добавлено. Чтобы узнать это, мне приходится смотреть на один уровень выше `it()` или искать описание в блоке `describe()`.

На данный момент это не кажется серьезным недостатком. Но как вы увидите позднее, с ростом размера списка сценариев эта структура выглядит все хуже. В больших файлах может возникнуть явление, которое я называю *утомительной прокруткой*: читателю теста приходится прокручивать файл теста вверх и вниз, чтобы разобраться в контексте и состоянии тестов. В результате сопровождение и чтение тестов превращаются из простого акта чтения в монотонную скучную работу.

Вложение отлично подходит для получения информации, но оно неудобно для людей, которым приходится искать в других местах, откуда взялся тот или иной элемент. Если вам доводилось заниматься отладкой стилей CSS в окне инспектора браузера, это ощущение вам знакомо. Вы видите, что какая-то ячейка по какой-то причине выделена жирным шрифтом. Вам приходится прокручивать код вверх, чтобы узнать, какой стиль обеспечил выделение `<div>` во вложенных ячейках таблицы под третьим узлом.

Листинг 2.14 показывает, что произойдет, если сделать следующий шаг вперед. Так как мы занимаемся устранением дублирования кода, так же можно вызвать `verify` в `beforeEach()` и удалить лишнюю строку из каждого `it()`. Фактически это означает размещение частей подготовки и действия в паттерне AAA в функцию `beforeEach()`.

Листинг 2.14. Включение частей подготовки и действия в beforeEach()

```
describe('PasswordVerifier', () => {
  let verifier;
  beforeEach(() => verifier = new PasswordVerifier1());
  describe('with a failing rule', () => {
    let fakeRule, errors;
    beforeEach(() => {
      fakeRule = input => ({passed: false, reason: 'fake reason'});
      verifier.addRule(fakeRule);
      errors = verifier.verify('any value');
    });
    it('has an error message based on the rule.reason', () => {
      expect(errors[0]).toContain('fake reason');
    });
    it('has exactly one error', () => {
      expect(errors.length).toBe(1);
    });
  });
});
```

Дублирование кода свелось к минимуму, но теперь вам также придется искать, откуда и как взялся массив `errors`, если вы захотите понять логику каждого `it()`.

Немного усложним ситуацию, добавим еще несколько базовых сценариев и посмотрим, будет ли масштабироваться этот подход при увеличении пространства задачи.

Листинг 2.15. Добавление дополнительных сценариев

```
describe('v6 PasswordVerifier', () => {
  let verifier;
  beforeEach(() => verifier = new PasswordVerifier1());
  describe('with a failing rule', () => {
    let fakeRule, errors;
    beforeEach(() => {
      fakeRule = input => ({passed: false, reason: 'fake reason'});
      verifier.addRule(fakeRule);
      errors = verifier.verify('any value');
    });
    it('has an error message based on the rule.reason', () => {
      expect(errors[0]).toContain('fake reason');
    });
    it('has exactly one error', () => {
      expect(errors.length).toBe(1);
    });
  });
  describe('with a passing rule', () => {
    let fakeRule, errors;
    beforeEach(() => {
      fakeRule = input => ({passed: true, reason: ''});
      verifier.addRule(fakeRule);
      errors = verifier.verify('any value');
    });
    it('has no errors', () => {
      expect(errors.length).toBe(0);
    });
  });
  describe('with a failing and a passing rule', () => {
    let fakeRulePass,fakeRuleFail, errors;
    beforeEach(() => {
      fakeRulePass = input => ({passed: true, reason: 'fake success'});
      fakeRuleFail = input => ({passed: false, reason: 'fake reason'});
      verifier.addRule(fakeRulePass);
      verifier.addRule(fakeRuleFail);
      errors = verifier.verify('any value');
    });
    it('has one error', () => {
      expect(errors.length).toBe(1);
    });
    it('error text belongs to failed rule', () => {
      expect(errors[0]).toContain('fake reason');
    });
  });
});
```

Вам нравится? Мне нет. Теперь мы видим пару новых проблем.

- Я уже начинаю видеть множество повторений в частях `beforeEach()`.
- Потенциал утомительной прокрутки значительно увеличивается, а количество вариантов влияния каждого `beforeEach()` на каждое состояние `it()` заметно возрастает.

В реальных проектах функции `beforeEach()` в файлах тестов часто превращаются в мусорную свалку. Люди сваливают туда все, что инициализируется в тестах: то, что нужно только некоторым тестам; то, что влияет на все остальные тесты; то, что вообще никем не используется. Люди по своей природе склонны класть свои вещи там, где удобнее, особенно если все остальные перед вами делали то же самое.

Я без энтузиазма отношусь к решению с `beforeEach()`. Давайте посмотрим, нельзя ли устраниТЬ некоторые из этих проблем, сохраняя дублирование на минимальном уровне.

2.7. РЕШЕНИЕ С ФАБРИЧНЫМ МЕТОДОМ

Фабричные методы представляют собой простые вспомогательные функции для построения объектов или специальных состояний, повторно использующие одну логику в разных местах. Возможно, нам удастся устраниТЬ часть дублирующеГося громоздкого кода, используя несколько фабричных методов для правил проходящих и непроходящих тестов в листинге 2.16.

Листинг 2.16. Добавление фабричных методов

```
describe('PasswordVerifier', () => {
  let verifier;
  beforeEach(() => verifier = new PasswordVerifier1());
  describe('with a failing rule', () => {
    let errors;
    beforeEach(() => {
      verifier.addRule(makeFailingRule('fake reason'));
      errors = verifier.verify('any value');
    });
    it('has an error message based on the rule.reason', () => {
      expect(errors[0]).toContain('fake reason');
    });
    it('has exactly one error', () => {
      expect(errors.length).toBe(1);
    });
  });
  describe('with a passing rule', () => {
    let errors;
```

```

beforeEach(() => {
  verifier.addRule(makePassingRule());
  errors = verifier.verify('any value');
});
it('has no errors', () => {
  expect(errors.length).toBe(0);
});
});
describe('with a failing and a passing rule', () => {
let errors;
beforeEach(() => {
  verifier.addRule(makePassingRule());
  verifier.addRule(makeFailingRule('fake reason'));
  errors = verifier.verify('any value');
});
it('has one error', () => {
  expect(errors.length).toBe(1);
});
it('error text belongs to failed rule', () => {
  expect(errors[0]).toContain('fake reason');
});
});
. .
const makeFailingRule = (reason) => {
  return (input) => {
    return { passed: false, reason: reason };
  };
};
const makePassingRule = () => (input) => {
  return { passed: true, reason: '' };
});
})
)

```

Фабричные методы `makeFailingRule()` и `makePassingRule()` делают функции `beforeEach()` чуть более понятными.

2.7.1. Полная замена `beforeEach()` фабричными методами

Что, если мы вообще откажемся от использования `beforeEach()` для инициализации? Что, если переключиться на использование небольших фабричных методов? Давайте посмотрим, что из этого получится.

Листинг 2.17. Замена `beforeEach()` фабричными методами

```

const makeVerifier = () => new PasswordVerifier1();
const passingRule = (input) => ({passed: true, reason: ''});

const makeVerifierWithPassingRule = () => {
  const verifier = makeVerifier();
  verifier.addRule(passingRule);
  return verifier;
};

```

```

const makeVerifierWithFailedRule = (reason) => {
  const verifier = makeVerifier();
  const fakeRule = input => ({passed: false, reason: reason});
  verifier.addRule(fakeRule);
  return verifier;
};

describe('PasswordVerifier', () => {
  describe('with a failing rule', () => {
    it('has an error message based on the rule.reason', () => {
      const verifier = makeVerifierWithFailedRule('fake reason');
      const errors = verifier.verify('any input');
      expect(errors[0]).toContain('fake reason');
    });
    it('has exactly one error', () => {
      const verifier = makeVerifierWithFailedRule('fake reason');
      const errors = verifier.verify('any input');
      expect(errors.length).toBe(1);
    });
  });
  describe('with a passing rule', () => {
    it('has no errors', () => {
      const verifier = makeVerifierWithPassingRule();
      const errors = verifier.verify('any input');
      expect(errors.length).toBe(0);
    });
  });
  describe('with a failing and a passing rule', () => {
    it('has one error', () => {
      const verifier = makeVerifierWithFailedRule('fake reason');
      verifier.addRule(passingRule);
      const errors = verifier.verify('any input');
      expect(errors.length).toBe(1);
    });
    it('error text belongs to failed rule', () => {
      const verifier = makeVerifierWithFailedRule('fake reason');
      verifier.addRule(passingRule);
      const errors = verifier.verify('any input');
      expect(errors[0]).toContain('fake reason');
    });
  });
});

```

По длине это решение получается примерно таким же, как в листинге 2.16, но, на мой взгляд, код намного легче читается, а следовательно, проще в сопровождении. Мы убрали функции `beforeEach()`, но без ущерба для сопровождаемости. Объем устраниенного дублирования пренебрежимо мал, но читабельность заметно улучшилась из-за удаления вложенных блоков `beforeEach()`.

Кроме того, сокращается риск утомительной прокрутки. Мне как читателю теста не нужно прокручивать файл вверх и вниз, чтобы узнать, когда объект создается или объявляется. Всю информацию можно извлечь прямо из `it()`. Мне не нужно

знать, как что-то создается, но мы знаем, где это происходит и с какими важными параметрами инициализируется. Все явно объясняется в коде.

При необходимости я могу углубиться в конкретные фабричные методы, и мне нравится, что каждый вызов `it()` инкапсулирует свое состояние. Вложенная структура `describe()` помогает понять, где вы находитесь, но состояние инициируется исключительно из блоков `it()`, а не за их пределами.

2.8. ВОЗВРАЩЕНИЕ К TEST()

Тесты в листинге 2.17 достаточно «самоинкапсулированы», так что блоки `describe()` действуют только как синтаксический сахар — удобство, упрощающее понимание. Теперь необходимость в них отпадает. При желании тесты можно записать так, как показано в листинге 2.18.

Листинг 2.18. Удаление вложенных describe

```
test('pass verifier, with failed rule, ' +
    'has an error message based on the rule.reason', () => {
  const verifier = makeVerifierWithFailedRule('fake reason');
  const errors = verifier.verify('any input');
  expect(errors[0]).toContain('fake reason');
});
test('pass verifier, with failed rule, has exactly one error', () => {
  const verifier = makeVerifierWithFailedRule('fake reason');
  const errors = verifier.verify('any input');
  expect(errors.length).toBe(1);
});
test('pass verifier, with passing rule, has no errors', () => {
  const verifier = makeVerifierWithPassingRule();
  const errors = verifier.verify('any input');
  expect(errors.length).toBe(0);
});
test('pass verifier, with passing and failing rule,' +
    ' has one error', () => {
  const verifier = makeVerifierWithFailedRule('fake reason');
  verifier.addRule(passingRule);
  const errors = verifier.verify('any input');
  expect(errors.length).toBe(1);
});
test('pass verifier, with passing and failing rule,' +
    ' error text belongs to failed rule', () => {
  const verifier = makeVerifierWithFailedRule('fake reason');
  verifier.addRule(passingRule);
  const errors = verifier.verify('any input');
  expect(errors[0]).toContain('fake reason');
});
```

Фабричные методы предоставляют всю необходимую функциональность без потери ясности каждого конкретного теста.

Мне нравится лаконичность листинга 2.18. Код легко понять. Возможно, здесь отчасти теряется структурная ясность, поэтому в одних ситуациях я выбираю решение без `describe`, а в других код лучше читается с вложенными `describe`. Вероятно, оптимальное сочетание простоты сопровождения и читабельности для вашего проекта лежит где-то между этими двумя подходами.

2.9. РЕФАКТОРИНГ: ПАРАМЕТРИЗОВАННЫЕ ТЕСТЫ

Попробуем отказаться от использования класса `Verifier` для работы по созданию и тестированию новых нестандартных правил. В листинге 2.19 приведено простое правило для присутствия хотя бы одной буквы верхнего регистра (я понимаю, что пароли с такими требованиями уже не считаются особо надежными, но для демонстрационных целей сойдет).

Листинг 2.19. Правила для паролей

```
const oneUpperCaseRule = (input) => {
  return {
    passed: (input.toLowerCase() !== input),
    reason: 'at least one upper case needed'
  };
};
```

Можно написать пару тестов так, как показано в листинге 2.20.

Листинг 2.20. Тестирование правила с вариациями

```
describe('one uppercase rule', function () {
  test('given no uppercase, it fails', () => {
    const result = oneUpperCaseRule('abc');
    expect(result.passed).toEqual(false);
  });
  test('given one uppercase, it passes', () => {
    const result = oneUpperCaseRule('Abc');
    expect(result.passed).toEqual(true);
  });
  test('given a different uppercase, it passes', () => {
    const result = oneUpperCaseRule('aBc');
    expect(result.passed).toEqual(true);
  });
});
```

В листинге 2.20 я продемонстрировал дублирование, которое может возникнуть при отработке одного сценария с небольшими вариациями во вводе единицы работы. В данном случае мы хотим убедиться в том, что позиция буквы верхнего регистра неважна — важно лишь ее наличие. Но это дублирование будет мешать позднее, если вы захотите изменить логику верхнего регистра или вам понадобится как-то изменить утверждения для этого сценария использования.

Параметризованные тесты в JavaScript могут создаваться разными способами, и в Jest предусмотрен один встроенный вариант: `test.each` (также существует синоним `it.each`). В листинге 2.21 показано, как использовать эту возможность для удаления дублирования из тестов.

Листинг 2.21. Использование `test.each`

```
describe('one uppercase rule', () => {
  test('given no uppercase, it fails', () => {
    const result = oneUpperCaseRule('abc');
    expect(result.passed).toEqual(false);
  });

  test.each(['Abc',
            'aBc'])
    ('given one uppercase, it passes', (input) => {
      const result = oneUpperCaseRule(input);
      expect(result.passed).toEqual(true);
    });
});
```

В этом примере тест будет повторен для каждого значения в массиве. Поначалу этот способ кажется слишком пространным, но, опробовав его, вы оцените простоту его использования. Кроме того, он хорошо читается.

Если вы хотите передать несколько параметров, заключите их в массив, как в листинге 2.22.

Листинг 2.22. Рефакторинг `test.each`

```
describe('one uppercase rule', () => {
  test.each([
    ['Abc', true],
    ['aBc', true],
    ['abc', false]
  ])('given %s, %s ', (input, expected) => {
    const result = oneUpperCaseRule(input);
    expect(result.passed).toEqual(expected);
  });
});
```

Впрочем, нам даже не обязательно использовать Jest. JavaScript достаточно гибок, чтобы при желании мы могли легко создать собственный параметризованный тест.

Листинг 2.23. Использование цикла `for` в базовом JavaScript

```
describe('one uppercase rule, with vanilla JS for', () => {
  const tests = {
    'Abc': true,
    'aBc': true,
```

```
'abc': false,
};

for (const [input, expected] of Object.entries(tests)) {
  test(`given ${input}, ${expected}`, () => {
    const result = oneUpperCaseRule(input);
    expect(result.passed).toEqual(expected);
  });
}
});
```

Решайте сами, какой вариант выбрать (я предпочитаю более простые решения и использую `test.each`). Суть в том, что Jest — всего лишь один из возможных инструментов. Паттерн параметризованных тестов может быть реализован разными способами. Он обладает большой силой, но и требует большой ответственности. Очень легко злоупотребить им и создать тесты, более сложные для понимания.

Обычно я стараюсь следить за тем, чтобы в каждом тесте отрабатывался только один сценарий (тип ввода). Если бы я анализировал этот тест в ходе реview, то сказал бы человеку, написавшему этот код, что в действительности тест проверяет два разных сценария: *без символа верхнего регистра* и пару примеров *с одним символом верхнего регистра*. Я бы предпочел разбить его на два разных теста.

В данном примере я хотел показать, что очень просто избавиться от большого количества тестов и поместить их все в одну большую конструкцию `test.each` — даже если это вредит читабельности, так что будьте осторожны в этой рискованной ситуации.

2.10. ПРОВЕРКА ОЖИДАЕМЫХ ОШИБОК

Иногда требуется спроектировать блок кода, который выдает ошибку в нужный момент с нужными данными. Что произойдет, если добавить в функцию `verify` код, который выдает ошибку при отсутствии настроенных правил, как в листинге 2.24?

Листинг 2.24. Выдача ошибки

```
verify (input) {
  if (this.rules.length === 0) {
    throw new Error('There are no rules configured');
  }
  ...
}
```

Код можно протестировать традиционным способом с использованием `try/catch`, чтобы *при отсутствии* ошибки тест не проходил.

Листинг 2.25. Тестирование исключений с try/catch

```
test('verify, with no rules, throws exception', () => {
  const verifier = makeVerifier();
  try {
    verifier.verify('any input');
    fail('error was expected but not thrown');
  } catch (e) {
    expect(e.message).toContain('no rules configured');
  }
});
```

Использование fail()

С технической точки зрения вызов `fail()` входит в устаревший API от исходного ответвления Jasmine, на базе которого создавался Jest. Этот вызов позволяет инициировать провал теста, но он не упоминается в официальной документации Jest API, и вместо него рекомендуется использовать `expect.assertions(1)`. Если ожидаемая секция `catch()` не достигается, то тест не проходит. Я обнаружил, что пока `fail()` все еще работает, этот вызов хорошо подходит для моих целей — продемонстрировать, почему вам не следует использовать конструкцию `try/catch` в юнит-тестах, если это вообще возможно.

Паттерн `try/catch` работает эффективно, но решение получается слишком длинным, и полностью вводить его с клавиатуры не хочется. Jest, как и большинство других фреймворков, предоставляет сокращенную запись для реализации именно таких сценариев — `expect().toThrowError()`.

Листинг 2.26. Использование expect().toThrowError()

```
test('verify, with no rules, throws exception', () => {
  const verifier = makeVerifier();
  expect(() => verifier.verify('any input'))
    .toThrowError(/no rules configured/);
});
```

Применение регулярного выражения вместо просмотра строки

Обратите внимание: я использую регулярное выражение для проверки того, что строка ошибки *содержит* заданную строку, а не равна ей; таким образом повышается надежность теста на случай, если строка изменится по краям. `toThrowError` существует в нескольких вариантах, информация о которых доступна по адресу <https://jestjs.io/>.

Снапшоты в Jest

В Jest существует уникальная функциональность снапшотов (Snapshots). Она позволяет отрендерить компонент (при работе в таком фреймворке, как React), а затем сопоставить текущий рендеринг с сохраненным снапшотом этого компонента, включая все его свойства и HTML. Я не стану надолго задерживаться на этой функциональности, но из того, что я видел, этой функцией часто серьезно злоупотребляют. С ней можно создать малопонятные тесты, которые выглядят примерно так:

```
it('renders', ()=>{
  expect(<MyComponent/>).toMatchSnapshot();
});
```

Здесь трудно понять, что именно тестируется, и к тому же тестируются сразу несколько сущностей, которые могут быть не связаны друг с другом. Кроме того, проверка может не пройти по многим причинам, которые не представляют для вас интереса, так что со временем удобство сопровождения этого теста будет снижаться. Наконец, это отличный повод не писать читабельные и простые в сопровождении тесты, потому что сроки поджимают, но при этом вам все равно придется показать тесты. Да, этот тест нельзя назвать бесполезным, но его легко использовать в местах, где другие виды тестов будут более подходящими.

Если вам нужна вариация на эту тему, попробуйте использовать `toMatchInlineSnapshot()`. Дополнительную информацию можно найти по адресу <https://jestjs.io/docs/en/snapshot-testing>.

2.11. НАЗНАЧЕНИЕ КАТЕГОРИЙ ТЕСТОВ

Если вам нужно запустить одну конкретную категорию тестов (например, только юнит-тесты, только интеграционные или только тесты, относящиеся к конкретной части приложения), то в настоящее время Jest не обладает возможностью определять категории тестовых случаев.

Впрочем, не все потеряно. У Jest имеется специальный флаг командной строки `--testPathPattern`, при помощи которого можно определить, как Jest будет искать тесты. Эту команду можно запустить с другим путем для конкретного типа тестов, которые вы хотите выполнить (например, «все тесты из каталога `integration`»). Полную информацию можно найти по адресу <https://jestjs.io/docs/en/cli>.

Другая альтернатива — создание отдельного файла `jest.config.js` для каждой категории тестов, содержащего собственную конфигурацию `testRegex` и другие свойства .

Листинг 2.27. Создание отдельных файлов `jest.config.js`

```
// jest.config.integration.js
var config = require('./jest.config')
config.testRegex = "integration\\\\.js$"
module.exports = config

// jest.config.unit.js
var config = require('./jest.config')
config.testRegex = "unit\\\\.js$"
module.exports = config
```

Также для каждой категории можно создать отдельный скрипт `прем`, который активирует командную строку Jest с нестандартным конфигурационным файлом: `jest -c my.custom.jest.config.js`.

Листинг 2.28. Использование нестандартных скриптов `прем`

```
//Package.json
...
"scripts": {
    "unit": "jest -c jest.config.unit.js",
    "integ": "jest -c jest.config.integration.js"
}
...
```

В следующей главе будет рассмотрен код с зависимостями и проблемами тестируемости и мы перейдем к обсуждению концепций фейков, шпионов, моков и стабов и узнаем, как с их помощью писать тесты для такого кода.

ИТОГИ

- Jest — популярный тестовый фреймворк с открытым кодом для приложений JavaScript. Он одновременно действует как *библиотека для тестирования*, используемая для написания тестов, *библиотека утверждений* для выполнения проверок в тестах, *инструмент для запуска тестов* и *подсистема вывода отчетов*.
- AAA (*Arrange – Act – Assert*) — популярный паттерн для структурирования тестов. Он предоставляет простую однородную структуру для всех тестов. После того как вы привыкнете к этой структуре, сможете легко читать и понимать любые тесты.
- В паттерне AAA в разделе *подготовки* тестируемая система и ее зависимости переводятся в нужное состояние. В разделе *действия* вы вызываете методы,

передаете подготовленные зависимости и сохраняете выходное значение (если оно есть). В разделе *проверки* проверяется результат.

- Хорошие имена тестов содержат информацию о тестируемой единице работы, сценарии или точке входа единицы работы, а также об ожидаемом поведении или точке выхода. Для обозначения такой схемы назначения имен существует удобная мнемоника USE (Unit, Scenario, Expectation).
- Jest предоставляет ряд функций для формирования структуры вокруг нескольких взаимосвязанных тестов. `describe()` — функция области действия, которая позволяет сгруппировать несколько тестов (или группы тестов). Хорошая метафора для `describe` — каталог, содержащий тесты или другие каталоги. `test()` — функция, обозначающая отдельный тест. `it()` — синоним для `test()`, но он лучше читается при использовании с `describe()`.
- `beforeEach()` помогает избежать дублирования за счет выделения кода, общего для вложенных функций `describe` и `it`.
- Использование `beforeEach()` часто приводит к необходимости утомительной прокрутки, когда вам приходится в разных местах кода искать, что делает тест.
- *Фабричные методы* с простыми тестами (без `beforeEach()`) улучшают читабельность и помогают избежать утомительной прокрутки.
- *Параметризованные тесты* уменьшают объем кода, необходимого для схожих тестов. Их недостаток — в процессе обобщения тесты начинают хуже читаться.
- Чтобы сохранить баланс между читабельностью теста и возможностями повторного использования кода, параметризуйте только входные значения. Создайте отдельные тесты для разных выходных значений.
- Jest не поддерживает классификацию тестов по категориям, но вы можете запускать группы тестов при помощи флага `--testPathPattern`. Также можно настроить значение `testRegex` в конфигурационном файле.

Часть 2

Приемы и методы

Мы описали основы в части 1, а теперь перейдем к приемам тестирования и рефакторинга, необходимым для написания тестов при реальной разработке.

В главе 3 рассматриваются стабы¹ (stubs) и их применение для устранения зависимостей. Мы поговорим о приемах рефакторинга, которые упрощают тестирование кода, и возможных препятствиях, которые могут мешать процессу.

В главе 4 мы перейдем к мокам² (mock objects) и тестированию взаимодействий; расскажем, чем моки отличаются от стабов, и исследуем концепцию фейков.

В главе 5 рассказывается о фреймворках для изоляции тестов и их применении для решения некоторых проблем повторения кода при ручном написании моков и стабов. В главе 6 речь пойдет об асинхронном коде: промисах, таймерах и событиях, а также о различных методах тестирования такого кода.

¹ Их также называют «заглушки». — Примеч. ред.

² También называют «имитации». — Примеч. ред.

Устранение зависимостей при помощи стабов

В ЭТОЙ ГЛАВЕ

- ✓ Типы зависимостей — моки, стабы и т. д.
- ✓ Причины для использования стабов
- ✓ Функциональные методы внедрения
- ✓ Модульные методы внедрения
- ✓ Объектно-ориентированные методы внедрения

В предыдущей главе вы написали свой первый юнит-тест с использованием Jest, и мы внимательнее присмотрелись к тому, удобно ли сопровождать такой тест. Сценарий был довольно простым и, что еще важнее, полностью автономным. Password Verifier не зависел от внешних модулей, и мы могли сосредоточиться на его функциональности, не беспокоясь о других факторах, которые могут вмешаться в его работу.

В той же главе мы использовали первые два типа точек выхода для наших примеров: точки выхода с возвращаемыми значениями и точки выхода на основе состояния. В этой главе рассматривается последняя разновидность — *сторонний вызов*. Кроме того, в главе будет представлено новое требование — зависимость кода от времени. Мы рассмотрим два разных подхода к реализации этого

требования: рефакторинг кода и *исправление ошибок во время исполнения программы* (*monkey-patching*) без рефакторинга.

Зависимость от внешних модулей или функций может усложнить (и усложнит!) написание тестов и обеспечение их повторяемости, а также сделает их ненадежными.

Внешние сущности, на которые мы полагаемся в своем коде, называются *зависимостями* (*dependencies*). Они будут более подробно определены позднее в этой главе. К зависимостям могут относиться такие факторы, как время, асинхронное выполнение, использование файловой системы или сети или просто нечто, что очень сложно настраивается либо выполняется в течение очень долгого времени.

3.1. ТИПЫ ЗАВИСИМОСТЕЙ

По моему опыту существуют два основных типа зависимостей, которые могут использоваться нашей единицей работы.

- *Исходящие зависимости* — зависимости, представляющие собой точку выхода нашей единицы работы: вызов логгера, сохранение информации в базе данных, отправка электронной почты, уведомление API или веб-перехватчика о произошедшем событии и т. д. Все эти термины — «вызов», «отправка», «уведомление» — подразумевают действия, направленные *наружу* из нашей единицы работы по сценарию «выстрелил и забыл». Каждое из них соотносится с точкой выхода или конечной точкой конкретного логического потока в единице работы.
- *Входящие зависимости* — зависимости, не являющиеся точками выхода. Они не содержат требований к итоговому поведению единицы работы и существуют только для того, чтобы представить единице работы специализированные данные или поведение для конкретного теста: результат запроса к базе данных, содержимое файла в файловой системе, сетевой ответ и т. д. Все они представляют собой пассивные блоки данных, которые заходят *внутрь* единицы работы как результат предыдущей операции.

На рис. 3.1 представлены эти виды зависимостей.

Некоторые зависимости могут быть одновременно входящими и исходящими — в одних тестах они представляют точки выхода, а в других используются для моделирования данных, входящих в приложение. Такие ситуации встречаются довольно редко, но они существуют — например, во внешнем API, который возвращает признак успеха/неудачи для исходящего сообщения.

Помня об этих типах зависимостей, посмотрим, как в книге «*xUnit Test Patterns*» определяются разные паттерны для сущностей, которые в тестах должны выглядеть как другие сущности.

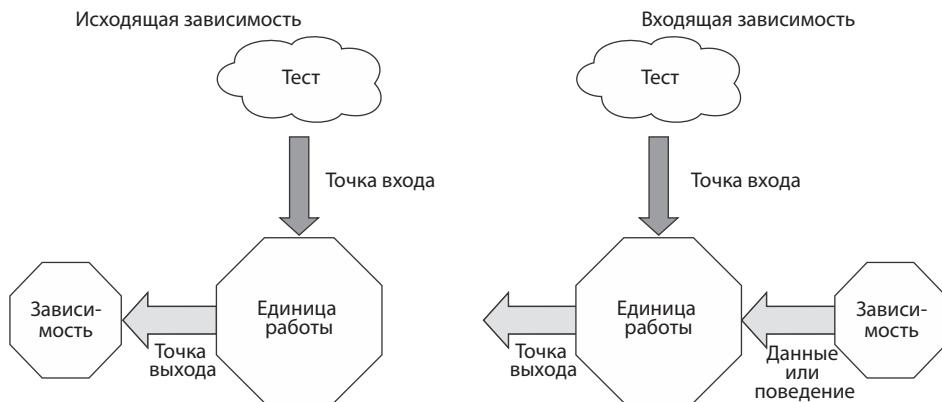


Рис. 3.1. Слева: точка выхода реализована как активация зависимости.
Справа: зависимость предоставляет непрямой ввод или поведение и не является точкой выхода

В таблице 3.1 я привожу свою трактовку некоторых паттернов с веб-сайта книги по адресу <http://mng.bz/n1WK>.

Другая возможная точка зрения на эти термины для оставшейся части книги:

- *стабы* разбивают входящие зависимости (непрямые вводы). Стабы представляют собой фиктивные модули, объекты или функции, поставляющие фиктивное поведение или данные в тестируемый код. Они не используются для проверок. В одном тесте может быть много стабов;
- *моки* разбивают исходящие зависимости (неявные выводы или точки выхода). Моки представляют собой фиктивные модули, объекты или функции, вызов которых мы проверяем в своих тестах. Мок представляет собой *точку выхода* в юнит-тесте. По этой причине рекомендуется иметь не более одного мока на тест.

К сожалению, во многих обсуждениях приходится слышать, что термин «мок» используется как универсальное обозначение как для стабов, так и для моков. Фразы типа «создадим для этого мок» или «у нас есть мок-база данных» могут создать путаницу. Между стабами и моками существуют огромные различия (моки должны встречаться не более одного раза в тесте), и вам следует использовать правильные термины, чтобы ясно показать, о чем идет речь.

Если сомневаетесь, используйте термин «тестовый дублер» или «фейк». Часто одна фейковая зависимость может применяться как стаб в одном тесте и как мок в другом тесте. Пример такого рода будет представлен позднее.

Тестовые паттерны xunit и выбор имен

Книга «xUnit Test Patterns: Refactoring Test Code» Джерарда Месароша — классический справочник паттернов юнит-тестирования. В ней определяются паттерны для фейков, используемых в вашем коде, как минимум пятью способами. Когда вы освоитесь с тремя типами, которые рассматриваются здесь, я рекомендую ознакомиться с дополнительными подробностями, приведенными в книге Месароша.

Заметим, что в «xUnit Test Patterns»дается следующее определение термина «фейк»: «Замена компонента, от которого зависит тестируемая система (SUT), гораздо более легковесной реализацией». Например, можно использовать базу данных в памяти вместо полноценного рабочего экземпляра.

Я же отношу эту разновидность тестовых двойников к стабам и использую термин «фейк» для всего, что реально не существует, практически в том же смысле, что и термин «тестовый дублер», но слово «фейк» короче и проще произносится.

Таблица 3.1. Прояснение терминологии относительно стабов и моков

| Категория | Паттерн | Предназначение | Применение |
|-----------|-------------------------------|---|---|
| | Тестовый дублер (test double) | Общее название для стабов и моков | Я также использую термин «фейк» |
| Стаб | Объект-пустышка (dummy) | Используется для определения значений, которые должны применяться в тестах только как неактуальные аргументы или вызовы методов SUT | Передайте как параметр точке входа или используйте в части подготовки в паттерне AAA |
| | Тестовый стаб (test stub) | Используется для независимой проверки логики, зависящей от непрямого ввода со стороны других программных компонентов | Внедрите как зависимость и настройте для возврата конкретных значений или поведения в SUT |
| Мок | Тестовый шпион (test spy) | Используется для независимой проверки логики, имеющей непрямые выводы в другие программные компоненты | Переопределите одну функцию для реального объекта и убедитесь в том, что фейковая функция была вызвана, как ожидалось |

Таблица 3.1 (продолжение)

| Категория | Паттерн | Предназначение | Применение |
|-----------|--------------------------|---|---|
| | Объект-мок (mock object) | Используется для независимой проверки логики, зависящей от непрямых выводов в другие программные компоненты | Внедрите фейк как зависимость в SUT и убедитесь в том, что фейк был вызван, как ожидалось |

Может показаться, что я привел слишком много информации за раз. Эти определения будут глубже исследованы дальше в этой главе. А пока начнем с малого — со *стабов*.

3.2. ПРИЧИНЫ ДЛЯ ИСПОЛЬЗОВАНИЯ СТАБОВ

Что, если вы столкнулись с задачей тестирования фрагмента кода, приведенного в листинге 3.1?

Листинг 3.1. verifyPassword с использованием времени

```
const moment = require('moment');
const SUNDAY = 0, SATURDAY = 6;

const verifyPassword = (input, rules) => {
    const dayOfWeek = moment().day();
    if ([SATURDAY, SUNDAY].includes(dayOfWeek)) {
        throw Error("It's the weekend!");
    }
    // ...
    // возвращает список найденных ошибок...
    return [];
};
```

В нашей системе проверки пароля появляется новая зависимость: система не может работать по выходным. Почему? Кто его знает... Говоря конкретнее, в модуле появляется прямая зависимость от `moment.js` — очень распространенной обертки даты/времени для JavaScript. Прямая работа с датами в JavaScript — занятие на любителя, и мы можем предположить, что во многих организациях используется похожий код.

Как прямое использование библиотеки для работы со временем влияет на наши юнит-тесты? Проблема в том, что эта прямая зависимость заставляет тесты принимать во внимание правильную дату и время, причем у нас нет прямой возможности влиять на дату и время в тестируемом приложении. В листинге 3.2 показан неудачный тест, который может выполняться только в выходные.

Листинг 3.2. Исходная версия юнит-тестов для verifyPassword

```

const moment = require('moment');
const {verifyPassword} = require("./password-verifier-time00");
const SUNDAY = 0, SATURDAY = 6, MONDAY = 2;

describe('verifier', () => {
    const TODAY = moment().day();

    //тест всегда выполняется, но может ничего не делать
    test('on weekends, throws exceptions', () => {
        if ([SATURDAY, SUNDAY].includes(TODAY)) { ←
            expect(()=> verifyPassword('anything',[]))
                .toThrow("It's the weekend!"); } } );
    //в будни тест даже не запускается
    if ([SATURDAY, SUNDAY].includes(TODAY)) { ←
        test('on a weekend, throws an error', () => {
            expect(()=> verifyPassword('anything',[]))
                .toThrow("It's the weekend!"); } );
    } );
});

```

Приведенный листинг включает две вариации одного теста. Одна проверяет текущую дату *внутри* теста, а в другой проверка выполняется *за пределами* теста, что означает, что тест вообще не выполняется в будние дни, а только по выходным. И это плохо.

Вернемся к одному из положительных качеств тестов, упоминавшихся в главе 1, — стабильности: каждый раз, когда я запускаю тест, это *тот же самый тест*, который выполнялся ранее. Используемые значения не изменяются. Проверки не изменяются. Если код не изменился (код тестов или рабочий код), то тест должен выдать точно такой же результат, как при предыдущих запусках.

Второй тест иногда даже не запускается. Уже этого достаточно, чтобы использовать фейк для устранения зависимости. Более того, мы не можем смоделировать выходной или будний день, что становится более чем убедительным стимулом для переработки тестируемого кода, чтобы он чуть лучше подходил для внесения зависимостей.

Но постойте, это еще не все. Тесты, использующие время, часто бывают ненадежными (нестабильными, flaky). Они могут не проходить в отдельных случаях, когда не меняется ничего, кроме времени. Этот тест — типичный кандидат для такого поведения, потому что при его локальном запуске мы получаем обратную связь только по *одному* из двух состояний. Если вы хотите узнать, как он ведет себя по выходным, просто подождите пару дней. Н-да.

Тесты могут стать ненадежными из-за граничных случаев, которые влияют на переменные, неподконтрольные нам в тесте. Типичные примеры — сетевые проблемы, возникающие в ходе сквозного тестирования, проблемы с подключением к базам данных или различные проблемы с серверами. Когда тест не проходит, этот факт легко отбросить со словами «просто запустите его снова» или «все в порядке, это просто [вставить какую-нибудь проблему с изменяемостью]».

3.3. ОБЩЕПРИНЯТЫЕ ПОДХОДЫ К СОЗДАНИЮ СТАБОВ

В следующих разделах поговорим о некоторых распространенных формах внедрения стабов в единицы работы. Сначала мы обсудим базовую параметризацию как первый шаг, а затем перейдем к рассмотрению следующих методов.

- *Функциональные подходы:*
 - функция как параметр;
 - частичное применение (каррирование);
 - фабричные функции;
 - функции-конструкторы.
- *Модульный подход:*
 - внедрение модулей.
- *Объектно-ориентированные подходы:*
 - внедрение через конструктор класса;
 - объект как параметр («утиная типизация»);
 - общий интерфейс как параметр (для этого мы воспользуемся TypeScript).

Мы рассмотрим каждый из этих подходов, начиная с простого случая контроля времени в своих тестах.

3.3.1. Создание стаба для времени с внедрением параметра

Я могу представить как минимум две веские причины для контроля за временем в ситуации, описанной ранее:

- чтобы устраниТЬ изменчивости в наших тестах;
- чтобы упростить моделирование любого сценария, связанного со временем, для которого нам хотелось бы протестировать свой код.

Простейший рефакторинг, который я только могу придумать, несколько улучшает повторяемость теста. Добавим в функцию параметр `currentDay` для задания

текущей даты. Тем самым мы устраним необходимость использования модуля `moment.js` в нашей функции, и эта ответственность будет переложена на сторону вызова функции. Тогда в наших тестах время можно будет определять жестко фиксированным способом, а тест и функция станут повторяемыми и стабильными. В листинге 3.3 приведен пример такого рефакторинга.

Листинг 3.3. `verifyPassword` с параметром `currentDay`

```
const verifyPassword2 = (input, rules, currentDay) => {
  if ([SATURDAY, SUNDAY].includes(currentDay)) {
    throw Error("It's the weekend!");
  }
  // ...
  // возвращает список найденных ошибок...
  return [];
};

const SUNDAY = 0, SATURDAY = 6, MONDAY = 1;
describe('verifier2 - dummy object', () => {
  test('on weekends, throws exceptions', () => {
    expect(() => verifyPassword2('anything',[],SUNDAY ))
      .toThrow("It's the weekend!");
  });
});
```

Добавляя параметр `currentDay`, мы фактически передаем контроль за временем на сторону вызова функции (наш тест). То, что здесь внедряется, формально называется *пустышкой* (*dummy*) — это просто данные без поведения, но с этого момента мы можем называть ее «стабом».

Этот подход является разновидностью *инверсии зависимостей* (dependency inversion). Похоже, термин «инверсия зависимостей» впервые появился в статье Джонсона и Фута (Johnson and Foote) «Designing Reusable Classes», опубликованной в журнале «Journal of Object-Oriented Programming» в 1988 году. Термин «инверсия зависимостей» также является одним из паттернов концепции SOLID, описанной Робертом Мартином в статье «Design Principles and Design Patterns» в 2000 г. Высокоуровневые аспекты проектирования будут более подробно рассмотрены в главе 8.

Добавление этого параметра — простой, но весьма эффективный рефакторинг. Кроме стабильности, он предоставляет еще пару преимуществ:

- теперь мы можем легко смоделировать любой день на свое усмотрение;
- тестируемый код не отвечает за импортирование времени, поэтому у него будет одной причиной меньше для изменений в случае перехода на другую библиотеку времени.

Здесь происходит «внедрение зависимости» для времени в нашу единицу работы. Структура точки входа изменяется, чтобы значение дня передавалось

в параметре. Функция стала «чистой» по стандартам функционального программирования в том смысле, что она не имеет побочных эффектов. В чистых функциях встроены внедрения всех их зависимостей — это одна из причин, по которым структуры функционального программирования обычно намного проще тестируются.

Может показаться немного странным, что параметр `currentDay` называется стабом, хотя это обычное целочисленное значение. Но на основании определений из «xUnit Test Patterns» можно сказать, что это «пустышка», а с моей точки зрения, эта разновидность относится к категории стабов. Чтобы быть стабом, необязательно быть чем-то сложным. Просто значение должно находиться под вашим контролем. Это стаб, потому что мы используем его для моделирования некоторого ввода или поведения, передаваемого тестируемому юниту. На рис. 3.2 эта ситуация представлена наглядно.

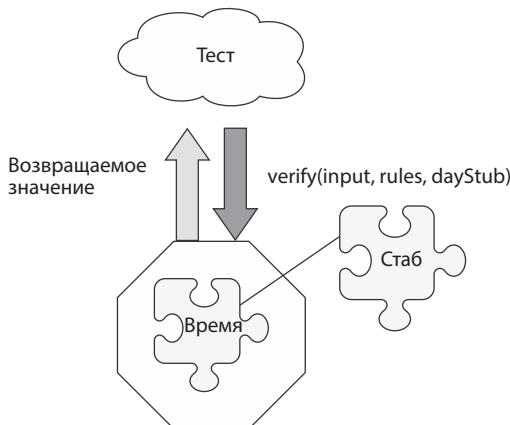


Рис. 3.2. Внедрение стаба для зависимости от времени

3.3.2. Зависимости, внедрения и контроль

В таблице 3.2 снова перечислены некоторые важные термины, которые обсуждались ранее и будут использоваться в оставшейся части этой главы.

Таблица 3.2. Терминология, используемая в этой главе

| | |
|--------------------|--|
| Зависимости | Сущности, затрудняющие тестирование и ухудшающие сопровождаемость кода, так как мы не можем контролировать их из своих тестов. Примеры — время, файловая система, сеть, случайные значения и т. д. |
| Контроль | Возможность предписать поведение зависимости. Говорят, что создатели зависимостей <i>контролируют</i> их, потому что у них есть возможность настроить эти зависимости до того, как они будут использоваться в тестируемом коде |

| | |
|-------------------------------|---|
| | <p>В листинге 3.1 наш тест не контролирует время, потому что его контролирует тестируемый модуль. Модуль решил всегда использовать текущую дату и время. В результате тест всегда делает одно и то же, что приводит к потере стабильности в тестах.</p> <p>В листинге 3.3 мы получили доступ к зависимости, инвертируя контроль над ней через параметр <code>currentDay</code>. Теперь тест контролирует время и может принять решение об использовании фиксированного времени. Тестируемый модуль должен использовать предоставленное время, что существенно упрощает наш тест</p> |
| Инверсия контроля | Проектирование кода с устранением внутреннего создания зависимости и вывода его наружу. В листинге 3.3 представлен один из способов решения этой задачи, основанный на <i>внедрении параметра</i> |
| Внедрение зависимостей | Акт передачи зависимости через интерфейс для внутреннего использования в блоке кода. Место, в котором внедряется зависимость, называется <i>точкой внедрения</i> (<i>injection point</i>). В нашем случае используется точка внедрения параметра. Другой термин для обозначения места, в котором может происходить внедрение сущностей, — <i>шов</i> |
| Шов | <p>Термин был предложен Майклом Физерсом в книге «Working Effectively with Legacy Code».</p> <p>Шов (<i>seam</i>) представляет собой место, в котором стыкуются два программных компонента и в котором возможно внедрение. Это место, в котором вы можете изменить поведение своей программы без редактирования ее «на месте». Примеры — параметры, функции, загрузчики модулей, замена функций, в объектно-ориентированном мире — интерфейсы классов, открытые виртуальные методы и т. д.</p> |

Швы в рабочем коде играют важную роль для облегчения сопровождения и улучшения читабельности юнит-тестов. Чем проще изменять и внедрять поведение или специальные данные в тестируемый код, тем проще будет писать, читать, а позднее и сопровождать тест при изменении рабочего кода. Некоторые паттерны и антипаттерны, относящиеся к проектированию кода, рассматриваются в главе 8.

3.4. ФУНКЦИОНАЛЬНЫЕ МЕТОДЫ ВНЕДРЕНИЯ

На данный момент решение оставляет желать лучшего. Добавление параметра решило проблему зависимости на уровне функции, но теперь каждая вызывающая сторона должна знать, как работать с датой. Такое решение получится длиннее, чем хотелось бы.

JavaScript поддерживает два основных стиля программирования — функциональный и объектно-ориентированный, поэтому я буду представлять решения в обоих стилях там, где это имеет смысл, а вы сможете выбрать тот вариант, который лучше подходит для вашей ситуации.

Не существует единственно правильного подхода. Сторонники функционального программирования выступают за простоту, ясность и проверяемость функционального стиля, но его освоение потребует серьезных усилий. Уже по этой причине будет разумно изучить оба подхода, чтобы вы могли выбрать тот, который лучше годится для команды, с которой вы работаете. Некоторые команды склонны отдавать предпочтение объектно-ориентированным решениям, потому что с ними они чувствуют себя более уверенно. Другие предпочитают функциональные решения. На мой взгляд, паттерны остаются в основном неизменными; мы просто преобразуем их в разные стили.

3.4.1. Внедрение функции

В листинге 3.4 приведен другой рефакторинг той же задачи: вместо объекта данных в параметре передается функция, которая возвращает объект данных.

Листинг 3.4. Внедрение зависимостей через функцию

```
const verifyPassword3 = (input, rules, getDayFn) => {
  const dayOfWeek = getDayFn();
  if ([SATURDAY, SUNDAY].includes(dayOfWeek)) {
    throw Error("It's the weekend!");
  }
  // ...
  // возвращает список найденных ошибок...
  return [];
};
```

Соответствующий тест приведен в листинге 3.5.

Листинг 3.5. Тестирование с функциональным внедрением

```
describe('verifier3 - dummy function', () => {
  test('on weekends, throws exceptions', () => {
    const alwaysSunday = () => SUNDAY;
    expect(()=> verifyPassword3('anything',[], alwaysSunday))
      .toThrow("It's the weekend!");
  });
});
```

Отличия от предыдущего теста минимальны, но использование функции как параметра является допустимым способом реализации внедрения. В других сценариях оно также хорошо подходит для включения специального поведения — например, моделирования особых случаев или исключений в тестируемом коде.

3.4.2. Внедрение зависимости через частичное применение

Фабричными функциями или методами (подкатегория «функций высшего порядка») называются функции, которые возвращают другие функции, предварительно настроенные с некоторым контекстом. В нашем случае таким контекстом может быть список правил и функция текущего дня. Вы получаете обратно новую функцию, которая может вызываться со строковым вводом и использует правила и функцию `getDay()`, настроенную при ее создании.

Код в листинге 3.6 фактически преобразует фабричную функцию в часть подготовки (Arrange) теста и вызывает возвращенную функцию в части действия (Act). Вполне элегантно.

Листинг 3.6. Использование фабричной функции высшего порядка

```
const SUNDAY = 0, . . . FRIDAY=5, SATURDAY = 6;

const makeVerifier = (rules, dayOfWeekFn) => {
  return function (input) {
    if ([SATURDAY, SUNDAY].includes(dayOfWeekFn())) {
      throw new Error("It's the weekend!");
    }
    // ...
  };
};

describe('verifier', () => {
  test('factory method: on weekends, throws exceptions', () => {
    const alwaysSunday = () => SUNDAY;
    const verifyPassword = makeVerifier([], alwaysSunday);

    expect(() => verifyPassword('anything'))
      .toThrow("It's the weekend!");
  });
});
```

3.5. МОДУЛЬНЫЕ МЕТОДЫ ВНЕДРЕНИЯ

JavaScript также поддерживает концепцию *модулей*, которые используются с `import` или `require`. Как реализовать концепцию внедрения зависимостей при прямом импортировании зависимости в тестируемом коде, например коде из листинга 3.1, который снова приводится ниже для удобства?

```
const moment = require('moment');
const SUNDAY = 0; const SATURDAY = 6;

const verifyPassword = (input, rules) => {
  const dayOfWeek = moment().day();
  if ([SATURDAY, SUNDAY].includes(dayOfWeek)) {
```

```

        throw Error("It's the weekend!");
    }
    // ...
    // возвращает список найденных ошибок...
    return [];
};
```

Как справиться с возникающей прямой зависимостью? Ответ: никак. Необходимо по-другому написать код, чтобы позднее эту зависимость можно было заменить. Необходимо создать *шов*, через который мы сможем заменять зависимости. Один из примеров такого рода см. ниже.

Листинг 3.7. Абстрагирование необходимых зависимостей

```

const originalDependencies = {
    moment: require('moment'),           | moment.js упаковывается
};                                         | в промежуточный объект

let dependencies = { ...originalDependencies };   ← | Объект, содержащий текущую
                                                       | зависимость, реальную или
                                                       | фиктивную

const inject = (fakes) => {
    Object.assign(dependencies, fakes);   ← | Функция, которая заменяет
    return function reset() {             ← | реальную зависимость фиктивной
        dependencies = { ...originalDependencies };   ← | Функция, которая возвращается
                                                       | к реальной зависимости
    }
};

const SUNDAY = 0; const SATURDAY = 6;

const verifyPassword = (input, rules) => {
    const dayOfWeek = dependencies.moment().day();
    if ([SATURDAY, SUNDAY].includes(dayOfWeek)) {
        throw Error(`It's the weekend!`);
    }
    // ...
    // возвращает список найденных ошибок...
    return [];
};

module.exports = {
    SATURDAY,
    verifyPassword,
    inject
};
```

Что здесь происходит? Обратите внимание на три новшества.

- Сначала прямая зависимость от `moment.js` заменяется объектом: `originalDependencies`. Он содержит импортацию модуля как часть своей реализации.

- Затем добавляется еще один объект: `dependencies`. Он по умолчанию получает все реальные зависимости, содержащиеся в объекте `originalDependencies`.
- Наконец, функция `inject`, которая также предоставляется как часть нашего модуля, позволяет каждому, кто импортирует наш модуль (как рабочий код, так и тесты), переопределить реальные зависимости специальными зависимостями (фейками).

При вызове `inject` возвращается функция `reset`, которая восстанавливает исходные зависимости в текущей переменной `dependencies`; при этом любые фейки, используемые в настоящее время, сбрасываются.

В листинге 3.8 представлен пример использования функций `inject` и `reset` в teste.

Листинг 3.8. Внедрение фейкового модуля с `inject()`

```
const { inject, verifyPassword, SATURDAY } = require('../password-verifier-time00-modular');

const injectDate = (newDay) => {
  const reset = inject({
    moment: function () {
      //здесь мы создаем фейковый API вместо moment.js
      return {
        day: () => newDay
      }
    }
  });
  return reset;
};

describe('verifyPassword', () => {
  describe('when its the weekend', () => {
    it('throws an error', () => {
      const reset = injectDate(SATURDAY); ← Предоставляет
                                                фейковый день

      expect(() => verifyPassword('any input'))
        .toThrow("It's the weekend!");
      reset(); ← Сброс зависимости
    });
  });
});
```

Проанализируем, что здесь происходит.

1. `injectDate` — простая вспомогательная функция, уменьшающая объем шаблонного кода в нашем teste. Она всегда строит фейковую структуру API `moment.js` и настраивает ее функцию `getDay` так, чтобы она возвращала параметр `newDay`.

2. Функция `injectDate` вызывает `inject` с новым фейковым API `moment.js`. При этом фейковой зависимости нашей единицы работы назначается зависимость, переданная в параметре.
3. Наш тест вызывает функцию `inject` со специально выбранным фейковым днем.
4. В конце теста вызывается функция `reset`, которая сбрасывает модульные зависимости единицы работы и заменяет их исходными.

Когда вы проделаете это пару раз, все начинает обретать смысл. Впрочем, также необходимо сделать несколько оговорок. С одной стороны, этот метод определенно решает проблему зависимостей в тестах и относительно прост в использовании. С другой стороны, у него также есть один огромный недостаток. Использование этого метода для замены модульной зависимости фейками тесно связывает ваши тесты с сигнатурой API заменяемых зависимостей. Если существуют сторонние зависимости — такие как `moment.js`, логгеры и вообще все, что вы не можете полностью контролировать, — ваши тесты станут очень хрупкими, когда придет время обновить или заменить зависимости чем-то с другим API (а это время всегда приходит). С одним-двумя тестами все не так плохо, но обычно приходится иметь дело с сотнями и тысячами тестов, из которых нужно вынести несколько общих зависимостей, а это иногда требует изменения и исправления сотен файлов — например, при замене логгера с критическим изменением API.

Я могу предложить два возможных способа предотвращения таких ситуаций.

- Никогда не импортируйте стороннюю зависимость, которую вы не можете полностью контролировать в своем коде. Всегда используйте промежуточную абстракцию, которую можно контролировать. Хорошим примером служит архитектура «порты и адаптеры» (другие названия этой архитектуры — «гексагональная архитектура» и «луковая архитектура»). С этой архитектурой создание фейков для таких внутренних API менее рискованно, потому что мы можем контролировать темп их изменений и это снизит хрупкость тестов. (Их рефакторинг можно провести внутри, и это никак не отразится на ваших тестах, даже если изменится мир вокруг.)
- Страйтесь избегать модульного внедрения; вместо этого лучше использовать один из других способов внедрения зависимостей, упоминаемых в этой книге: параметры функций, каррирование, конструкторы и интерфейсы (о которых речь пойдет в следующем разделе). Все эти варианты предоставляют достаточно богатый выбор вместо прямого импортирования.

3.6. ПЕРЕХОД К ОБЪЕКТАМ С ФУНКЦИЯМИ-КОНСТРУКТОРАМИ

Функции-конструкторы позволяют достичь того же результата, что и фабричные функции (в чуть более объектно-ориентированном стиле JavaScript), но они возвращают нечто вроде объекта с методами, которые можно вызывать. Для вызова этой функции и получения такого специального объекта используется ключевое слово `new`.

В листинге 3.9 показано, как будут выглядеть тот же код и тесты в этом стиле.

Листинг 3.9. Использование функции-конструктора

```
const Verifier = function(rules, dayOfWeekFn)
{
    this.verify = function (input) {
        if ([SATURDAY, SUNDAY].includes(dayOfWeekFn())) {
            throw new Error("It's the weekend!");
        }
        // ...
    };
};

const {Verifier} = require("./password-verifier-time01");

test('constructor function: on weekends, throws exception', () => {
    const alwaysSunday = () => SUNDAY;
    const verifier = new Verifier([], alwaysSunday);

    expect(() => verifier.verify('anything'))
        .toThrow("It's the weekend!");
});
```

Вы можете посмотреть на все это и спросить: «А зачем переходить на объекты?» Ответ зависит от контекста текущего проекта, его технологического стека, знаний вашей команды в области функционального и объектно-ориентированного программирования, а также многих других нетехнических факторов. Полезно иметь в арсенале такой инструмент, чтобы воспользоваться им, когда возникнет необходимость. Помните об этом, читая несколько следующих разделов.

3.7. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ МЕТОДЫ ВНЕДРЕНИЯ

Если вы предпочитаете более объектно-ориентированный стиль или работаете на объектно-ориентированном языке (например, C# или Java), вот несколько распространенных паттернов, широко применяемых в объектно-ориентированном мире для внедрения зависимостей.

3.7.1. Внедрение через конструктор

Внедрение через конструктор — термин, которым я бы описал решение, когда зависимости могут внедряться через конструктор класса. В мире JavaScript Angular — самый известный фронтенд-фреймворк, использующий эту структуру для внедрения «сервисов», что просто является синонимом «зависимостей» в терминологии Angular. Это решение также может применяться во многих других ситуациях.

Наличие класса с состоянием имеет свои преимущества. Клиентам не нужно будет повторять одни и те же действия: достаточно настроить класс однократно, а затем многократно повторно использовать подготовленный класс.

Если бы мы решили создать версию Password Verifier с состоянием и захотели внедрить функцию даты через конструктор, это могло бы выглядеть так, как показано в листинге 3.10.

Листинг 3.10. Решение с внедрением через конструктор

```
class PasswordVerifier {
    constructor(rules, dayOfWeekFn) {
        this.rules = rules;
        this.dayOfWeek = dayOfWeekFn;
    }

    verify(input) {
        if ([SATURDAY, SUNDAY].includes(this.dayOfWeek())) {
            throw new Error("It's the weekend!");
        }
        const errors = [];
        // ...
        return errors;
    };
}

test('class constructor: on weekends, throws exception', () => {
    const alwaysSunday = () => SUNDAY;
    const verifier = new PasswordVerifier([], alwaysSunday);

    expect(() => verifier.verify('anything'))
        .toThrow("It's the weekend!");
});
```

Решение очень похоже на схему с функцией-конструктором из раздела 3.6. Это более классово-ориентированная структура, с которой многие разработчики с опытом объектно-ориентированной разработки чувствуют себя увереннее. Но решение стало длиннее. Как видите, чем более подход объектно-ориентированный, тем больше кода требуется. Это часть объектно-ориентированной игры, и одна из причин, по которой все чаще выбираются функциональные стили, — это их лаконичность.

Поговорим о простоте в сопровождении тестов. Если бы я написал второй тест с этим классом, я бы выделил создание класса через конструктор в удобную маленькую фабричную функцию, которая возвращает экземпляр тестируемого класса. Если (а вернее, когда) сигнатура конструктора изменится и нарушит сразу много тестов, мне придется внести исправления только в одном месте, и все тесты снова заработают (см. листинг 3.11).

Листинг 3.11. Добавление вспомогательной фабричной функции в тесты

```
describe('refactored with constructor', () => {
  const makeVerifier = (rules, dayFn) => {
    return new PasswordVerifier(rules, dayFn);
  };

  test('class constructor: on weekends, throws exceptions', () => {
    const alwaysSunday = () => SUNDAY;
    const verifier = makeVerifier([],alwaysSunday);

    expect(() => verifier.verify('anything'))
      .toThrow("It's the weekend!");
  });

  test('class constructor: on weekdays, with no rules, passes', () => {
    const alwaysMonday = () => MONDAY;
    const verifier = makeVerifier([],alwaysMonday);

    const result = verifier.verify('anything');
    expect(result.length).toBe(0);
  });
});
```

Обратите внимание: это решение отличается от решения с фабричной функцией из раздела 3.4.2. Сейчас фабричная функция размещается в наших *тестах*; в том случае она размещалась в рабочем коде. Данное решение упрощает сопровождение тестов, и оно может работать как с объектно-ориентированным, так и с функциональным рабочим кодом, потому что оно скрывает, как создается или настраивается функция или объект. Оно создает прослойку абстракции в наших тестах, чтобы зависимость от создания или настройки функции или объекта можно было разместить в одном месте в тестах.

3.7.2. Внедрение объекта вместо функции

А здесь конструктор класса получает функцию во втором параметре.

```
constructor(rules, dayOfWeekFn) {
  this.rules = rules;
  this.dayOfWeek = dayOfWeekFn;
}
```

Сделаем следующий шаг в объектно-ориентированном дизайне и используем объект вместо функции в качестве параметра. Для этого придется немного по-трудиться: провести рефакторинг кода.

Начнем с создания нового файла с именем `time-provider.js`, который будет содержать реальный объект с зависимостью от `moment.js`. Объект содержит всего одну функцию с именем `getDay()`.

```
import moment from "moment";

const RealTimeProvider = () => {
    this.getDay = () => moment().day()
};


```

Затем внесем изменения в параметр, чтобы в нем передавался объект с функцией.

```
const SUNDAY = 0, MONDAY = 1, SATURDAY = 6;
class PasswordVerifier {
    constructor(rules, timeProvider) {
        this.rules = rules;
        this.timeProvider = timeProvider;
    }

    verify(input) {
        if ([SATURDAY, SUNDAY].includes(this.timeProvider.getDay())) {
            throw new Error("It's the weekend!");
        }
        ...
    }
}


```

Наконец, предоставим стороне, которой понадобился экземпляр `PasswordVerifier`, возможность предварительно настроить ее провайдером реального времени по умолчанию. Для этого прибегнем к помощи новой функции `passwordVerifierFactory`; она должна будет использоваться всем рабочим кодом, которому понадобится экземпляр верификатора.

```
const passwordVerifierFactory = (rules) => {
    return new PasswordVerifier(new RealTimeProvider())
};


```

В листинге 3.12 приведен весь блок нового кода.

Листинг 3.12. Внедрение объекта

```
import moment from "moment";

const RealTimeProvider = () => {
    this.getDay = () => moment().day()
};

const SUNDAY = 0, MONDAY=1, SATURDAY = 6;
```

```

class PasswordVerifier {
  constructor(rules, timeProvider) {
    this.rules = rules;
    this.timeProvider = timeProvider;
  }

  verify(input) {
    if ([SATURDAY, SUNDAY].includes(this.timeProvider.getDay())) {
      throw new Error("It's the weekend!");
    }
    const errors = [];
    // ...
    return errors;
  };
}

const passwordVerifierFactory = (rules) => {
  return new PasswordVerifier(new RealTimeProvider())
};

```

Контейнеры IoC и внедрение зависимостей

Существует много других способов соединения `PasswordVerifier` и `TimeProvider`. Я выбрал ручное внедрение, чтобы не усложнять пример. Многие современные фреймворки позволяют настроить внедрение зависимостей в тестируемые объекты, что дает возможность определить, как должен конструироваться объект. Angular — один из таких фреймворков.

Если вы используете такие библиотеки, как Spring в Java, Autofac или StructureMap в C#, конструирование объектов легко настраивается внедрением через конструктор, и это избавляет вас от необходимости создавать специализированные функции. Эта функциональность обычно объединяется под термином «контейнеры IoC» (Inversion of Control, «инверсия контроля») или «контейнеры DI» (Dependency Injection, «внедрение зависимостей»). Я не буду использовать их в книге, чтобы избежать ненужных подробностей. Они не нужны для создания хороших тестов. Собственно, обычно я и не использую контейнеры IoC в тестах. Я почти всегда применяю специальные фабричные функции для внедрения зависимостей. На мой взгляд, с ними тесты проще читать и анализировать.

Даже для тестов, покрывающих код Angular, не обязательно использовать DI-фреймворк Angular для внедрения зависимости в объект в памяти; можно вызвать конструктор этого объекта напрямую и передать фейковые данные. Пока это делается в фабричной функции, мы не жертвуем простотой в сопровождении, а также не добавляем в тесты лишний код, если он не является абсолютно необходимым для тестов.

Как использовать такую структуру в тестах, где требуется внедрить фейковый объект вместо фейковой функции? Сначала мы сделаем это вручную, чтобы вы увидели, что ничего страшного в этом нет. Потом мы воспользуемся помощью фреймворка, но вы увидите, что иногда ручное программирование фейковых объектов может сделать ваш тест более читабельным, чем применение фреймворка — такого, как Jasmine, Jest или Sinon (они будут рассмотрены в главе 5).

Сначала в файле теста создается новый фейковый объект с такой же сигнатурой функции, как у провайдера реального времени, но который может контролироваться нашими тестами. В таком случае можно воспользоваться паттерном с конструктором.

```
function FakeTimeProvider(fakeDay) {
    this.getDay = function () {
        return fakeDay;
    }
}
```

ПРИМЕЧАНИЕ Если вы работаете в более объектно-ориентированном стиле, можно выбрать решение с созданием простого класса, наследующего от общего интерфейса. Оно будет рассмотрено далее в этой главе.

Затем в тестах создается объект `FakeTimeProvider`, который внедряется в тестируемый экземпляр `verifier`.

```
describe('verifier', () => {
    test('on weekends, throws exception', () => {
        const verifier =
            new PasswordVerifier([], new FakeTimeProvider(SUNDAY));

        expect(()=> verifier.verify('anything'))
            .toThrow("It's the weekend!");
    });
});
```

Весь код файла теста приведен в листинге 3.13.

Листинг 3.13. Создание стаба вручную

```
function FakeTimeProvider(fakeDay) {
    this.getDay = function () {
        return fakeDay;
    }
}

describe('verifier', () => {
    test('class constructor: on weekends, throws exception', () => {
        const verifier =
            new PasswordVerifier([], new FakeTimeProvider(SUNDAY));
```

```

expect(() => verifier.verify('anything'))
    .toThrow("It's the weekend!");
});
});

```

Этот код работает, потому что JavaScript по умолчанию предоставляет вам значительную свободу действий. Как и в Ruby или Python, часто удается обойтись «утиной типизацией». Концепция «утиной типизации» формулируется следующим образом: если что-то ходит как утка и крякает как утка, то мы будем работать с ним как с уткой. В данном случае реальный и фейковый объекты реализуют одну функцию, хотя и являются совершенно разными объектами. Мы просто передаем один объект вместо другого, и рабочий код это должно устроить.

Конечно, мы будем знать лишь то, что рабочий код это устраивает и что мы не допустили никаких ошибок и не забыли ничего, что относится к сигнатурам функций во время выполнения. А если вам нужно чуть больше уверенности, можно попробовать сделать то же самое способом, более безопасным по отношению к типам.

3.7.3. Выделение общего интерфейса

Можно пойти еще дальше и, если вы используете TypeScript или другой язык с сильной типизацией (например, Java или C#), начать использовать интерфейсы для обозначения ролей наших зависимостей. При этом создается своего рода контракт, который должен соблюдаться как реальными, так и фейковыми объектами на уровне компилятора.

Сначала мы определяем новый интерфейс (обратите внимание: приводится код TypeScript).

```

export interface TimeProviderInterface {
    getDay(): number;
}

```

Затем определяется провайдер реального времени, который реализует наш интерфейс в рабочем коде.

```

import * as moment from "moment";
import {TimeProviderInterface} from "./time-provider-interface";

export class RealTimeProvider implements TimeProviderInterface {
    getDay(): number {
        return moment().day();
    }
}

```

Наконец, мы обновляем конструктор `PasswordVerifier` для получения зависимости нового типа `TimeProviderInterface` вместо параметра типа `RealTimeProvider`. Мы абстрагируем роль провайдера времени и объявляем, что нас не интересует, какой объект передается, при условии, что он соответствует интерфейсу этой роли.

```
export class PasswordVerifier {
    private _timeProvider: TimeProviderInterface;

    constructor(rules: any[], timeProvider: TimeProviderInterface) {
        this._timeProvider = timeProvider;
    }
    verify(input: string[]): string[] {
        const isWeekened = [SUNDAY, SATURDAY]
            .filter(x => x === this._timeProvider.getDay())
            .length > 0;
        if (isWeekened) {
            throw new Error("It's the weekend!")
        }
        // ...
        return [];
    }
}
```

Теперь у нас имеется интерфейс, который определяет, как выглядит «утка», и мы можем реализовать собственную «утку» в своих тестах. Решение очень похоже на код предыдущего теста, но с одним серьезным различием: правильность сигнатур методов будет обеспечиваться компилятором.

Так выглядит провайдер фейкового времени в нашем тестовом файле:

```
class FakeTimeProvider implements TimeProviderInterface {
    fakeDay: number;
    getDay(): number {
        return this.fakeDay;
    }
}
```

А вот как выглядит наш тест:

```
describe('password verifier with interfaces', () => {
    test('on weekends, throws exceptions', () => {
        const stubTimeProvider = new FakeTimeProvider();
        stubTimeProvider.fakeDay = SUNDAY;
        const verifier = new PasswordVerifier([], stubTimeProvider);

        expect(() => verifier.verify('anything'))
            .toThrow("It's the weekend!");
    });
});
```

В листинге 3.14 приведена полная версия кода.

Листинг 3.14. Выделение общего интерфейса в рабочем коде

```
export interface TimeProviderInterface { getDay(): number; }

export class RealTimeProvider implements TimeProviderInterface {
    getDay(): number {
        return moment().day();
    }
}

export class PasswordVerifier {
    private _timeProvider: TimeProviderInterface;

    constructor(rules: any[], timeProvider: TimeProviderInterface) {
        this._timeProvider = timeProvider;
    }

    verify(input: string[]): string[] {
        const isWeekend = [SUNDAY, SATURDAY]
            .filter(x => x === this._timeProvider.getDay())
            .length > 0;
        if (isWeekend) {
            throw new Error("It's the weekend!");
        }
        return [];
    }
}

class FakeTimeProvider implements TimeProviderInterface{
    fakeDay: number;
    getDay(): number {
        return this.fakeDay;
    }
}

describe('password verifier with interfaces', () => {
    test('on weekends, throws exceptions', () => {
        const stubTimeProvider = new FakeTimeProvider();
        stubTimeProvider.fakeDay = SUNDAY;
        const verifier = new PasswordVerifier([], stubTimeProvider);

        expect(() => verifier.verify('anything'))
            .toThrow("It's the weekend!");
    });
});
```

Мы перешли от чисто функционального решения к объектно-ориентированному с сильной типизацией. Какой из вариантов лучше подходит для вашей команды

и вашего проекта? Единственно правильного ответа на существует. Мы еще вернемся к обсуждению проектировочных решений в главе 8. А здесь я в основном хотел показать, что, какой бы вариант вы ни выбрали, паттерн внедрения остается в целом неизменным. Просто для его реализации используются разные понятия или языковые средства.

Именно возможность внедрения позволяет нам моделировать то, что было бы практически невозможно протестировать в реальной жизни. В таких случаях концепция стабов проявляет себя в полной мере. Мы можем приказать своим стабам возвращать фейковые значения и даже моделировать исключения в нашем коде, чтобы увидеть, как он обрабатывает ошибки, возникающие из-за зависимостей. Внедрение предоставляет такую возможность. Внедрение также делает тесты более повторяемыми, стабильными и достоверными. Тема достоверности рассматривается в третьей части книги. В следующей главе мы поговорим о моках и выясним, чем они отличаются от стабов.

ИТОГИ

- *Тестовый дублер* — обобщающий термин, описывающий любые виды нерабочих, фейковых зависимостей в тестах. Существуют пять разновидностей тестовых дублеров, которые можно сгруппировать в две категории: *моки* и *стабы*.
- *Моки* помогают эмулировать и анализировать *исходящие зависимости*, то есть зависимости, представляющие точку выхода нашей единицы работы. Тестируемая система (SUT) вызывает исходящие зависимости для изменения их состояния. *Стабы* помогают эмулировать *входящие зависимости*: тестируемая система вызывает их для получения входных данных.
- Стабы помогают заменить ненадежную зависимость фейковой, надежной зависимостью и, как следствие, избежать *неустойчивости тестов*.
- Существует несколько способов внедрения стабов в единицу работы.
 - *Функция как параметр* — внедрение функции вместо простого значения.
 - *Частичное применение (каррирование)* и *фабричные функции* — создание функции, которая возвращает другую функцию, включающую часть контекста. Этот контекст может включать зависимость, которую вы заменили стабом.
 - *Модульное внедрение* — замена модуля фейковым с тем же API. Такие решения получаются слишком хрупкими. Если API модуля, для которого вы создаете фейк, изменится в будущем, это может потребовать значительного рефакторинга.
 - *Функция-конструктор* — в основном то же, что частичное применение.

- *Внедрение через конструктор* — стандартный объектно-ориентированный прием с внедрением зависимости через конструктор.
- *Объект как параметр («утиная типизация»)* — в JavaScript можно внедрить любую зависимость вместо требуемой — при условии, что она реализует те же функции.
- *Общий интерфейс как параметр* — то же, что объект как параметр, но с проверкой на стадии компиляции. Для этого способа необходим язык с сильной типизацией (такой, как TypeScript).

Тестирование взаимодействий с использованием моков

В ЭТОЙ ГЛАВЕ

- ✓ Тестирование взаимодействий: определение
- ✓ Причины для использования моков
- ✓ Внедрение и использование моков
- ✓ Работа со сложными интерфейсами
- ✓ Частичные моки

В предыдущей главе мы решили проблему тестирования кода, правильное выполнение которого зависит от других объектов. Мы применяли стабы, чтобы тестируемый код получал весь необходимый ввод и единицу работы можно было протестировать в изоляции.

До настоящего момента мы писали только тесты, работавшие с первыми двумя из трех типов точек выхода, которые может иметь единица работы: *возвращаемое значение* и *изменение состояния системы* (об этих типах можно подробнее прочитать в главе 1). В этой главе рассматриваются возможности тестирования третьей разновидности точек выхода — вызовов сторонних функций, модулей или объектов. Это важно, потому что нам часто приходится работать с кодом, зависящим от чего-то, что мы не можем контролировать. Умение тестировать

код такого рода — важный навык для юнит-тестирования. Фактически мы будем искать способы доказать, что единица работы в конечном итоге вызывает функцию, которую мы не контролируем, и будем стараться определить, какие значения были переданы в аргументах.

Методы, рассмотренные ранее, в данном случае не подойдут, потому что сторонние функции обычно не имеют специализированных API, при помощи которых можно было бы проверить правильность вызова. Вместо этого они выполняют свои операции на внутреннем уровне, что упрощает их сопровождение. Как же проверить, что ваша единица работы правильно взаимодействует со сторонними функциями? Воспользуйтесь моками.

4.1. ТЕСТИРОВАНИЕ ВЗАИМОДЕЙСТВИЙ, МОКИ И СТАБЫ

Тестированием взаимодействий называется проверка того, как единица работы взаимодействует и отправляет сообщения (то есть вызывает функции) зависимости, не находящейся под ее контролем. Моки-функции (или объекты) используются для проверки того, что вызов внешней зависимости был сделан правильно.

Вспомним различия между моками и стабами, рассмотренные в главе 3. Главное различие связано с направлением потока информации.

- *Мок* используется для устранения исходящих зависимостей. Моки являются фейковыми модулями, объектами или функциями, вызовы которых мы проверяем в своих тестах. Мок представляет *точку выхода* в юнит-тесте. Если к ней не применяется проверка (`assert`), то она не используется как мок.

Тест должен содержать не более одного мока, что объясняется соображениями сопровождаемости и читабельности. (Эта тема более подробно рассматривается в части 3 книги, где речь идет о написании простых в сопровождении тестов.)

- *Стаб* используется для устранения входящих зависимостей. Стабы представляют собой фейковые модули, объекты или функции, предоставляющие фиктивное поведение или данные тестируемому коду. Проверки к ним не применяются, и один тест может содержать много стабов.

Стабы являются промежуточными пунктами, а не точками выхода, потому что данные или поведение входят в единицу работы. Они являются точками взаимодействий, но не представляют конечный результат единицы работы. Скорее, это взаимодействия *на пути* к достижению интересующего нас конечного результата, так что мы не рассматриваем их как точки выхода.

На рис. 4.1 сравниваются эти две категории.

Рассмотрим простой пример точки выхода к зависимости, которую мы не контролируем: вызов логгера.

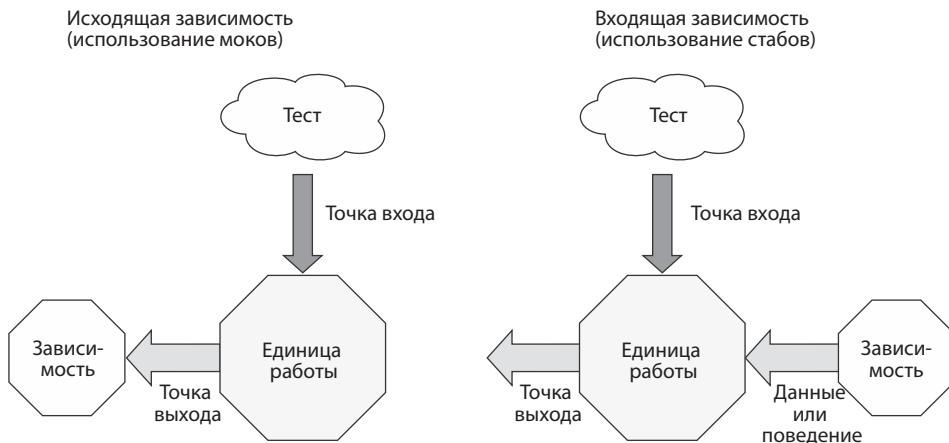


Рис. 4.1. Слева: точка выхода, реализованная как вызов зависимости.

Справа: зависимость предоставляет непрямой ввод или поведение и не является точкой выхода

4.2. ЗАВИСИМОСТЬ ОТ ЛОГГЕРА

Для примера возьмем функцию из Password Verifier. Допустим, у вас имеется сложный логгер (то есть логгер с дополнительными функциями и параметрами, так что его интерфейс нетривиален). Одно из требований нашей функции — вызов логгера, когда проверка пароля прошла или не прошла. См листинг 4.1.

Листинг 4.1. Прямая зависимость от сложного логгера

```
// для создания фейка не могут использоваться традиционные способы внедрения
const log = require('./complicated-logger');

const verifyPassword = (input, rules) => {
  const failed = rules
    .map(rule => rule(input))
    .filter(result => result === false);
  if (failed.count === 0) {
    // тестирование при помощи традиционных способов внедрения
```

```

log.info('PASSED');
return true; // Точка выхода
}
//тестирование при помощи традиционных способов внедрения невозможно
log.info('FAIL'); // Точка выхода
return false; //
};

const info = (text) => {
  console.log(`INFO: ${text}`);
};

const debug = (text) => {
  console.log(`DEBUG: ${text}`);
};

```

Происходящее изображено на рис. 4.2. Наша функция `verifyPassword` является точкой входа в единицу работы, и еще существуют две точки выхода: одна возвращает значение, а другая вызывает `log.info()`.

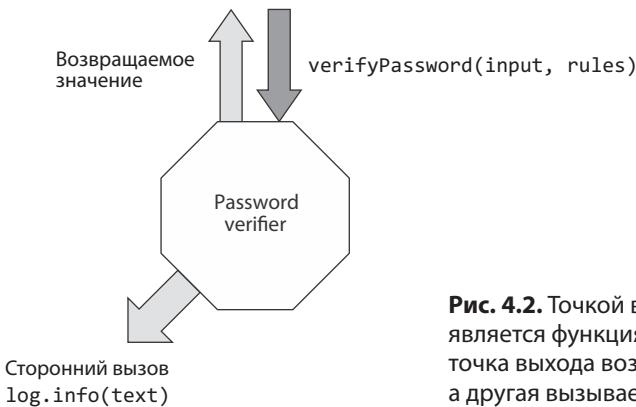


Рис. 4.2. Точкой входа в Password Verifier является функция `verifyPassword`. Одна точка выхода возвращает значение, а другая вызывает `log.info()`

К сожалению, мы не можем убедиться в том, что логгер был действительно вызван, традиционными средствами или без использования специфических приемов Jest, которые я обычно применяю только при отсутствии других вариантов, так как эти приемы усложняют чтение и сопровождение тестов (подробнее об этом далее в этой главе).

Давайте сделаем то, что мы обычно делаем с зависимостями: *абстрагируем их*. Есть много способов создания швов в коде. Напомню, что *швами* называются места, в которых соединяются две части кода, и швы могут использоваться для внедрения фейковых сущностей. В таблице 4.1 перечислены самые распространенные способы абстрагирования зависимостей.

Таблица 4.1. Методы внедрения фейков

| Стиль | Метод |
|--------------------------|---|
| Стандартный | Введение параметра |
| Функциональный | Применение каррирования Переход к функциям высшего порядка |
| Модульный | Абстрагирование модульной зависимости |
| Объектно-ориентированный | Внедрение нетипизированного объекта Внедрение интерфейса |

4.3. СТАНДАРТНЫЙ СТИЛЬ: РЕФАКТОРИНГ С ВВЕДЕНИЕМ ПАРАМЕТРА

Самым очевидным первым шагом будет введение нового параметра в тестируемый код.

Листинг 4.2. Внедрение параметра для логгера-мока

```
const verifyPassword2 = (input, rules, logger) => {
    const failed = rules
        .map(rule => rule(input))
        .filter(result => result === false);

    if (failed.length === 0) {
        logger.info('PASSED');
        return true;
    }
    logger.info('FAIL');
    return false;
};
```

В листинге 4.3 показано, как написать простейший тест для этого кода с использованием простого механизма замыкания.

Листинг 4.3. Ручная реализация мока

```
describe('password verifier with logger', () => {
    describe('when all rules pass', () => {
        it('calls the logger with PASSED', () => {
            let written = '';
            const mockLog = {
                info: (text) => {
                    written = text;
                }
            };
        });
    });
});
```

```

    verifyPassword2('anything', [], mockLog);

    expect(written).toMatch(/PASSED/);
});
});
});

```

Сначала обратите внимание на то, что переменной присваивается имя `mockXXX` (`mockLog` в этом примере) для отражения того факта, что в тесте присутствует мок-функция или объект. Я использую эту схему назначения имен, потому что хочу, чтобы вы, как читатель теста, знали, что в конце теста следует ожидать проверки (или *верификации*) мока. Такой подход к выбору имен убережет читателя кода от неожиданностей и сделает тест намного более предсказуемым. Эта схема именования используется только для сущностей, которые действительно являются моками.

Наш первый мок:

```

let written = '';
const mockLog = {
  info: (text) => {
    written = text;
  }
};

```

Он содержит всего одну функцию, которая имитирует сигнатуру функции `info` логгера. Кроме того, он сохраняет переданный параметр (`text`), чтобы мы могли проверить, что он используется для вызова далее в тесте. Если переменная `written` содержит правильный текст, это доказывает, что наша функция была вызвана; следовательно, мы доказали, что точка выхода правильно вызывается из нашей единицы работы.

На стороне `verifyPassword2` выполненный нами рефакторинг весьма типичен. Он мало чем отличается от того, что мы делали в предыдущей главе, где стаб выделялся как зависимость. Стабы и моки часто обрабатываются одинаково в отношении рефакторинга и создания швов в коде приложения.

Что дает этот простой рефакторинг в параметр?

- Нам не приходится явно импортировать (через `require`) логгер в тестируемом коде. Это означает, что, если вы когда-либо измените реальную зависимость логгера, у тестируемого кода будет на одну причину для изменений меньше.
- Теперь у нас появляется возможность внедрения *любого логгера* на свой выбор в тестируемый код при условии, что он предоставляет тот же интерфейс (или по крайней мере содержит метод `info`). Это означает, что мы можем предоставить мок, который выполняет нашу работу: мок-логгер помогает убедиться в том, что он был вызван правильно.

ПРИМЕЧАНИЕ Наш мок моделирует только часть интерфейса логгера (в нем отсутствует функция `debug`), что делает этот пример разновидностью «утиной типизации». Эта концепция обсуждалась в главе 3: если что-то ходит как утка и крякает как утка, то мы можем использовать его как фейковый объект.

4.4. ПОЧЕМУ ВАЖНО ОТЛИЧАТЬ МОКИ ОТ СТАБОВ

Почему я уделяю столько внимания терминологии? Если вы не понимаете, чем моки отличаются от стабов, или неправильно используете термины, у вас получатся тесты, которые проверяют сразу несколько условий и которые трудно читать и сопровождать. Правильное обозначение понятий помогает избежать этих проблем.

С учетом того, что мок представляет требование нашей единицы работы («вызывает логгер», «отправляет электронную почту» и т. д.), а стаб представляет входную информацию или поведение («запрос к базе данных возвращает `false`», «эта конкретная конфигурация выдает ошибку»), можно сформулировать простое правило: тест может включать несколько стабов, но обычно в нем не должно быть более *одного мока*, потому что это будет означать, что в одном teste вы проверяете сразу несколько требований.

Если вы не можете (или не хотите) различать эти понятия (то есть правильно выбирать термины), у вас могут появиться тесты с несколькими моками или с проверкой стабов; и то и другое имеет отрицательные последствия для тестов. Последовательность при выборе имен предоставляет ряд преимуществ.

- *Читабельность* — имена тестов могут стать намного более обобщенными и трудными для понимания. Вы захотите, чтобы люди могли прочитать имя теста и узнать все, что происходит или тестируется в нем, без необходимости читать код теста.
- *Простота в сопровождении* — если вы не различаете моки и стабы, это может привести к тому, что вы будете применять проверки к стабам (по неосторожности или даже потому, что не считаете это важным). Такая информация не приносит особой пользы и усиливает связи между тестами и внутренним рабочим кодом. Хорошим примером такого рода служит проверка того, что к базе данных был выдан запрос. Вместо того чтобы проверять, что запрос к базе данных возвращает некоторое значение, будет намного лучше проверить, что изменение ввода из базы данных приводит к изменению поведения приложения.
- *Достоверность* — если один тест содержит несколько моков (требований) и проверка первого мока не проходит, многие тестовые фреймворки не выполняют остаток теста (под строкой неудачной проверки) из-за выданного исключения. Это означает, что другие моки не проверяются и вы не получите от них результаты.

Чтобы лучше понять последний пункт, представьте доктора, который должен принять решение, видя только 30 % симптомов своего пациента, — лечение может быть выбрано неправильно. Если вы не видите, где скрываются ошибки, или вместо одного предположения нарушаются сразу два (потому что одно из них было скрыто после первого сбоя), вы с большей вероятностью исправите то, что не нужно, или внесете исправление не в том месте.

В книге «XUnit Test Patterns» Джерарда Месароша такая ситуация называется *рулеткой утверждений* (<http://xunitpatterns.com/Assertion%20Roulette.html>). Мне нравится это сравнение. Происходящее напоминает азартную игру: вы принимаетесь закрывать комментариями строки кода в своем тесте, и начинается веселье (возможно, с алкоголем).

Не все является моком

К сожалению, многие люди склонны обозначать словом «мок» все сущности, которые не являются реальными, например «мок-база данных» или «мок-сервис». В большинстве случаев обычно подразумевается использование стаба.

Впрочем, это простительно. Такие фреймворки, как Mockito, jMock и большинство изолирующих фреймворков (я не называю их мок-фреймворками по причинам, которые уже упоминались выше), используют термин «мок» для обозначения как моков, так и стабов.

Существуют более новые фреймворки (такие, как Sinon и testdouble в JavaScript, NSubstitute и FakeItEasy в .NET и т. д.), которые способствовали изменению терминологии. Надеюсь, эта тенденция продолжится.

4.5. МОКИ В МОДУЛЬНОМ СТИЛЕ

Модульное внедрение зависимостей рассматривалось в предыдущей главе, но теперь мы посмотрим, как использовать его для внедрения моков и моделировать ответы от них.

4.5.1. Пример рабочего кода

Рассмотрим чуть более сложный пример. В этом сценарии функция `verifyPassword` зависит от двух внешних зависимостей:

- логгер;
- сервис конфигурации.

Сервис конфигурации предоставляет требуемый уровень логирования. Обычно этот тип кода перемещается в специальный модуль, но в примерах этой книги я размещаю логику, которая вызывает `logger.info` и `logger.debug` прямо в тестируемом коде.

Листинг 4.4. Жесткая модульная зависимость

```
const { info, debug } = require("./complicated-logger");
const { getLogLevel } = require("./configuration-service");

const log = (text) => {
  if (getLogLevel() === "info") {
    info(text);
  }
  if (getLogLevel() === "debug") {
    debug(text);
  }
};

const verifyPassword = (input, rules) => {
  const failed = rules
    .map((rule) => rule(input))
    .filter((result) => result === false);

  if (failed.length === 0) {
    log("PASSED");           ←
    return true;
  }
  log("FAIL");             ←
  return false;
};

module.exports = {
  verifyPassword,
};
```



Допустим, мы поняли, что ошибка возникает при вызове логгера. Мы изменили способ проверки сбоев и теперь вызываем логгер с результатом `PASSED`, когда количество сбоев положительно (вместо нуля). Как при помощи юнит-теста доказать, что эта ошибка существует (или мы исправили ее)?

Проблема в том, что мы импортируем (или подключаем с `require`) модули непосредственно в наш код. Если вы хотите заменить модуль-логгер, придется либо заменить файл, либо выполнить какой-нибудь магический ритуал через Jest API. Я не рекомендую так поступать, потому что использование этих методов причиняет больше боли и страданий, чем обычно при работе с кодом.

4.5.2. Рефакторинг рабочего кода в стиле модульного внедрения

Модульные зависимости можно абстрагировать в объект и разрешить пользователю нашего модуля заменить этот объект так, как показано в листинге 4.5.

Листинг 4.5. Рефакторинг с паттерном модульного внедрения

```
const originalDependencies = {
  log: require('./complicated-logger'), | Для хранения исходных
}; | зависимостей

let dependencies = { ...originalDependencies }; ← Промежуточный уровень

const resetDependencies = () => {
  dependencies = { ...originalDependencies }; ← Функция сброса зависимостей
};

const injectDependencies = (fakes) => {
  Object.assign(dependencies, fakes); ← | Функция
}; | переопределения
      | зависимостей

const verifyPassword = (input, rules) => {
  const failed = rules
    .map(rule => rule(input))
    .filter(result => result === false);

  if (failed.length === 0) {
    dependencies.log.info('PASSED');
    return true;
  }
  dependencies.log.info('FAIL');
  return false;
};

module.exports = {
  verifyPassword,
  injectDependencies, | API предоставляется
  resetDependencies | пользователям модуля
};
```

Здесь больше рабочего кода, и он кажется более сложным, но это позволяет нам относительно легко заменять зависимости в наших тестах, если нам приходится работать в таком модульном стиле.

Переменная `originalDependencies` всегда содержит исходные зависимости, чтобы они никогда не терялись между тестами. `dependencies` — наш промежуточный

уровень абстракции. По умолчанию в ней хранятся исходные зависимости, но наши тесты могут приказать тестируемому коду заменить эту переменную нестандартными зависимостями (ничего не зная о внутреннем строении модуля). `injectDependencies` и `resetDependencies` — открытый API, который предоставляет модулем для переопределения и сброса зависимостей.

4.5.3. Пример теста с внедрением в модульном стиле

В листинге 4.6 показано, как может выглядеть тест для модульного внедрения.

Листинг 4.6. Тестирование с модульным внедрением

```
const {
  verifyPassword,
  injectDependencies,
  resetDependencies,
} = require("./password-verifier-injectable");

describe("password verifier", () => {
  afterEach(resetDependencies);

  describe("given logger and passing scenario", () => {
    it("calls the logger with PASS", () => {
      let logged = "";
      const mockLog = { info: (text) => (logged = text) };
      injectDependencies({ log: mockLog });

      verifyPassword("anything", []);
      expect(logged).toMatch(/PASSED/);
    });
  });
});
```

Теперь вы можете достаточно легко внедрять модули для целей тестирования, если не будете забывать использовать функцию `resetDependencies` после каждого теста. Очевидная главная проблема состоит в том, что этот подход требует, чтобы каждый модуль предоставлял функции `inject` и `reset`, которые могут использоваться извне. Это требование может соответствовать (или нет) текущим ограничениям проектирования, но, если соответствует, вы сможете абстрагировать их в функции, пригодные для повторного использования, и избавиться от большого объема шаблонного кода.

4.6. МОКИ В ФУНКЦИОНАЛЬНОМ СТИЛЕ

Рассмотрим несколько функциональных стилей, которые могут использоваться для внедрения моков в тестируемый код.

4.6.1. Использование стиля каррирования

Давайте реализуем прием каррирования, представленный в главе 3, для внедрения логгера в более функциональном стиле. В листинге 4.7 используется `lodash` (библиотека, упрощающая функциональное программирование на JavaScript), чтобы каррирование можно было реализовать без большого объема шаблонного кода.

Листинг 4.7. Применение каррирования к нашей функции

```
const verifyPassword3 = _.curry((rules, logger, input) => {
  const failed = rules
    .map(rule => rule(input))
    .filter(result => result === false);
  if (failed.length === 0) {
    logger.info('PASSED');
    return true;
  }
  logger.info('FAIL');
  return false;
});
```

Изменений совсем немного: вызов `_.curry` в первой строке и его закрытие в конце блока.

Листинг 4.8 показывает, как может выглядеть тест для такого кода.

Листинг 4.8. Тестирование каррированной функции с внедрением зависимости

```
describe("password verifier", () => {
  describe("given logger and passing scenario", () => {
    it("calls the logger with PASS", () => {
      let logged = "";
      const mockLog = { info: (text) => (logged = text) };
      const injectedVerify = verifyPassword3([], mockLog);

      // частично примененная функция может передаваться
      // в другие места кода
      // без необходимости внедрения логгера
      injectedVerify("anything");

      expect(logged).toMatch(/PASSED/);
    });
  });
});
```

Наш тест активирует функцию с первыми двумя аргументами (внедряя зависимости `rules` и `logger`, фактически возвращая частично примененную функцию), после чего вызывает возвращенную функцию `injectedVerify` с итоговым вводом, тем самым показывая читателю кода:

- как эта функция должна применяться в реальной жизни;
- какие используются зависимости.

В остальном она почти не отличается от предыдущего теста.

4.6.2. Работа с функциями высшего порядка без каррирования

В листинге 4.9 приведена другая вариация на тему функционального программирования: там используется функция высшего порядка, но без каррирования. Можно сразу определить, что в коде не применяется каррирование, потому что все параметры должны передаваться в аргументах функции, чтобы она правильно работала.

Листинг 4.9. Внедрение мока в функцию высшего порядка

```
const makeVerifier = (rules, logger) => {
  return (input) => {
    const failed = rules
      .map(rule => rule(input))
      .filter(result => result === false);

    if (failed.length === 0) {
      logger.info('PASSED');
      return true;
    }
    logger.info('FAIL');
    return false;
  };
};
```



На этот раз я явно создаю фабричную функцию, которая возвращает *предварительно настроенную функцию-верификатор*, уже содержащую `rules` и `logger` в зависимостях своего замыкания.

А теперь рассмотрим тест для этого кода. Тест должен сначала вызвать фабричную функцию `makeVerifier`, а затем вызвать функцию, возвращенную этой функцией (`passVerify`).

Листинг 4.10. Тестирование с использованием фабричной функции

```

describe("higher order factory functions", () => {
  describe("password verifier", () => {
    test("given logger and passing scenario", () => {
      let logged = "";
      const mockLog = { info: (text) => (logged = text) };
      const passVerify = makeVerifier([], mockLog); ← Вызывает фабричную
                                                 функцию
      passVerify("any input"); ← Вызывает полученную
                                 функцию
      expect(logged).toMatch(/PASSED/);
    });
  });
});

```

4.7. МОКИ В ОБЪЕКТНО-ОРИЕНТИРОВАННОМ СТИЛЕ

Итак, мы рассмотрели некоторые функциональные и модульные стили; обратимся теперь к объектно-ориентированным. Люди с опытом объектно-ориентированной разработки чувствуют себя с этим подходом намного увереннее, а людям с опытом функционального программирования он не понравится. Но в жизни нам часто приходится считаться с мнением других.

4.7.1. Рефакторинг рабочего кода для внедрения

В листинге 4.11 показано, как может выглядеть внедрение в решении на базе классов в JavaScript. У классов есть конструкторы, и мы используем конструктор для того, чтобы заставить вызывающую сторону класса передать параметры. Это не единственное возможное решение, но оно очень часто встречается и приносит пользу в объектно-ориентированных архитектурах, потому что с ним обязательность этих параметров выражается явно и практически неоспоримо в языках с сильной типизацией (таких, как Java или C) и при использовании TypeScript. Мы хотим быть уверены: тот, кто использует наш код, знает, что необходимо сделать для его правильной настройки.

Листинг 4.11. Внедрение через конструктор на базе класса

```

class PasswordVerifier {
  _rules;
  _logger;

  constructor(rules, logger) {
    this._rules = rules;
    this._logger = logger;
  }
}

```

```

verify(input) {
  const failed = this._rules
    .map(rule => rule(input))
    .filter(result => result === false);

  if (failed.length === 0) {
    this._logger.info('PASSED');
    return true;
  }
  this._logger.info('FAIL');
  return false;
}
}

```

Перед вами стандартный класс, который получает пару параметров конструктора, а затем использует их внутри функции `verify`. Листинг 4.12 показывает, как может выглядеть тест.

Листинг 4.12. Внедрение мока-логгера в параметре конструктора

```

describe("duck typing with function constructor injection", () => {
  describe("password verifier", () => {
    test("logger&passing scenario,calls logger with PASSED", () => {
      let logged = "";
      const mockLog = { info: (text) => (logged = text) };
      const verifier = new PasswordVerifier([], mockLog);
      verifier.verify("any input");

      expect(logged).toMatch(/PASSED/);
    });
  });
});

```

Внедрение моков выполняется достаточно прямолинейно, как и при внедрении стабов, продемонстрированном в предыдущей главе. Если бы мы использовали свойства вместо конструктора, это означало бы, что зависимости *необязательны*. С конструктором мы явно указываем на их обязательность.

В языках с сильной типизацией — таких, как Java или C#, — фейковые логгеры часто выделяются в отдельный класс.

```

class FakeLogger {
  logged = "";

  info(text) {
    this.logged = text;
  }
}

```

Мы просто реализуем функцию `info` в классе, но вместо того, чтобы что-то регистрировать, она просто сохраняет значение, переданное в параметре функции, в общедоступной переменной, которую можно позднее проверить в teste.

Обратите внимание: я назвал фейковый объект не `MockLogger` и не `StubLogger`, а `FakeLogger`. Это сделано для того, чтобы класс можно было повторно использовать в нескольких тестах. В одних тестах он может использоваться как стаб, в других — как мок. Слово «fake» я использую для обозначения всего, что не является *реальным*. Другой стандартный термин для обозначения подобных концепций — «тестовый дублер». Термин «фейк» короче, поэтому я предпопчитаю его.

В своих тестах мы создаем экземпляр класса и передаем его в параметре конструктора, после чего выполняем проверку по переменной `logged` класса.

```
test("logger + passing scenario, calls logger with PASSED", () => {
  let logged = "";
  const mockLog = new FakeLogger();
  const verifier = new PasswordVerifier([], mockLog);
  verifier.verify("any input");

  expect(mockLog.logged).toMatch(/PASSED/);
});
```

4.7.2. Рефакторинг рабочего кода с внедрением интерфейсов

Интерфейсы играют важную роль во многих объектно-ориентированных программах. Их можно рассматривать как одну из вариаций на тему *полиморфизма*: объекты могут заменять друг друга при условии, что они реализуют один и тот же интерфейс. В JavaScript и других языках (например, Ruby) интерфейсы не нужны, потому что язык поддерживает идею «утиной типизации» без необходимости преобразования объекта к конкретному интерфейсу. Я не стану затрагивать достоинства и недостатки «утиной типизации». Вы должны иметь возможность использовать любой метод на ваше усмотрение в выбранном вами языке. В JavaScript для использования интерфейсов можно обратиться к TypeScript. Ваш *компилятор* (или *транспилятор*) поможет гарантировать правильность использования типов на основании их сигнатур.

В листинге 4.13 приведены три файла с программным кодом: первый описывает новый интерфейс `ILogger`, второй — простой класс `SimpleLogger`, реализующий этот интерфейс, а третий — наш класс `PasswordVerifier`, который использует только интерфейс `ILogger` для получения экземпляра логгера. `PasswordVerifier` не знает фактический тип внедряемого логгера.

Листинг 4.13. Рабочий код получает интерфейс ILogger

```

export interface ILogger {
    info(text: string);
}                                | Новый интерфейс,
                                | являющийся частью рабочего
                                | кода

//этот класс может иметь зависимости от файлов или сети
class SimpleLogger implements ILogger { ←
    info(text: string) {           | Теперь логгер реализует
        }                           | этот интерфейс
    }

export class PasswordVerifier { ←
    private _rules: any[];
    private _logger: ILogger;       | Теперь верификатор
                                    | использует этот
                                    | интерфейс
    constructor(rules: any[], logger: ILogger) { ←
        this._rules = rules;
        this._logger = logger;
    }

    verify(input: string): boolean { ←
        const failed = this._rules
            .map(rule => rule(input))
            .filter(result => result === false);

        if (failed.length === 0) { ←
            this._logger.info('PASSED');
            return true;
        }
        this._logger.info('FAIL');
        return false;
    }
}

```

Обратите внимание на некоторые изменения в рабочем коде. Я добавил в него новый интерфейс, и существующий логгер теперь реализует его. Структура была изменена, чтобы логгер можно было заменять. Кроме того, класс `PasswordVerifier` работает с интерфейсом вместо класса `SimpleLogger`. Это позволяет мне заменить экземпляр класса `logger` фейковым — вместо жесткой зависимости от реального логгера.

В листинге 4.14 показано, как может выглядеть тест в языке с сильной типизацией, но с написанным вручную фейковым объектом, реализующим интерфейс `ILogger`.

Листинг 4.14. Внедрение мока ILogger, реализованного вручную

```

class FakeLogger implements ILogger {
    written: string;
    info(text: string) {

```

```

        this.written = text;
    }
}

describe('password verifier with interfaces', () => {
    test('verify, with logger, calls logger', () => {
        const mockLog = new FakeLogger();
        const verifier = new PasswordVerifier([], mockLog);

        verifier.verify('anything');

        expect(mockLog.written).toMatch(/PASS/);
    });
});

```

В этом примере я создал класс `FakeLogger`, который был написан вручную. Делает он не так много: переопределяет один метод в интерфейсе `ILogger` и сохраняет параметр `text` для будущего утверждения. Это значение предоставляется в поле класса `written`. После этого вы сможете убедиться в том, что фейковый логгер был вызван, проверяя это поле.

Я сделал это вручную, потому что хотел показать, что даже в объектно-ориентированной области паттерны повторяются. Вместо *функции*-мока у нас теперь используется *мок*, но код и тест работают так же, как в предыдущих примерах.

Соглашения об именах интерфейсов

Я применяю схему назначения имен, в которой к интерфейсу логгера добавляется префикс «`I`», потому что это имя будет использоваться полиморфно (то есть я абстрагирую его роль в системе). В TypeScript такая схема именования интерфейсов не всегда применима. Например, когда мы используем интерфейс для определения структуры набора параметров (по сути, он будет являться сильно типизированной структурой), то имена без «`I`» кажутся мне более логичными.

А пока на это можно смотреть так: когда вы собираетесь реализовать интерфейс более одного раза, снабдите его префиксом «`I`», если хотите более явно выразить предполагаемое использование этого интерфейса.

4.8. СЛОЖНЫЕ ИНТЕРФЕЙСЫ

Что произойдет, если интерфейс имеет более сложную структуру, например, если он содержит более одной-двух функций или каждая функция получает более одного-двух параметров?

4.8.1. Пример сложного интерфейса

В листинге 4.15 приведен пример сложного интерфейса и рабочего кода, в котором используется сложный логгер, внедряемый как интерфейс. Интерфейс `IComplicatedLogger` содержит четыре функции, каждая из которых имеет один или несколько параметров. Для каждой функции в тестах придется определять фейк, что может привести к проблемам со сложностью и сопровождением кода и тестов.

Листинг 4.15. Работа со сложным интерфейсом (рабочий код)

```
export interface IComplicatedLogger {
    info(text: string)
    debug(text: string, obj: any)
    warn(text: string)
    error(text: string, location: string, stacktrace: string)
}
export class PasswordVerifier2 {
    private _rules: any[];
    private _logger: IComplicatedLogger;
    constructor(rules: any[], logger: IComplicatedLogger) {
        this._rules = rules;
        this._logger = logger;
    }
    ...
}
```

Как видите, новый интерфейс `IComplicatedLogger` будет частью рабочего кода, которая обеспечит возможность замены логгера. Реализацию реального логгера я не привожу, потому что для наших примеров она неважна. Это преимущество абстрагирования с использованием интерфейсов: к ним не нужно обращаться напрямую. Кроме того, обратите внимание на то, что типом параметра, ожидаемым в конструкторе класса, является интерфейс `IComplicatedLogger`. Это позволяет заменить экземпляр класса логгером, как это делалось ранее.

4.8.2. Написание тестов со сложными интерфейсами

В листинге 4.16 показано, как выглядит тест. Он должен переопределить все без исключения функции интерфейса, в результате чего вы получаете длинный и раздражающий шаблонный код.

Листинг 4.16. Код теста со сложным интерфейсом логгера

```
describe("working with long interfaces", () => {
    describe("password verifier", () => {
        class FakeComplicatedLogger
            implements IComplicatedLogger {
                infoWritten = "";
                // Фейковый класс логгера,
                // реализующий новый интерфейс
            }
    })
})
```

```
debugWritten = "";
errorWritten = "";
warnWritten = "";

debug(text: string, obj: any) {
    this.debugWritten = text;
}

error(text: string, location: string, stacktrace: string) {
    this.errorWritten = text;
}

info(text: string) {
    this.infoWritten = text;
}

warn(text: string) {
    this.warnWritten = text;
}
}

...

test("verify passing, with logger, calls logger with PASS", () => {
    const mockLog = new FakeComplicatedLogger();

    const verifier = new PasswordVerifier2([], mockLog);
    verifier.verify("anything");

    expect(mockLog.infoWritten).toMatch(/PASSED/);
});

test("A more JS oriented variation on this test", () => {
    const mockLog = {} as IComplicatedLogger;
    let logged = "";
    mockLog.info = (text) => (logged = text);

    const verifier = new PasswordVerifier2([], mockLog);
    verifier.verify("anything");

    expect(logged).toMatch(/PASSED/);
});
});
});
```

Здесь снова объявляется фейковый класс логгера (`FakeComplicatedLogger`), реализующий интерфейс `IComplicatedLogger`. Посмотрите, сколько здесь шаблонного кода. Это особенно справедливо при работе с объектно-ориентированными языками с сильной типизацией — такими, как Java, C или C++. От всего этого шаблонного кода можно избавиться, и об этом будет рассказано в следующей главе.

4.8.3. Недостатки прямого использования сложных интерфейсов

У использования длинных, сложных интерфейсов в тестах есть и другие недостатки.

- Если переданные аргументы сохраняются вручную, то проверка нескольких аргументов у нескольких методов и вызовов получается более громоздкой.
- Вероятно, ваш код зависит от сторонних, а не внутренних интерфейсов, и с течением времени ваши тесты станут более хрупкими.
- Даже если ваш код зависит от внутренних интерфейсов, у длинных интерфейсов больше причин для изменений, а следовательно, их будет больше и у ваших тестов.

Что это означает для нас? Я настоятельно рекомендую использовать только фейковые интерфейсы, удовлетворяющие двум условиям:

- вы контролируете интерфейсы (то есть они не созданы третьей стороной);
- они могут адаптироваться к потребностям вашей единицы работы или компонента.

4.8.4. Принцип разделения интерфейсов

Второе из представленных условий может потребовать дополнительных объяснений. Оно связано с *принципом разделения интерфейсов* (ISP, Interface Segregation Principle; https://en.wikipedia.org/wiki/Interface_segregation_principle). Этот принцип означает, что, если у вас имеется интерфейс, содержащий больше функциональности, чем требуется, следует создать уменьшенный, упрощенный интерфейс-адаптер, содержащий только нужную вам функциональность, желательно с меньшим количеством функций, более грамотно выбранными именами и меньшим количеством параметров.

Это значительно упростит ваши тесты. Если абстрагировать реальные зависимости, вам не придется изменять свои тесты при изменении сложных интерфейсов — только один файл класса-адаптера. Пример такого рода будет приведен в главе 5.

4.9. ЧАСТИЧНЫЕ МОКИ

В JavaScript, а также в большинстве других языков и сопутствующих им фреймворках тестирования можно взять существующие объекты и функции и начать «шпионить» за ними, иначе говоря, позднее вы сможете проверить, вызывались ли они, сколько раз и с какими аргументами.

Фактически при этом *части* реального объекта превращаются в функции-моки, тогда как остальные части остаются реальным объектом. При этом создаются более сложные и, как следствие, более хрупкие тесты, но иногда этот вариант может быть вполне жизнеспособным, особенно если вы имеете дело с унаследованным кодом (за дополнительной информацией обращайтесь к главе 12).

4.9.1. Пример частичного мока

В листинге 4.17 показано, как может выглядеть такой тест. Мы создаем реальный логгер, а потом просто переопределяем одну из существующих реальных функций с использованием нестандартной функции.

Листинг 4.17. Пример частичного мока

```
describe("password verifier with interfaces", () => {
  test("verify, with logger, calls logger", () => {
    const testableLog: RealLogger = new RealLogger(); ← Создает экземпляр
    let logged = "";
    testableLog.info = (text) => (logged = text); ← Создает мок для одной
    // из его функций
    const verifier = new PasswordVerifier([], testableLog);
    verifier.verify("any input");

    expect(logged).toMatch(/PASSED/);
  });
});
```

В этом teste я создаю экземпляр `RealLogger`, а в следующей строке заменяю одну из существующих функций фейковой. А если говорить конкретнее, я использую функцию-мок, которая позволяет отслеживать параметр ее последнего вызова в специальной переменной.

Здесь важно то, что переменная `testableLog` является *частичным моком*. Это означает, что, по крайней мере, часть ее внутренней реализации не является фейковой, может иметь реальные зависимости и логику.

Иногда в использовании частичных моков есть смысл, особенно когда вы работаете с унаследованным кодом и вам нужно изолировать часть существующего кода от его зависимостей. Эта тема будет более подробно рассмотрена в главе 12.

4.9.2. Объектно-ориентированный пример частичного мока

Одна объектно-ориентированная версия частичного мока использует наследование для переопределения функций из реальных классов, чтобы мы могли

убедиться в том, что они были вызваны. В листинге 4.18 показано, как это можно сделать с использованием наследования и переопределений в JavaScript.

Листинг 4.18. Объектно-ориентированный пример частичного мока

```
class TestableLogger extends RealLogger { ← Наследует от реального
    logged = "";
    info(text) {
        this.logged = text; | Переопределяет одну
    } | из его функций
    // функции error() и debug()
    // остаются "реальными"
}

describe("partial mock with inheritance", () => {
    test("verify with logger, calls logger", () => {
        const mockLog: TestableLogger = new TestableLogger();

        const verifier = new PasswordVerifier([], mockLog);
        verifier.verify("any input");

        expect(mockLog.logged).toMatch(/PASSED/);
    });
});
```

В своих тестах я наследую от реального класса логгера, а затем использую в тестах производный класс вместо исходного. Этот прием обычно называется «извлечь и переопределить» (Extract and Override); о нем можно больше узнать в книге Майкла Физерса «Working Effectively with Legacy Code».

Обратите внимание: я присвоил фейковому классу логгера имя «`TestableXXX`», потому что это тестируемая версия реального рабочего кода, содержащая смесь фейкового и реального кода, и такое соглашение помогает мне ясно донести этот факт для читателя. Я также разместил класс непосредственно рядом с тестами. Моему рабочему коду не нужно знать о том, что этот класс существует. Стиль «извлечь и переопределить» требует, чтобы мой класс в рабочем коде допускал наследование, а функция допускала переопределение. В JavaScript эта проблема не столь актуальна, но в Java и C# это явные решения из области проектирования, которые необходимо принять (хотя существуют фреймворки, позволяющие обойти это правило; мы обсудим их в следующей главе).

В этом сценарии мы наследуем от класса, который не тестируется напрямую (`RealLogger`). Мы используем этот класс для тестирования другого класса (`PasswordVerifier`). Тем не менее этот метод может вполне эффективно использоваться для изоляции и создания стабов (или моков) отдельных функций из классов, которые тестируются напрямую. Эта тема будет более подробно рассмотрена далее в книге, когда речь пойдет об унаследованном коде и методах рефакторинга.

ИТОГИ

- *Тестирование взаимодействий* — механизм проверки взаимодействий единицы работы с исходящими зависимостями: какие вызовы совершаются и с какими параметрами. Тестирование взаимодействий относится к третьему типу точек выхода: сторонние модули, объекты или системы. (Первые два типа — возвращаемое значение и изменение состояния.)
- Для тестирования взаимодействий используются *моки* — тестовые дублеры, заменяющие исходящие зависимости. *Стабы* заменяют входящие зависимости. Для проверки взаимодействий в тестах используются моки, но не стабы. В отличие от моков, взаимодействия со стабами являются подробностями реализации, и проверяться они не должны.
- Тест может содержать несколько стабов, но обычно не более одного мока, потому что это означало бы, что вы тестируете более одного требования в одном teste.
- Как и в случае со стабами, существует несколько способов внедрения мока в единицу работы:
 - *стандартный* — основанный на внедрении параметра;
 - *функциональный* — использующий частичное применение или фабричные функции;
 - *модульный* — абстрагирование модульных зависимостей;
 - *объектно-ориентированный* — использующий нетипизированный объект (в таких языках, как JavaScript) или типизированный интерфейс (в TypeScript).
- В JavaScript сложный интерфейс может быть реализован частично, что помогает сократить объем шаблонного кода. Также существует возможность создания *частичных моков*, когда вы наследуете от реального класса и заменяете только часть его методов фейками.

5

Изолирующие фреймворки

В ЭТОЙ ГЛАВЕ

- ✓ Определение изолирующих фреймворков и их предназначение
- ✓ Две основные разновидности фреймворков
- ✓ Создание фейков для модулей в Jest
- ✓ Создание фейков для функций в Jest
- ✓ Объектно-ориентированные фейки с `substitute.js`

В предыдущих главах мы рассмотрели ручное написание моков и стабов, а также возникающие при этом проблемы: когда интерфейс, для которого вы хотите создать фейки, требовал создания длинного, ненадежного и повторяющегося кода. Нам приходилось объявлять специальные переменные, создавать специальные функции или наследовать от классов, использующих эти переменные, что, по сути, делало работу более сложной, чем необходимо (в большинстве случаев).

В этой главе будут рассмотрены некоторые элегантные решения таких проблем с помощью *изолирующих фреймворков* – библиотек, пригодных для повторного использования, которые позволяют создавать и настраивать фейковые объекты

во время выполнения. Такие объекты называются *динамическими стабами и динамическими моками.*

Я использую термин «изолирующие фреймворки», потому что они позволяют изолировать единицу работы от ее зависимостей. Во многих источниках они называются «мок-фреймворками», но я стараюсь избегать этого названия, потому что они могут использоваться как с моками, так и со стабами. В этой главе мы рассмотрим некоторые существующие фреймворки JavaScript и возможности их использования в модульных, функциональных и объектно-ориентированных стилях. Вы увидите, как применять такие фреймворки для тестирования разных условий и создания стабов, моков и других интересных конструкций.

Тем не менее суть не в конкретных фреймворках, которые я здесь представлю. При их использовании вы увидите, что благодаря этим API ваши тесты приобретают положительные характеристики (читабельность, простота сопровождения, надежность, долговечность и т. д.). Кроме того, вы узнаете, какими особенностями обладает хороший изолирующий фреймворк, и, наоборот, при каких условиях он может стать препятствием для тестирования.

5.1. ОПРЕДЕЛЕНИЕ ИЗОЛИРУЮЩИХ ФРЕЙМВОРКОВ

Начну с базового определения, которое может показаться несколько бесцветным. Тем не менее оно неизбежно должно быть обобщенным, чтобы включать все изолирующие фреймворки.

Изолирующий фреймворк — это набор программируемых API, обеспечивающих возможность динамического создания, настройки и верификации моков и стабов в форме объекта или функции. При использовании изолирующего фреймворка эти задачи часто выполняются проще и быстрее, а код получается более коротким, чем при ручном кодировании моков и стабов.

При правильном использовании изолирующие фреймворки могут избавить разработчика от необходимости писать повторяющийся код для проверки или моделирования взаимодействий между объектами, а при использовании в правильных местах с ними можно создать тесты, которые будут существовать годами, не требуя исправлений после каждого незначительного изменения в рабочем коде. При неправильном использовании они могут создать путаницу и хаос вплоть до того, что вы либо не сможете читать собственные тесты, либо не сможете доверять им. Так что будьте осторожны. Некоторые правила и анти-примеры обсуждаются в части 3 этой книги.

5.1.1. Выбор разновидности: свободные и типизированные

Так как JavaScript поддерживает разные парадигмы программирования, фреймворки можно разбить на две основные категории.

- *Свободные изолирующие фреймворки JavaScript* — базовые изолирующие фреймворки, ориентированные на JavaScript (такие, как Jest и Sinon). Эти фреймворки также обычно лучше подходят для более функциональных стилей программирования, потому что они требуют меньше церемоний и шаблонного кода для выполнения своей работы.
- *Типизированные изоляционные фреймворки JavaScript* — более объектно-ориентированные и TypeScript-совместимые изолирующие фреймворки (такие, как `substitute.js`). Они чрезвычайно удобны при работе с целыми классами и интерфейсами.

Выбор разновидности, которую вы в итоге будете использовать в своем проекте, зависит от ряда факторов (включая личные предпочтения, стиль и читабельность), но начинать следует с главного вопроса: для какого типа зависимостей вам в основном придется создавать фейки?

- *Модульные зависимости (`import`, `require`)* — Jest и другие фреймворки со свободной типизацией должны сработать достаточно хорошо.
- *Функциональные зависимости (отдельные функции и функции высшего порядка, простые параметры и значения)* — скорее всего, вам подойдут Jest и другие фреймворки со свободной типизацией.
- *Полные объекты, иерархии объектов и интерфейсы* — обратите внимание на более объектно-ориентированные фреймворки (такие, как `substitute.js`).

Вернемся к Password Verifier и посмотрим, как можно имитировать те же типы зависимостей, которые уже встречались нам в предыдущих главах, но на этот раз с использованием фреймворка.

5.2. ДИНАМИЧЕСКОЕ СОЗДАНИЕ ФЕЙКОВ ДЛЯ МОДУЛЕЙ

Для людей, которые пытаются тестировать код с прямыми зависимостями от модулей, использующими `require` или `import`, изолирующие фреймворки (такие, как Jest или Sinon) предоставляют мощные средства динамического создания фейков для целых модулей с минимумом кода. Так как изначально мы выбрали Jest как основной фреймворк тестирования, будем придерживаться данного выбора в примерах этой главы.

На рис. 5.1 изображена архитектура Password Verifier с двумя зависимостями:

- сервис конфигурации, который помогает выбрать уровень вывода в логе (INFO или ERROR);
- сервис логгинга, который вызывается как точка выхода нашей единицы работы при проверке пароля.

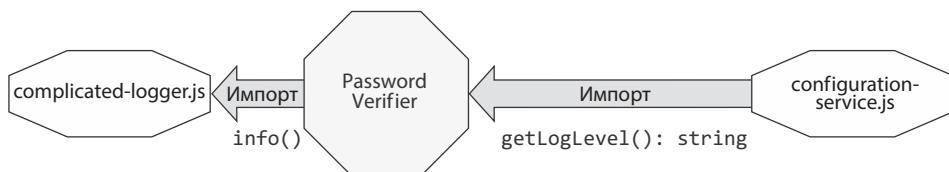


Рис. 5.1. Password Verifier имеет две зависимости: входящую для определения уровня вывода и исходящую для создания записи журнала

Стрелки представляют поток поведения в единице работы. Также можно рассматривать стрелки в контексте терминов *команда* и *запрос*. Мы обращаемся с запросом к сервису конфигурации (для получения уровня вывода), но отправляем команды логгеру.

Разделение команд и запросов

Существует школа проектирования, ориентированная на идеи разделения команд и запросов (command/query separation). Если вам захочется больше узнать об этих терминах, я настоятельно рекомендую статью Мартина Фаулера 2005 года (<https://martinfowler.com/bliki/CommandQuerySeparation.html>). Этот паттерн чрезвычайно полезен, когда вы анализируете разные идеи проектирования, но в нашей книге мы не будем уделять ему особого внимания.

В листинге 5.1 представлен код Password Verifier с жесткой зависимостью от модуля логгера.

Листинг 5.1. Код с жестко запрограммированными модульными зависимостями

```

const { info, debug } = require("./complicated-logger");
const { getLogLevel } = require("./configuration-service");

const log = (text) => {
  if (getLogLevel() === "info") {
    info(text);
  }
}
  
```

```

if (getLogLevel() === "debug") {
  debug(text);
}

const verifyPassword = (input, rules) => {
  const failed = rules
    .map((rule) => rule(input))
    .filter((result) => result === false);

  if (failed.length === 0) {
    log("PASSED");
    return true;
  }
  log("FAIL");
  return false;
};

```

В этом примере нам придется понять, как решить две задачи.

- Смоделировать значения, возвращаемые функцией `getLogLevel` сервиса `configuration` (стаб).
- Убедиться в том, что функция `info` модуля `logger` была вызвана (мок).

На рис. 5.2 наглядно изображены эти задачи.

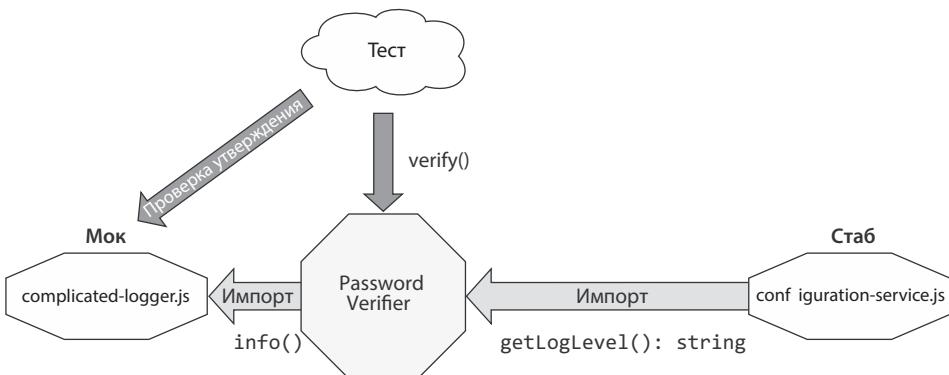


Рис. 5.2. Тест создает стаб для входящей зависимости (сервис конфигурации) и мок для исходящей зависимости (логгер)

Jest предоставляет несколько способов реализации как моделирования, так и проверки, и один из самых удобных вариантов основан на включении вызова `jest.mock([имя модуля])` в начале файла, за которым следуют вызовы `require` для фейковых модулей в наших тестах, чтобы их можно было настроить.

Листинг 5.2. Прямое создание фейка для модульных API с использованием jest.mock()

```

jest.mock("./complicated-logger");
jest.mock("./configuration-service");           | Создает фейк для модулей логгера

const { stringMatching } = expect;
const { verifyPassword } = require("./password-verifier");
const mockLoggerModule = require("./complicated-logger");
const stubConfigModule = require("./configuration-service"); ← | Получает фейковые
                                                               экземпляры модулей

describe("password verifier", () => {
  afterEach(jest.resetAllMocks); ← | Приказывает Jest сбросить
                                         поведение фейкового
                                         модуля между тестами

  test('with info log level and no rules,
        it calls the logger with PASSED', () => {
    stubConfigModule.getLogLevel.mockReturnValue("info"); ← | Настраивает стаб
                                                               для возврата
                                                               фейкового значения
                                                               "info"

    verifyPassword("anything", []);
  });

  expect(mockLoggerModule.info)
    .toHaveBeenCalledWith(stringMatching(/PASS/));           | Проверяет, что мок
                                                               был вызван правильно

});

test('with debug log level and no rules,
      it calls the logger with PASSED', () => {
  stubConfigModule.getLogLevel.mockReturnValue("debug"); ← | Изменяет
                                                               конфигурацию
                                                               стаба

  verifyPassword("anything", []);

  expect(mockLoggerModule.debug)
    .toHaveBeenCalledWith(stringMatching(/PASS/));           | Выполняет проверку
                                                               по фейковому логгеру,
                                                               как делалось ранее
});

```

Использование Jest избавляет от необходимости вводить большой объем кода, и тесты все еще нормально читаются.

5.2.1. Некоторые особенности API Jest, на которые следует обратить внимание

Jest использует слово «mock» («мок») практически везде — как для моков, так и для стабов, что может создать определенную путаницу. Было бы замечательно, если бы в Jest было слово «stub» («стаб») наряду со словом «мок», чтобы код лучше читался.

Кроме того, из-за особенностей работы механизма *поднятия* (hoisting) в JavaScript строки, в которых создаются фейки для модулей (посредством `jest.mock`), должны находиться в начале файла. Об этом можно подробнее

прочитать в статье Ашутоша Верма (Ashutosh Verma) «Understanding Hoisting in JavaScript» по адресу <http://mng.bz/j11r>.

В Jest есть много других API и механизмов, которые стоит изучить, если вас интересует эта тема. Чтобы представить полную картину, зайдите на сайт <https://jestjs.io/>; эта тема выходит за рамки нашей книги, которая посвящена паттернам, а не инструментам.

Ряд других фреймворков, в числе которых Sinon (<https://sinonjs.org>), также поддерживает фейковые модули. Работать с Sinon довольно приятно, насколько это может быть возможно с изолирующими фреймворками, но, как и многие другие фреймворки из мира JavaScript (и как Jest), он предоставляет слишком много разных способов решения одной задачи, что часто создает путаницу. Тем не менее без этих фреймворков создавать фейки для модулей вручную довольно утомительно.

5.2.2. Абстрагирование прямых зависимостей

API `jest.mock` и других похожих фреймворков отвечает совершенно реальным потребностям разработчиков, тех, кому нужно протестировать модули со встроенными зависимостями, которые нельзя просто заменить (то есть они не контролируются). Эта проблема очень часто встречается в ситуациях с унаследованным кодом, о чём будет рассказано в главе 12.

С другой стороны, API `jest.mock` также позволяет создавать моки для кода, который находится под вашим контролем и который мог бы выиграть от абстрагирования реальных зависимостей за более простыми, короткими, внутренними API. Такой подход, также называемый *гексагональной* или *луковой архитектурой* (также *порты и адаптеры*), чрезвычайно полезен для обеспечения долгосрочной сопровождаемости кода. За дополнительной информацией об архитектурах такого типа обращайтесь к статье Алистера Кокберна (Alistair Cockburn) «Hexagonal Architecture» по адресу <https://alistair.cockburn.us/hexagonal-architecture/>.

Почему прямые зависимости создают потенциальные проблемы? Используя эти API напрямую, мы также вынуждены создавать фейки для модульных API прямо в наших тестах вместо их абстракций. Структура прямых API жестко связывается с реализацией тестов, а следовательно, если (а вернее, когда) эти API изменятся, нам также придется изменять многие свои тесты.

Рассмотрим короткий пример. Допустим, ваш код зависит от популярного фреймворка логирования (например, Winston), причем зависит напрямую в сотнях и тысячах мест в коде. Затем представьте, что для Winston выходит критическое обновление. Это создаст массу проблем, которых можно было бы избежать, приняв необходимые меры заранее и не допуская выхода

ситуации из-под контроля. Одно из решений — простая абстракция в отдельном файле адаптера, который станет единственным местом, содержащим ссылку на логгер. Такая абстракция может предоставлять более простой внутренний API логгинга, который находится под вашим контролем; тем самым предотвращаются крупномасштабные нарушения в коде. Мы вернемся к этой теме в главе 12.

5.3. ФУНКЦИОНАЛЬНЫЕ ДИНАМИЧЕСКИЕ МОКИ И СТАБЫ

Мы рассмотрели модульные зависимости, теперь перейдем к созданию фейков для простых функций. Это уже неоднократно происходило в предыдущих главах, но мы всегда делали это вручную. Такое решение отлично работает для стабов, но для моков оно быстро начинает раздражать.

В листинге 5.3 продемонстрирован ручной подход, который использовался ранее.

Листинг 5.3. Ручное создание мока функции для проверки того, что она была вызвана

```
test("given logger and passing scenario", () => {
  let logged = "";
  const mockLog = { info: (text) => (logged = text) };
  const passVerify = makeVerifier([], mockLog);

  passVerify("any input");

  expect(logged).toMatch(/PASSED/);
});
```

Объявляет переменную для хранения переданного значения

Сохраняет переданное значение в этой переменной

Проверка по значению переменной

Решение работает: мы можем убедиться в том, что функция логгера была вызвана, но это серьезная работа, которая быстро начинает утомлять. На помощь приходят изолирующие фреймворки — такие, как Jest. `jest.fn()`, — самый простой способ избавиться от такого кода. В листинге 5.4 показано, как это делается.

Листинг 5.4. Использование `jest.fn()` для простых моков-функций

```
test('given logger and passing scenario', () => {
  const mockLog = { info: jest.fn() };
  const verify = makeVerifier([], mockLog);

  verify('any input');

  expect(mockLog.info)
    .toHaveBeenCalledWith(stringMatching(/PASS/));
});
```

Сравните этот код с предыдущим примером. Различия нетривиальны, но они экономят много времени. Здесь мы используем `jest.fn()` для получения функции, которая автоматически отслеживается Jest, так что можем позднее запросить информацию о ней через Jest API вызовом `toHaveBeenCalledWith()`. Решение получается компактным и элегантным и хорошо работает во всех случаях, когда требуется отслеживать вызовы конкретной функции. Функция `stringMatching` является примером *сопоставителя* (matcher). Сопоставитель обычно определяется как вспомогательная функция, которая может выполнить проверку по значению параметра, переданного функции. В документации Jest термин используется достаточно свободно, а полный список сопоставителей в Jest доступен по адресу <https://jestjs.io/docs/en/expect>.

Итак, `jest.fn()` хорошо работает для моков и стабов, основанных на отдельных функциях. Переходим к более объектно-ориентированным проблемам.

5.4. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ ДИНАМИЧЕСКИЕ МОКИ И СТАБЫ

Как было показано ранее, `jest.fn()` является примером вспомогательной функции, создающей фейки для одиночных функций. Она хорошо работает в функциональном мире, но все меняется при попытке использовать ее с полноценными интерфейсами API или классами, содержащими много функций.

5.4.1. Использование фреймворка со свободной типизацией

Ранее я упоминал о том, что изолирующие фреймворки делятся на две категории. Начнем с первой категории — со свободной типизацией. В листинге 5.5 представлена попытка решения проблемы с `IComplicatedLogger`, рассмотренной в предыдущей главе.

Листинг 5.5. Интерфейс `IComplicatedLogger`

```
export interface IComplicatedLogger {
  info(text: string, method: string)
  debug(text: string, method: string)
  warn(text: string, method: string)
  error(text: string, method: string)
}
```

На ручное создание стаба или мока для этого интерфейса может потребоваться очень много времени, потому что вам придется вспомнить параметры каждого отдельного метода, как показано в листинге 5.6.

Листинг 5.6. Стабы, написанные вручную, создают большой объем шаблонного кода

```
describe("working with long interfaces", () => {
  describe("password verifier", () => {
    class FakeLogger implements IComplicatedLogger {
      debugText = "";
      debugMethod = "";
      errorText = "";
      errorMethod = "";
      infoText = "";
      infoMethod = "";
      warnText = "";
      warnMethod = "";

      debug(text: string, method: string) {
        this.debugText = text;
        this.debugMethod = method;
      }

      error(text: string, method: string) {
        this.errorText = text;
        this.errorMethod = method;
      }
      ...
    }

    test("verify, w logger & passing, calls logger with PASS", () => {
      const mockLog = new FakeLogger();
      const verifier = new PasswordVerifier2([], mockLog);

      verifier.verify("anything");

      expect(mockLog.infoText).toMatch(/PASSED/);
    });
  });
});
```

Мало того, что написание этого фейка вручную потребует значительного времени и усилий. А если вам понадобится, чтобы он вернул в тесте конкретное значение, или вы захотите смоделировать ошибку при вызове функции логгера? Это можно сделать, но код стремительно становится все более уродливым.

При использовании изолирующего фреймворка код для решения этой задачи будет простым, читабельным и коротким. Воспользуемся `jest.fn()` для той же задачи и посмотрим, что из этого выйдет.

Листинг 5.7. Создание моков для отдельных функций интерфейсов с jest.fn()

```

import stringMatching = jasmine.stringMatching;

describe("working with long interfaces", () => {
  describe("password verifier", () => {
    test("verify, w logger & passing, calls logger with PASS", () => {
      const mockLog: IComplicatedLogger = {
        info: jest.fn(),
        warn: jest.fn(),
        debug: jest.fn(),
        error: jest.fn(),
      };
      const verifier = new PasswordVerifier2([], mockLog);
      verifier.verify("anything");

      expect(mockLog.info)
        .toHaveBeenCalledWith(stringMatching(/PASS/));
    });
  });
});

```

Создание мока
с использованием
Jest

В целом неплохо. Здесь мы просто описываем свой объект и присоединяем функцию `jest.fn()` к каждой функции интерфейса. Это избавляет от необходимости вводить большой объем кода, но есть один важный момент: каждый раз, когда интерфейс изменяется (например, в него добавляется функция), нам придется возвращаться к коду, в котором определяется этот объект, и добавлять новую функцию. С базовым JavaScript проблема будет не столь серьезной, но она все равно может создать значительные затруднения, если в тестируемом коде используется функция, которая не была определена в тесте.

В любом случае может быть разумно вынести создание такого фейкового объекта в фабричный вспомогательный метод, чтобы создание находилось только в одном месте.

5.4.2. Переход на фреймворк с поддержкой типов

Обратимся ко второй категории фреймворков и опробуем `substitute.js` (www.npmjs.com/package/@fluffy-spoon/substitute). Необходимо выбрать один вариант; мне нравится версия этого фреймворка для C#, которая использовалась в предыдущем издании книги.

С `substitute.js` (и предположением, что вы применяете TypeScript) можно написать код следующего вида:

Листинг 5.8. Использование substitute.js с созданием фейка для целого интерфейса

```
import { Substitute, Arg } from "@fluffy-spoon/substitute";

describe("working with long interfaces", () => {
  describe("password verifier", () => {
    test("verify, w logger & passing, calls logger w PASS", () => {
      const mockLog = Substitute.for<IComplicatedLogger>(); ← Генерирует
      const verifier = new PasswordVerifier2([], mockLog);   фейковый объект
      verifier.verify("anything");

      mockLog.received().info(   Убеждается в том,
        Arg.is((x) => x.includes("PASSED")),   что фейковый
        "verify"                                объект был вызван
      );
    });
  });
});
```

В этом листинге мы генерируем фейковый объект, что избавляет от необходимости заниматься другими функциями, кроме тестируемых, даже если сигнатура объекта в будущем изменится. Затем используется `.received()` как механизм верификации, а также другой сопоставитель аргументов `Arg.js` (на этот раз из API `substitute.js`), который работает так же, как поиск совпадений в строках из `Jasmine`. Дополнительное преимущество: при добавлении новых функций в сигнатуру объекта снижается вероятность того, что вам придется изменять тест, и нет необходимости добавлять эти функции в какие-либо тесты, использующие ту же сигнатуру объекта.

Изолирующие фреймворки и паттерн AAA

Обратите внимание: способ использования изолирующего фреймворка неплохо соответствует структуре AAA (Arrange-Act-Assert, «подготовь-действуй-проверь»), рассмотренной в главе 1. Вы начинаете с подготовки фейкового объекта, затем действуете с тем, что тестируете, и выполняете проверку по некоторому критерию в конце теста.

Впрочем, так удобно было не всегда. В старые времена (около 2006 года) большинство изолирующих фреймворков с открытым кодом не поддерживали концепцию AAA и вместо этого использовали концепцию, которая называлась Record-Replay, то есть «запись — воспроизведение»

(речь идет о Java и C#). Это был неудобный механизм, в котором приходилось сообщать изолирующему API, что фейковый объект находится в режиме записи, а затем вызывать для этого объекта методы так, как они должны были вызываться из рабочего кода. Затем нужно было приказать изолирующему API переключиться в режим воспроизведения, и только после этого вы могли передать свой фейковый объект рабочему коду. С примером можно ознакомиться на сайте Baeldung по адресу www.baeldung.com/easymock.

По сравнению с сегодняшними возможностями написания тестов, использующих намного более читабельную модель AAA, концепция Record-Replay была настоящей бедой и стоила разработчикам миллионов часов мучительного чтения тестов в попытках разобраться, где же что-то пошло не так.

Если у вас есть первое издание этой книги, вы можете ознакомиться с примером использования Record-Replay с фреймворком Rhino Mocks (который изначально использовал тот же механизм).

Ладно, с моками понятно. А как насчет стабов?

5.5. ДИНАМИЧЕСКОЕ СОЗДАНИЕ СТАБОВ

В Jest используется очень простой API, чтобы моделировать возвращаемые значения для модульных и функциональных зависимостей: `mockReturnValue()` и `mockReturnValueOnce()`.

Листинг 5.9. Создание стаба для возвращаемого значения фейковой функции с использованием `jest.fn()`

```
test("fake same return values", () => {
  const stubFunc = jest.fn()
    .mockReturnValue("abc");

  //значение остается тем же
  expect(stubFunc()).toBe("abc");
  expect(stubFunc()).toBe("abc");
  expect(stubFunc()).toBe("abc");
});

test("fake multiple return values", () => {
  const stubFunc = jest.fn()
    .mockReturnValueOnce("a")
    .mockReturnValueOnce("b")
    .mockReturnValueOnce("c");
});
```

```
//значение остается тем же
expect(stubFunc()).toBe("a");
expect(stubFunc()).toBe("b");
expect(stubFunc()).toBe("c");
expect(stubFunc()).toBe(undefined);
});
```

Заметим, что в первом тесте мы назначаем *постоянное* возвращаемое значение на всю продолжительность теста. Я предпочитаю этот метод написания тестов, если возможно, потому что он легко читается и сопровождается. Если вам нужно смоделировать несколько значений, используйте `mockReturnValueOnce()`.

Если необходимо смоделировать ошибку или сделать что-то более сложное, используйте `mockImplementation()` и `mockImplementationOnce()`:

```
yourStub.mockImplementation(() => {
  throw new Error();
});
```

5.5.1. Объектно-ориентированный пример с моком и стабом

Немного усложним поведение Password Verifier.

- Допустим, программа Password Verifier неактивна в течение определенного периода, когда выполняется обновление ПО, так называемого окна обслуживания (maintenance window).
- Когда окно обслуживания активно, вызов `verify()` для `Verifier` должен приводить к вызову `logger.info()` с сообщением «under maintenance» («происходит обслуживание»).
- В остальных случаях `logger.info()` вызывается с результатом «passed» («прошел») или «failed» («не прошел»).

Для этой цели (и для демонстрации решений по объектно-ориентированному проектированию) мы введем интерфейс `MaintenanceWindow`, который будет внедряться в конструктор `Password Verifier`, как показано на рис. 5.3.

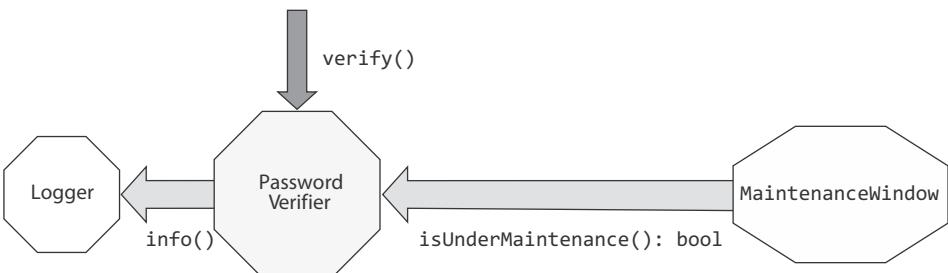


Рис. 5.3. Использование интерфейса `MaintenanceWindow`

В листинге 5.10 приведен код Password Verifier с использованием новой зависимости.

Листинг 5.10. Password Verifier с зависимостью MaintenanceWindow

```
export class PasswordVerifier3 {
    private _rules: any[];
    private _logger: IComplicatedLogger;
    private _maintenanceWindow: MaintenanceWindow;

    constructor(
        rules: any[],
        logger: IComplicatedLogger,
        maintenanceWindow: MaintenanceWindow
    ) {
        this._rules = rules;
        this._logger = logger;
        this._maintenanceWindow = maintenanceWindow;
    }

    verify(input: string): boolean {
        if (this._maintenanceWindow.isUnderMaintenance()) {
            this._logger.info("Under Maintenance", "verify");
            return false;
        }
        const failed = this._rules
            .map((rule) => rule(input))
            .filter((result) => result === false);

        if (failed.length === 0) {
            this._logger.info("PASSED", "verify");
            return true;
        }
        this._logger.info("FAIL", "verify");
        return false;
    }
}
```

Интерфейс `MaintenanceWindow` внедряется как параметр конструктора (то есть используется внедрение через конструктор) и применяется для определения того, где должна выполняться (или не выполняться) проверка пароля, и для отправки соответствующего сообщения логгеру.

5.5.2. Стабы и моки с `substitute.js`

Теперь мы используем `substitute.js` вместо Jest для создания стаба интерфейса `MaintenanceWindow` и мока интерфейса `IComplicatedLogger`. На рис. 5.4 показано, как это происходит.

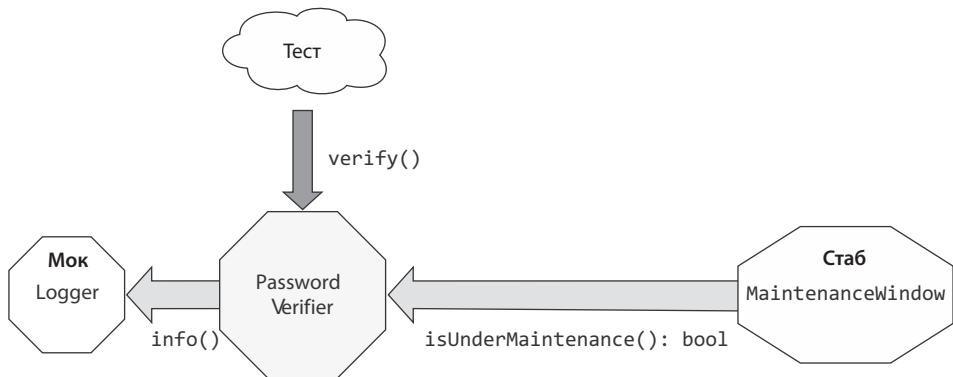


Рис. 5.4. Зависимость MaintenanceWindow

Создание стабов и моков с `substitute.js` происходит одинаково: для них используется функция `Substitute.for<T>`. Стабы могут настраиваться функцией `.returns`, а для верификации моков может использоваться функция `.received`. Обе функции являются частью фейкового объекта, возвращаемого `Substitute.for<T>()`.

Вот как происходит создание и настройка стаба:

```
const stubMainWindow = Substitute.for<MaintenanceWindow>();
stubMainWindow.isUnderMaintenance().returns(true);
```

Создание мока и верификация выглядят так:

```
const mockLog = Substitute.for<IComplicatedLogger>();
...
/// позднее в конце теста...
mockLog.received().info("Under Maintenance", "verify");
```

В листинге 5.11 приведен полный код двух тестов, в которых используются мок и стаб.

Листинг 5.11. Тестирование Password Verifier с substitute.js

```
import { Substitute } from "@fluffy-spoon/substitute";

const makeVerifierWithNoRules = (log, maint) =>
  new PasswordVerifier3([], log, maint);

describe("working with substitute part 2", () => {
  test("verify, during maintenance, calls logger", () => {
    const stubMainWindow = Substitute.for<MaintenanceWindow>();
```

```

stubMainWindow.isUnderMaintenance().returns(true);
const mockLog = Substitute.for<IComplicatedLogger>();
const verifier = makeVerifierWithNoRules(mockLog, stubMainWindow);

verifier.verify("anything");

mockLog.received().info("Under Maintenance", "verify");
});

test("verify, outside maintanance, calls logger", () => {
  const stubMainWindow = Substitute.for<MaintenanceWindow>();
  stubMainWindow.isUnderMaintenance().returns(false);
  const mockLog = Substitute.for<IComplicatedLogger>();
  const verifier = makeVerifierWithNoRules(mockLog, stubMainWindow);

  verifier.verify("anything");

  mockLog.received().info("PASSED", "verify");
});
});

```

Динамически создаваемые объекты позволяют успешно и относительно легко моделировать значения в наших тестах. Я рекомендую изучить тот изолирующий фреймворк, который вам хотелось бы использовать. В этой книге `substitute.js` приводится только в качестве примера. Это далеко не единственный существующий фреймворк.

Этот тест не требует ручной реализации фейков, но стоит заметить, что читабельность кода уже оставляет желать лучшего. Функциональные дизайны обычно намного компактнее. В объектно-ориентированной среде это иногда оказывается неизбежным злом. С другой стороны, мы можем легко провести рефакторинг создания различных вспомогательных методов, моков и стабов, чтобы тест был проще и лучше читался. Подробнее об этом в части 3 данной книги.

5.6. ПРЕИМУЩЕСТВА И ОПАСНОСТИ ИЗОЛИРУЮЩИХ ФРЕЙМВОРКОВ

После всего, что было рассказано в этой главе, мы видим основные преимущества изолирующих фреймворков.

- *Простота создания модульных фейков* — с модульными зависимостями бывает трудно работать без шаблонного кода, а изолирующие фреймворки помогают такой код устраниить. Впрочем, этот пункт также можно отнести к недостаткам, потому что он подталкивает нас к сильному связыванию (coupling) кода со сторонними реализациями.
- *Простота моделирования значений или ошибок* — ручное написание моков для сложных интерфейсов может быть непростым делом. Фреймворки в этом помогают.

- *Простота создания фейков* — изолирующие фреймворки могут использоватьсь для упрощения создания как моков, так и стабов.

Хотя у изолирующих фреймворков есть много преимуществ, при работе с ними также возникают некоторые риски. Ниже перечислены возможные проблемы, заслуживающие особого внимания.

5.6.1. Моки в большинстве случаев не нужны

Самая большая ловушка, в которую вас может завести использование изолирующих фреймворков, — уверенность в том, что вы можете легко создать фейк для чего угодно, и мысль о том, что в первую очередь вам нужны моки. Я не говорю, что стабы не нужны, но моки не должны быть обычным делом для большинства юнит-тестов. Помните, что единица работы может иметь три разновидности точек выхода: возвращаемые значения, изменения состояния и вызов сторонней зависимости. Только один из этих типов выигрывает от присутствия мока в вашем тесте. В двух других они не нужны.

В моих тестах моки присутствуют, возможно, в 2–5 % случаев. Остальные тесты обычно являются тестами возвращаемого значения или тестами состояний. В функциональных структурах количество моков должно быть почти нулевым, не считая некоторых граничных случаев.

Если вы пишете тест, чтобы убедиться в том, что объект или функция были вызваны, хорошенько подумайте, можно ли доказать ту же функциональность не с помощью мока, а с верификацией возвращаемого значения или изменения в поведении всей единицы работы снаружи (например, что функция выдает исключение там, где ранее она этого не делала). В главе 6 книги «Unit Testing Principles, Practices, and Patterns»¹ Владимира Хорикова (Manning, 2020) содержится подробное описание того, как проводить рефакторинг тестов на основе взаимодействий в более простые и надежные тесты, которые вместо этого проверяют возвращаемое значение.

5.6.2. Нечитаемый код теста

Использование мока в тесте несколько затрудняет чтение, но тест все еще остается достаточно понятным, чтобы посторонний мог просмотреть код и понять, что здесь происходит. Присутствие многих моков (или многих ожиданий) в одном тесте может испортить читабельность теста настолько, что его будет трудно сопровождать или даже просто понять, что здесь тестируется.

Если вы обнаруживаете, что ваш тест стал нечитабельным или его логика стала непонятной, рассмотрите возможность удаления некоторых моков или ожиданий моков либо разделения теста на несколько меньших, более понятных тестов.

¹ Хориков В. Принципы юнит-тестирования. СПб.: Питер.

5.6.3. Проверка не того, что нужно

Моки позволяют убедиться в том, что методы ваших интерфейсов (или отдельные функции) были вызваны, но это не обязательно означает, что вы тестируете именно то, что нужно. Многие разработчики, не имеющие опыта тестирования, проверяют все подряд просто потому, что могут, а не потому, что это имеет смысл. Несколько примеров:

- Проверка того, что внутренняя функция вызывает другую внутреннюю функцию (а не точку выхода).
- Проверка того, что стаб был вызван (входящие зависимости не должны проверяться; это антипаттерн «излишняя детализация», рассматриваемый в разделе 5.6.5).
- Проверка вызова чего-либо просто из-за того, что вам сказали написать тест, а вы не уверены в том, что именно должно тестироваться. (Здесь будет уместно выяснить, правильно ли вы понимаете требования.)

5.6.4. Наличие более одного мока на тест

Считается хорошей практикой тестирувать только одно требование на тест. Тестирование более одного требования может привести к путанице и проблемам с сопровождением теста. Наличие двух моков на тест — то же самое, что тестируирование нескольких конечных результатов одной единицы работы (нескольких точек выхода).

Для каждой точки выхода рассмотрите возможность написания отдельного теста, так как она может считаться отдельным требованием. Скорее всего, при таком подходе имена ваших тестов также станут более точными и понятными. Если вы не можете выбрать имя для своего теста, потому что он делает слишком много всего и имя становится слишком общим (например, «XWorksOK»), стоит подумать о разбиении теста на несколько.

5.6.5. Излишняя детализация тестов

Если ваш тест имеет слишком много ожиданий (`x收到了().X()`, `x收到了().Y()` и т. д.), он может стать слишком хрупким, а его работоспособность будет нарушаться при малейших изменениях рабочего кода, даже если общая функциональность сохраняется. Тестирование взаимодействий — палка о двух концах: тестируете слишком много, и вы начинаете терять картину в целом — общую функциональность; тестируете слишком мало, и вы упускаете важные взаимодействия между единицами работы.

Есть разные способы найти баланс.

- *По возможности используйте стабы вместо моков* — если более 5 % ваших тестов используют моки, возможно, вы где-то переусердствовали. Стабы

могут присутствовать везде. О моках этого не скажешь. Тестируя необходиимо только один сценарий за раз. Чем больше у вас моков, тем больше верификаций будет выполняться в конце теста, но обычно важна только одна из них. Все остальное — шум относительно текущего сценария тестиирования.

- *По возможности избегайте использования стабов в качестве моков* — используйте стабы только для подстановки смоделированных значений в тестируемую единицу работы или для выдачи исключений. Не проверяйте, что методы были вызваны для стабов.

ИТОГИ

- Изолирующие фреймворки позволяют динамически создавать, настраивать и проверять моки и стабы в форме объекта или функции. Изолирующие фреймворки экономят много времени по сравнению с ручным написанием фейков, особенно в ситуациях с модульными зависимостями.
- Существуют две разновидности изолирующих фреймворков: со свободной типизацией (такие, как Jest и Sinon) и с сильной типизацией (такие, как `substitute.js`). Фреймворки со свободной типизацией требуют меньшего объема шаблонного кода и хорошо подходят для программирования в функциональном стиле; фреймворки с сильной типизацией хорошо подходят для работы с классами и интерфейсами.
- Изолирующие фреймворки могут заменять целые модули, но вместо этого лучше постараться абстрагировать прямые зависимости и создать фейки для этих абстракций. Это поможет сократить объем рефакторинга, необходимого при изменении API модуля.
- Там, где это возможно, следует отдавать предпочтение тестируанию возвращаемых значений или тестируанию на основе состояния (вместо тестиования взаимодействий), чтобы тесты ограничивались минимальными предположениями о подробностях внутренней реализации.
- Моки должны использоваться только в том случае, если другого способа тестиирования не существует, потому что в конечном счете они приводят к появлению тестов, которые будет трудно сопровождать.
- Выберите способ работы с изолирующими фреймворками на основании кодовой базы, над которой вы работаете. В унаследованных проектах иногда возникает необходимость в создании фейков для целых модулей, так как это может быть единственным способом добавления тестов в такие проекты. В проектах, строящихся с нуля, старайтесь добавлять подходящие абстракции поверх сторонних модулей. Все сводится кциальному выбору инструмента для работы, поэтому обязательно составьте общую картину при выборе подхода к конкретной проблеме при тестиировании.

Юнит-тестирование асинхронного кода

В ЭТОЙ ГЛАВЕ

- ✓ `async, done()` и `await`
- ✓ Уровни интеграционных и юнит-тестов для `async`
- ✓ Паттерн Extract Entry Point («выделение точки входа»)
- ✓ Паттерн Extract Adapter («выделение адаптера»)
- ✓ Создание стабов, увеличение счетчика таймера и его сброс

Если вы работаете с обычным синхронным кодом, ожидание того, когда завершатся действия, выполняется *неявно*. Вы не беспокоитесь об этом и даже особо не задумываетесь. Однако при работе с асинхронным кодом ожидание завершения действий становится *явно* выполняемой операцией, находящейся под вашим контролем. Асинхронность приводит к потенциальному усложнению кода и тестов для этого кода, потому что вам приходится явно выражать свои намерения относительно ожидания того, когда завершатся действия.

Чтобы продемонстрировать суть проблемы, начнем с простого примера загрузки данных.

6.1. АСИНХРОННАЯ ЗАГРУЗКА ДАННЫХ

Допустим, у вас имеется модуль, который проверяет работоспособность сайта example.com. Для этого он загружает контент по основному URL и проверяет наличие конкретного слова «illustrative», чтобы определить, работает ли сайт. Мы рассмотрим две разные и очень простые реализации этой функциональности. В первой используется механизм обратного вызова, а во второй — механизм `async/await`.

На рис. 6.1 изображены точки входа и выхода для целей нашего примера. Обратите внимание: направление стрелки обратного вызова изменено, чтобы более очевидно показать, что это другая разновидность точки выхода.

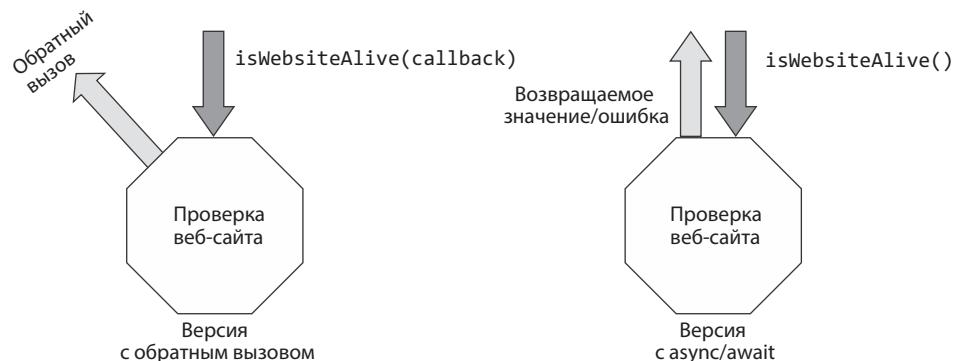


Рис. 6.1. Сравнение обратного вызова `isWebsiteAlive()` с версией `async/await`

Первоначальная версия кода приведена в листинге 6.1. Для загрузки контента по URL используется `node-fetch`.

Листинг 6.1. Версии с обратным вызовом `isWebsiteAlive()` и `await`

```
//Версия с обратным вызовом
const fetch = require("node-fetch");
const isWebsiteAliveWithCallback = (callback) => {
  const website = "http://example.com";
  fetch(website)
    .then((response) => {
      if (!response.ok) {
        //Как смоделировать проблему с сетью?
        throw Error(response.statusText); ←
      }
      return response;
    })
    .then((response) => response.text())
    .then((text) => {
      if (text.includes("illustrative")) {
        callback({ success: true, status: "ok" });
      } else {
        callback({ success: false, status: "error" });
      }
    })
  );
};

//Версия с await
const isWebsiteAlive = async () => {
  const website = "http://example.com";
  try {
    const response = await fetch(website);
    if (!response.ok) {
      throw Error(response.statusText);
    }
    const text = await response.text();
    if (text.includes("illustrative")) {
      return { success: true, status: "ok" };
    } else {
      return { success: false, status: "error" };
    }
  } catch (error) {
    return { success: false, status: "error" };
  }
};
```

```

    //Как протестировать этот путь?
    callback({ success: false, status: "text missing" });
}
})
.catch((err) => {
    //Как протестировать эту точку выхода?
    callback({ success: false, status: err });
});
};

// Версия с await
const isWebsiteAliveWithAsyncAwait = async () => {
    try {
        const resp = await fetch("http://example.com");
        if (!resp.ok) {
            //Как смоделировать ответ, отличный от OK?
            throw resp.statusText; ← Выдает специальную ошибку для
        }
        const text = await resp.text();
        const included = text.includes("illustrative");
        if (included) {
            return { success: true, status: "ok" };
        }
        // Как смоделировать отличающийся контент на сайте?
        throw "text missing";
    } catch (err) {
        return { success: false, status: err }; ← Упаковывает
    }
};

```

ПРИМЕЧАНИЕ В этом коде я предполагаю, что вы знаете, как работают промисы в JavaScript. Если вам потребуется больше информации, рекомендую ознакомиться с документацией Mozilla по адресу <http://mng.bz/W11a>.

В этом примере любые ошибки, связанные со сбоями соединений или отсутствием текста на веб-странице, преобразуются в обратный вызов или возвращаемое значение, которое служит признаком неудачи для пользователя нашей функции.

6.1.1. Первая попытка написания интеграционного теста

Так как в листинге 6.1 все жестко запрограммировано, как протестировать этот код? Возможно, вашей первой реакцией будет попытка написать интеграционный тест. Листинг 6.2 показывает, как можно было бы написать такой тест для версии с обратным вызовом.

Листинг 6.2. Первоначальная версия интеграционного теста

```
test("NETWORK REQUIRED (callback): correct content, true", (done) => {
    samples.isWebsiteAliveWithCallback((result) => {
        expect(result.success).toBe(true);
    });
});
```

```

    expect(result.status).toBe("ok");
    done();
});
});

```

Чтобы протестировать функцию, точкой выхода которой является функция обратного вызова, мы можем передать ей нашу собственную функцию обратного вызова, в которой можно:

- проверить правильность переданных значений;
- предписать программе для запуска тестов прекратить ожидание через любой механизм, предоставленный фреймворком тестирования (в данном случае функция `done()`).

6.1.2. Ожидание действия

Так как мы используем обратные вызовы как точки выхода, наш тест должен явно ожидать завершения параллельного выполнения. Это параллельное выполнение может происходить в цикле событий JavaScript, или в отдельном потоке, или даже в отдельном процессе, если вы используете другой язык.

В паттерне AAA часть действия требует ожидания. Во многих фреймворках тестирования для этой цели имеются специальные вспомогательные функции. В данном случае можно воспользоваться необязательным обратным вызовом `done`, который Jest предоставляет для передачи сигнала о том, что тест должен находиться в ожидании до явного вызова `done()`. Если `done()` не вызывается, то произойдет тайм-аут и по умолчанию тест завершится сбоем через 5 секунд (конечно, это время можно настроить).

В Jest также существуют другие средства для тестирования асинхронного кода, два из которых будут рассмотрены далее в этой главе.

6.1.3. Интеграционное тестирование `async/await`

Как насчет версии `async/await`? Формально можно написать тест, который почти не отличается от предыдущего, так как `async/await` — не более чем «синтаксический сахар» для работы с промисами.

Листинг 6.3. Интеграционный тест с обратными вызовами и `.then()`

```

test("NETWORK REQUIRED (await): correct content, true", (done) => {
  samples.isWebsiteAliveWithAsyncAwait().then((result) => {
    expect(result.success).toBe(true);
    expect(result.status).toBe("ok");
    done();
  });
});

```

Тем не менее тест с такими обратными вызовами, как `done()` и `then()`, читается намного хуже, чем тест, использующий паттерн AAA. К счастью, нет необходимости усложнять себе жизнь обязательным применением обратных вызовов. В тестах также можно использовать синтаксис `await`. Это заставляет вас ставить ключевое слово `async` перед тестовыми функциями, но, как показывает листинг 6.4, в целом ваши тесты станут проще и понятнее.

Листинг 6.4. Интеграционные тесты с `async/await`

```
test("NETWORK REQUIRED2 (await): correct content, true", async () => {
  const result = await samples.isWebsiteAliveWithAsyncAwait();
  expect(result.success).toBe(true);
  expect(result.status).toBe("ok");
});
```

Наличие асинхронного кода, позволяющего использовать синтаксис `async/await`, превращает наш тест *почти* в обыденный тест на основе значений. Точка входа также является точкой выхода, как было показано на рис. 6.1.

И хотя вызов был упрощен, он все еще остается асинхронным; это объясняет, почему я называю такой тест интеграционным. Какие проблемы может создавать эта разновидность тестов? Давайте разберемся.

6.1.4. Проблемы с интеграционными тестами

Тесты, написанные нами, не такие плохие, какими обычно бывают интеграционные тесты. Они получились относительно короткими и понятными, но при этом им присущи недостатки всех интеграционных тестов.

- *Большая продолжительность выполнения* — по сравнению с юнит-тестами интеграционные работают на порядок медленнее. Иногда их выполнение занимает секунды и даже минуты.
- *Ненадежность (неустойчивость)* — интеграционные тесты могут выдавать непоследовательные результаты (разный хронометраж в зависимости от того, где выполнялся тест, непоследовательные успешные или неуспешные результаты и т. д.).
- *Тестирование кода и условий окружения, которые могут быть несущественными* — интеграционные тесты проверяют разные части кода, которые могут не иметь отношения к тому, что вас интересует. (В нашем случае это библиотека `node-fetch`, условия сети, наличие брандмауэра, функциональность внешнего веб-сайта и т. д.).
- *Длительный анализ* — когда интеграционный тест не проходит, его анализ и отладка требуют больше времени, потому что у сбоя может быть много причин.

- *Усложнение моделирования* — для интеграционных тестов сложнее смоделировать негативный тест (моделирование неверного контента на веб-сайте, неработоспособности веб-сайта, сетевых сбоев и т. д.).
- *Меньшая достоверность результатов* — вы можете решить, что неудача интеграционного теста обусловлена внешней проблемой, тогда как в действительности она возникает из-за ошибки в коде. Достоверность будет более подробно рассмотрена в следующей главе.

Означает ли все это, что вам не стоит писать интеграционные тесты? Нет, я полагаю, что интеграционные тесты абсолютно необходимы, но не нужно создавать их в большом количестве, чтобы иметь достаточную уверенность в коде. То, что не покрывают интеграционные тесты, должно покрываться низкоуровневыми тестами — юнит-тестами, тестами компонентов или API. Эта стратегия будет подробно рассмотрена в главе 10, посвященной стратегиям тестирования.

6.2. СОЗДАНИЕ КОДА, УДОБНОГО ДЛЯ ЮНИТ-ТЕСТОВ

Как протестировать код при помощи юнит-теста? Продемонстрирую несколько паттернов, которые я использую для того, чтобы улучшить тестируемость кода (то есть чтобы упростить внедрение или предотвращение зависимостей или проверить точки выхода).

- *Паттерн Extract Entry Point* (*«выделение точки входа»*) — выделение частей рабочего кода, содержащих чистую логику, в отдельные функции и рассмотрение этих функций как точек входа наших тестов.
- *Паттерн Extract Adapter* (*«выделение адаптера»*) — выделение сущности, асинхронной по своей природе, и ее абстрагирование для замены чем-то синхронным.

6.2.1. Выделение точки входа

В этом паттерне мы возьмем конкретную единицу асинхронной работы и разобьем ее на две части:

- асинхронная часть (которая остается без изменений);
- обратные вызовы, активируемые при завершении асинхронного выполнения. Они выделяются в новые функции; они в итоге становятся точками входа для чисто логической единицы работы, которая может активироваться чистыми юнит-тестами.

Идея представлена на рис. 6.2: в части «До» изображена одна единица работы, содержащая асинхронный код, смешанный с логикой, которая обрабатывает

асинхронные результаты на внутреннем уровне и возвращает результат через обратный вызов или механизм промисов. На шаге 1 логика выделяется в отдельную функцию (или функции), содержащую только результаты асинхронной работы как входные данные. На шаге 2 эти функции выносятся наружу, чтобы они могли использоваться как точки входа наших юнит-тестов.

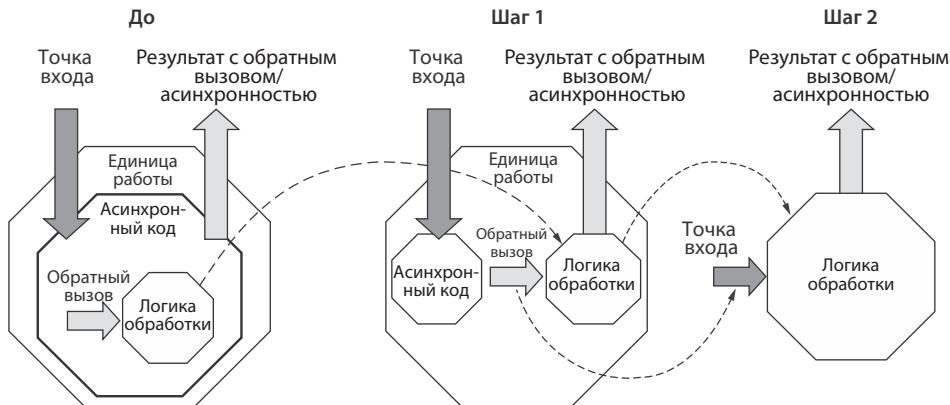


Рис. 6.2. Выделение внутренней логики обработки в отдельную единицу работы способствует упрощению тестов, потому что новую единицу работы можно проверить синхронно и без привлечения внешних зависимостей

Такая схема предоставляет нам важную возможность тестирования логической обработки асинхронных обратных вызовов (и простого моделирования ввода). В то же время мы можем написать высокоуровневый интеграционный тест для исходной единицы работы, чтобы иметь уверенность в том, что асинхронная координация тоже работает правильно.

Если выполнить интеграционные тесты для всех наших сценариев, вы окажетесь один на один со множеством подолгу выполняемых и ненадежных тестов. Нам хотелось бы, чтобы наши тесты большей частью были быстрыми и стабильными, а поверх них работала небольшая прослойка интеграционных тестов, которая обеспечит работу всей промежуточной координации. В таком случае нам не придется жертвовать скоростью и удобством сопровождения ради достоверности.

Пример выделения единицы работы

Применим этот паттерн к коду из листинга 6.1. На рис. 6.3 представлены действия, которые необходимо выполнить.

- ❶ Состояние «До» содержит логику обработки, интегрированную в функцию `isWebsiteAlive()`.

- ❷ Весь логический код, применяемый к результатам загрузки, выделяется и размещается в двух разных функциях: одна обрабатывает случай успеха, а другая — случай ошибки.
- ❸ Эти две функции выносятся наружу, чтобы их можно было напрямую вызывать из юнит-тестов.

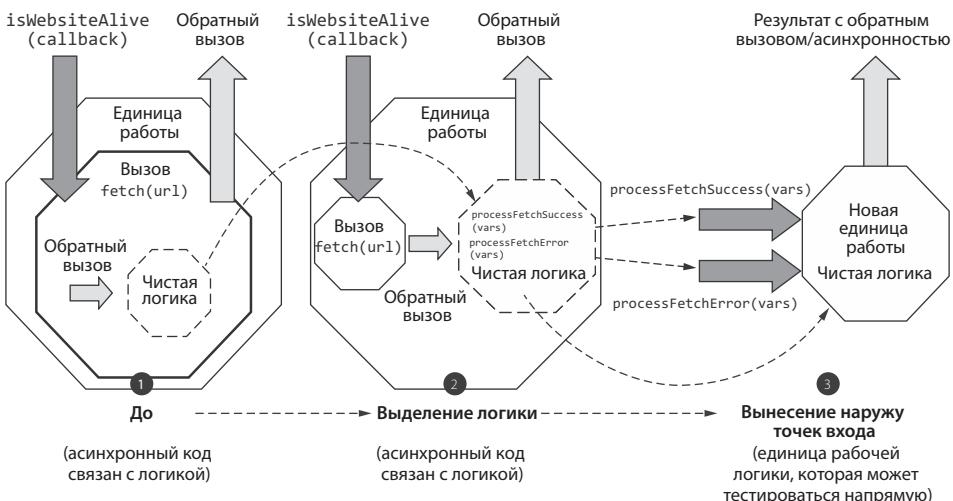


Рис. 6.3. Выделение логики обработки успеха и ошибки из isWebSiteAlive() для раздельного тестирования этой логики

Код, полученный в результате рефакторинга, приведен в листинге 6.5.

Листинг 6.5. Выделение точек входа с callback

```
//Точка входа
const isWebsiteAlive = (callback) => {
  fetch("http://example.com")
    .then(throwOnInvalidResponse)
    .then((resp) => resp.text())
    .then((text) => {
      processFetchSuccess(text, callback);
    })
    .catch((err) => {
      processFetchError(err, callback);
    });
};

const throwOnInvalidResponse = (resp) => {
  if (!resp.ok) {
    throw Error(resp.statusText);
  }
  return resp;
};
```

```
//Точка входа
const processFetchSuccess = (text, callback) => {
  if (text.includes("illustrative")) {
    callback({ success: true, status: "ok" });
  } else {
    callback({ success: false, status: "missing text" });
  }
};

//Точка входа
const processFetchError = (err, callback) => {
  callback({ success: false, status: err });
};
```

Новые точки входа (единицы работы)

Как видно из листинга, исходная единица работы, с которой мы начали, теперь содержит три точки входа вместо одной. Новые точки входа могут использоваться для юнит-тестирования, тогда как исходная все еще может использоваться для интеграционного тестирования, как показано на рис. 6.4.

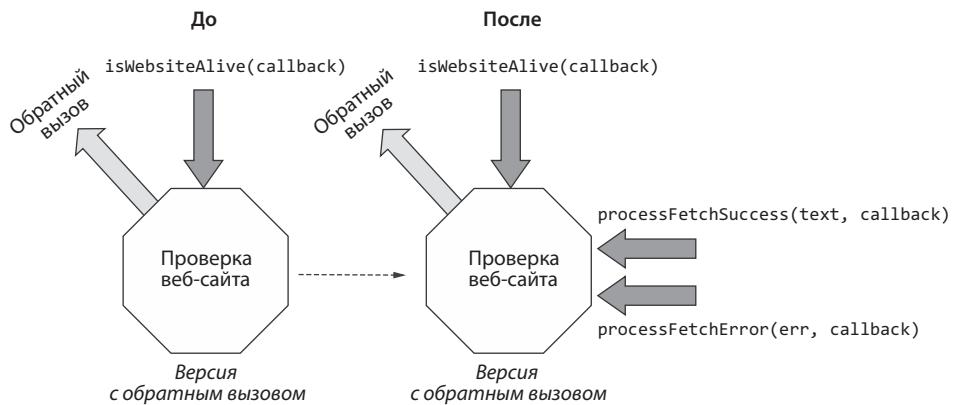


Рис. 6.4. Новые точки входа, добавленные после выделения двух новых функций.

Новые функции теперь могут тестируться простыми юнит-тестами вместо интеграционных тестов, которые были необходимы до рефакторинга

Для исходной точки входа все еще нужны интеграционные тесты, но не более чем один-два. Любые другие сценарии могут моделироваться с использованием чисто логических точек входа, быстро и безболезненно.

Теперь можно написать юнит-тесты, активирующие новые точки входа, как показано в листинге 6.6.

Листинг 6.6. Юнит-тесты с выделенными точками входа

```

describe("Website alive checking", () => {
  test("content matches, returns true", (done) => {
    samples.processFetchSuccess("illustrative", (err, result) => { ←
      expect(err).toBeNull();
      expect(result.success).toBe(true);
      expect(result.status).toBe("ok");
      done();
    });
  });
  test("website content does not match, returns false", (done) => {
    samples.processFetchSuccess("bad content", (err, result) => { ←
      expect(err.message).toBe("missing text");
      done();
    });
  });
  test("When fetch fails, returns false", (done) => {
    samples.processFetchError("error text", (err, result) => { ←
      expect(err.message).toBe("error text");
      done();
    });
  });
});

```

Активирует новые точки входа

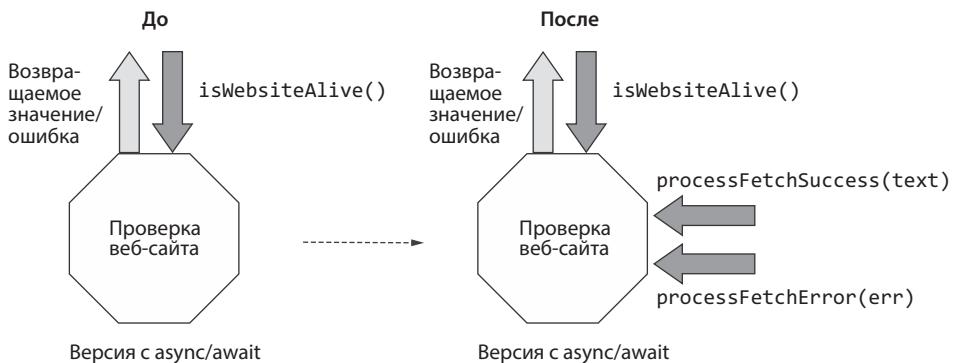
Обратите внимание на то, что мы активируем новую точку входа напрямую, и можем легко моделировать разные условия. В этих тестах нет ничего асинхронного, но нам все еще нужна функция `done()`, потому что обратные вызовы могут вообще не активироваться и необходимо иметь возможность обнаруживать эту ситуацию.

Нам все еще нужен хотя бы один интеграционный тест, который даст уверенность в том, что асинхронная координация работает между точками входа. Здесь исходный интеграционный тест может помочь, но более не нужно писать все сценарии тестов как интеграционные (подробнее об этом в главе 10).

Выделение точки входа с `await`

Тот же паттерн, который мы только что применили, может хорошо работать для стандартных функциональных структур `async/await`. Этот рефакторинг изображен на рис. 6.5.

Представляя синтаксис `async/await`, мы можем вернуться к написанию линейного кода без использования аргументов обратных вызовов. Функция `isWebsiteAlive()` начинает выглядеть почти так же, как обычный синхронный код, который возвращает значения и выдает ошибки там, где это необходимо.

**Рис. 6.5.** Выделение точек входа с async/await

В листинге 6.7 показано, как это выглядит в рабочем коде.

Листинг 6.7. Функция, написанная с async/await вместо обратных вызовов

```
//Точка входа
const isWebsiteAlive = async () => {
  try {
    const resp = await fetch("http://example.com");
    throwIfResponseNotOK(resp);
    const text = await resp.text();
    return processFetchContent(text);
  } catch (err) {
    return processFetchError(err);
  }
};

const throwIfResponseNotOK = (resp) => {
  if (!resp.ok) {
    throw resp.statusText;
  }
};

//Точка входа
const processFetchContent = (text) => {
  const included = text.includes("illustrative");
  if (included) {
    return { success: true, status: "ok" };
  }
  return { success: false, status: "missing text" };
};

//Точка входа
const processFetchError = (err) => {
  return { success: false, status: err };
};
```

В правой части листинга 6.7 приведены комментарии, объясняющие особенности использования async/await:

- Комментарий к `processFetchContent`: "Возвращает значение вместо активации обратного вызова"

В отличие от примеров с обратными вызовами, мы используем `return` или `throw` для обозначения успехов или неудач. Это распространенный паттерн написания кода с `async/await`.

Тесты тоже упрощаются, как показано в листинге 6.8.

Листинг 6.8. Тестирование точек входа, выделенных с `async/await`

```
describe("website up check", () => {
  test("on fetch success with good content, returns true", () => {
    const result = samples.processFetchContent("illustrative");
    expect(result.success).toBe(true);
    expect(result.status).toBe("ok");
  });

  test("on fetch success with bad content, returns false", () => {
    const result = samples.processFetchContent("text not on site");
    expect(result.success).toBe(false);
    expect(result.status).toBe("missing text");
  });

  test("on fetch fail, throws ", () => {
    expect(() => samples.processFetchError("error text"))
      .toThrowError("error text");
  });
});
```

И снова обратите внимание на то, что нам не нужно добавлять какие-либо ключевые слова, относящиеся к `async/await`, или явно указывать на необходимость ожидания выполнения, потому что логическая единица работы отделена от асинхронных частей, которые усложняют нашу задачу.

6.2.2. Паттерн «выделение адаптера»

Паттерн «выделение адаптера» использует подход, противоположный предыдущему. Мы рассматриваем асинхронный блок кода точно так же, как любую зависимость, упоминаемую в предыдущих главах, — как нечто такое, что нам хотелось бы заменить в своих тестах для повышения уровня управления. Вместо того чтобы выделять логический код в собственный набор точек входа, мы извлекаем асинхронный код (нашу *зависимость*) и абстрагируем ее в адаптере, который позднее может внедряться как любая другая зависимость. На рис. 6.6 показано, как это происходит.

Также часто используется создание специального интерфейса адаптера, упрощенного для нужд потребителя этой зависимости. Другое название такого метода — *принцип разделения интерфейсов*. В данном случае мы создаем модуль `network-adapter` (адаптер сети), который скрывает реальную функциональность загрузки и имеет собственные функции, как показано на рис. 6.7.

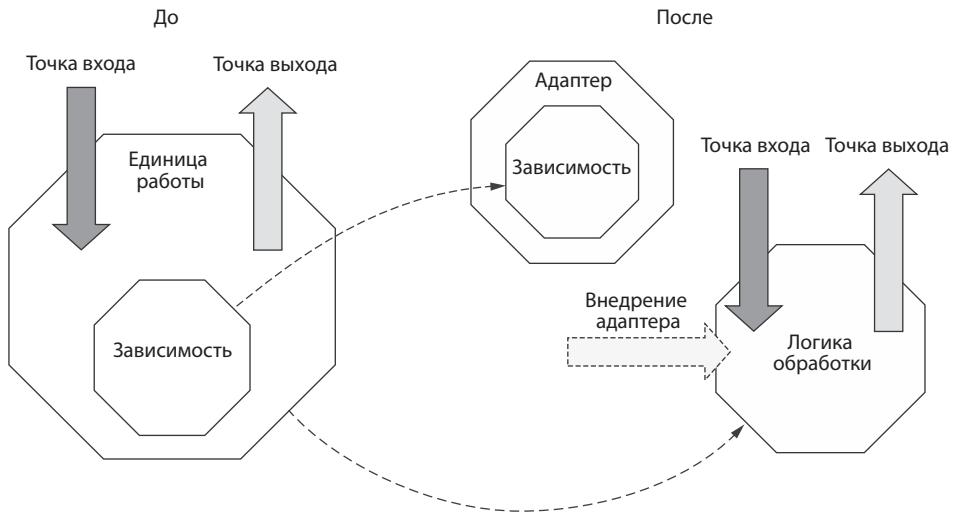


Рис. 6.6. Выделение зависимости и упаковка ее в адаптер позволяют упростить эту зависимость и заменить ее фейком в тестах

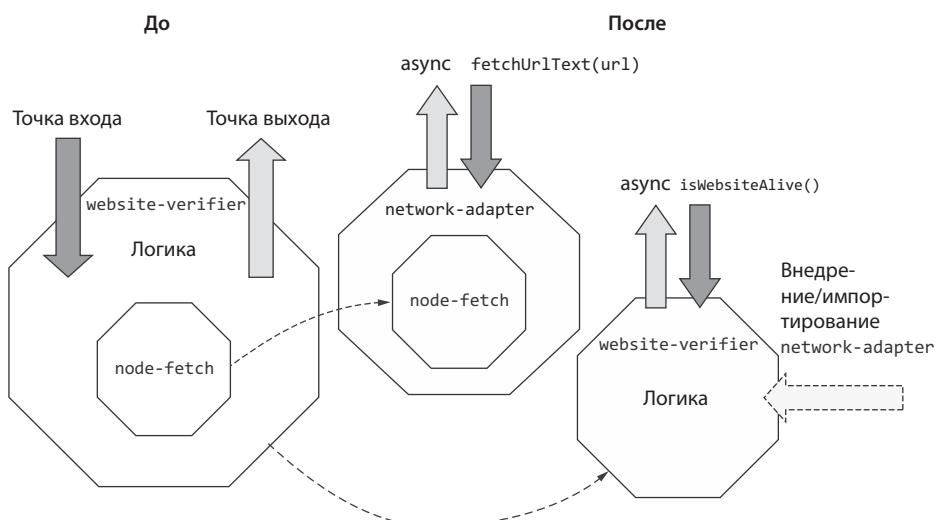


Рис. 6.7. Упаковка модуля node-fetch в собственный модуль network-adapter помогает нам предоставить только ту функциональность, которая нужна нашему приложению, выраженную на языке, наиболее подходящем для имеющейся задачи

Принцип разделения интерфейсов

Термин *принцип разделения интерфейсов* был предложен Робертом Мартином. Представьте зависимость базы данных с десятками функций, скрытых за адаптером, интерфейс которого может содержать только пару функций со специально выбранными именами и параметрами. Задача адаптера — скрыть сложность и упростить как код потребителя, так и тесты, в которых он моделируется. За дополнительной информацией о разделении интерфейсов обращайтесь к статье в Википедии по адресу https://en.wikipedia.org/wiki/Interface_segregation_principle.

В листинге 6.9 показано, как может выглядеть модуль `network-adapter`.

Листинг 6.9. Код `network-adapter`

```
const fetch = require("node-fetch");

const fetchUrlText = async (url) => {
  const resp = await fetch(url);
  if (resp.ok) {
    const text = await resp.text();
    return { ok: true, text: text };
  }
  return { ok: false, text: resp.statusText };
};
```

Обратите внимание: модуль `network-adapter` — единственный модуль в проекте, импортирующий `node-fetch`. Если эта зависимость когда-нибудь в будущем изменится, вполне возможно, что достаточно будет изменить только один этот файл. Мы также упростили функцию как по имени, так и по функциональности. При этом скрыта необходимость загружать статус и текст по URL, и мы абстрагируем их в одной удобной функции.

Теперь нужно решить, как использовать адаптер. Для начала его можно использовать в модульном стиле. Затем мы применим функциональный и объектно-ориентированный подход с сильно типизированным интерфейсом.

Модульный адаптер

В листинге 6.10 продемонстрировано модульное использование `network-adapter` в исходной версии функции `isWebsiteAlive()`.

Листинг 6.10. `isWebsiteAlive()` с использованием модуля `network-adapter`

```
const network = require("./network-adapter");
```

```
const isWebsiteAlive = async () => {
  try {
    const result = await network.fetchUrlText("http://example.com");
    if (!result.ok) {
      throw result.text;
    }
    const text = result.text;
    return processFetchSuccess(text);
  } catch (err) {
    throw processFetchFail(err);
  }
};
```

В этой версии мы напрямую импортируем модуль `network-adapter`, для которого позднее в наших тестах будет создан фейк.

Юнит-тесты для этого модуля приведены в листинге 6.11. Так как мы используем модульную структуру, то можем создать фейк для модуля с помощью `jest.mock()`. Не беспокойтесь, модуль также будет внедрен в дальнейших примерах.

Листинг 6.11. Создание фейка для `network-adapter` с `jest.mock`

```
jest.mock("./network-adapter");           ← Сохраняет переданное значение в этой переменной
const stubSyncNetwork = require("./network-adapter");   ← Импортирует фейковый
const webverifier = require("./website-verifier");       модуль

describe("unit test website verifier", () => {
  beforeEach(jest.resetAllMocks);           ← Сбрасывает все стэбы, чтобы избежать
                                            потенциальных проблем в других тестах

  test("with good content, returns true", async () => {
    stubSyncNetwork.fetchUrlText.mockReturnValue({        ← Моделирует возвращаемое
      ok: true,                                         значение от стэба модуля
      text: "illustrative",
    });
    const result = await webverifier.isWebsiteAlive();
    expect(result.success).toBe(true);
    expect(result.status).toBe("ok");
  });

  test("with bad content, returns false", async () => {
    stubSyncNetwork.fetchUrlText.mockReturnValue({
      ok: true,
      text: "<span>hello world</span>",
    });
    const result = await webverifier.isWebsiteAlive();
    expect(result.success).toBe(false);
    expect(result.status).toBe("missing text");
  });
});
```

Использование
`await` в тестах

Обратите внимание: мы снова применяем `async/await`, потому что вернулись к использованию исходной точки входа, которая была описана в начале этой главы. Но сам факт применения `await` не означает, что тесты выполняются

асинхронно. Код теста и рабочий код, который он вызывает, фактически выполняются линейно с асинхронно ориентированной сигнатурой. Использовать `async/await` для функциональных и объектно-ориентированных структур необходимо, потому что этого требует точка входа.

Я назвал свою фейковую сеть `stubSyncNetwork`, чтобы более ясно выразить синхронную природу теста. В противном случае при взгляде на тест будет трудно сказать, выполняется код линейно или асинхронно.

Функциональный адаптер

В функциональном паттерне проектирования структура модуля `network-adapter` остается неизменной, но его внедрение в `website-verifier` происходит иначе. Как показано в листинге 6.12, в точку входа добавляется новый параметр.

Листинг 6.12. Функциональный паттерн внедрения для `isWebsiteAlive()`

```
const isWebsiteAlive = async (network) => {
  const result = await network.fetchUrlText("http://example.com");
  if (result.ok) {
    const text = result.text;
    return onFetchSuccess(text);
  }
  return onFetchError(result.text);
};
```

В этой версии мы ожидаем, что модуль `network-adapter` внедряется через общий параметр нашей функции. В функциональном стиле можно использовать функции высшего уровня и каррирование для настройки предварительно внедренной функции с нашей сетевой зависимостью. В наших тестах можно просто передать фейковую сеть через этот параметр. В том, что касается структуры внедрения, почти ничего не изменилось по сравнению с предыдущими примерами, не считая того факта, что мы более нигде не импортируем модуль `network-adapter`. Сокращение объема `import` и `require` способствует достижению простоты в сопровождении в долгосрочной перспективе.

В листинге 6.13 приведены упрощенные версии наших тестов с меньшим объемом шаблонного кода.

Листинг 6.13. Юнит-тест с функциональным внедрением `network-adapter`

```
const webverifier = require("./website-verifier");

const makeStubNetworkWithResult = (fakeResult) => { ←
  return {
    fetchUrlText: () => {
      return fakeResult;
    },
  };
};
```

Новая вспомогательная функция для создания нестандартного объекта, который соответствует важным частям интерфейса `network-adapter`

```

describe("unit test website verifier", () => {
  test("with good content, returns true", async () => {
    const stubSyncNetwork = makeStubNetworkWithResult({
      ok: true,
      text: "illustrative",
    });
    const result = await webverifier.isWebsiteAlive(stubSyncNetwork);
    expect(result.success).toBe(true);
    expect(result.status).toBe("ok");
  });
}

test("with bad content, returns false", async () => {
  const stubSyncNetwork = makeStubNetworkWithResult({
    ok: true,
    text: "unexpected content",
  });
  const result = await webverifier.isWebsiteAlive(stubSyncNetwork); ←
  expect(result.success).toBe(false);
  expect(result.status).toBe("missing text");
});
...

```

Внедрение
нестандартного
объекта

Внедрение
нестандартного
объекта

Заметим, что нам не приходится включать большой объем шаблонного кода в начало файла, как это было в модульной схеме. Не нужно создавать фейк для модуля косвенно (через `jest.mock`), повторно импортировать его для наших тестов (посредством `require`) и сбрасывать состояние Jest с использованием `jest.resetAllMocks`. Все, что нужно сделать, — это вызвать новую вспомогательную функцию `makeStubNetworkWithResult` из каждого теста, чтобы сгенерировать новый фейковый адаптер сети, а затем внедрить фейковую сеть, передавая ее в параметре точки входа.

Объектно-ориентированный адаптер на основе интерфейса

Мы рассмотрели модульные и функциональные схемы. Теперь обратимся к объектно-ориентированной. В объектно-ориентированной парадигме можно взять внедрение параметра, которое выполнялось ранее, и преобразовать его в паттерн внедрения через конструктор. Начнем с адаптера сети и его интерфейсов (сигнатура открытого API и результатов) в листинге 6.14.

Листинг 6.14. NetworkAdapter и его интерфейсы

```

export interface INetworkAdapter {
  fetchUrlText(url: string): Promise<NetworkAdapterFetchResults>;
}
export interface NetworkAdapterFetchResults {
  ok: boolean;
  text: string;
}

```

`ch6-async/6-fetch-adapter-interface-oo/network-adapter.ts`

```
export class NetworkAdapter implements INetworkAdapter {
    async fetchUrlText(url: string):
        Promise<NetworkAdapterFetchResults> {
        const resp = await fetch(url);
        if (resp.ok) {
            const text = await resp.text();
            return Promise.resolve({ ok: true, text: text });
        }
        return Promise.reject({ ok: false, text: resp.statusText });
    }
}
```

В листинге 6.15 создается класс `WebsiteVerifier` с конструктором, получающим параметр `INetworkAdapter`.

Листинг 6.15. Класс WebsiteVerifier с внедрением через конструктор

```
export interface WebsiteAliveResult {
    success: boolean;
    status: string;
}

export class WebsiteVerifier {
    constructor(private network: INetworkAdapter) {}

    isWebsiteAlive = async (): Promise<WebsiteAliveResult> => {
        let netResult: NetworkAdapterFetchResults;
        try {
            netResult = await this.network.fetchUrlText("http://example.com");
            if (!netResult.ok) {
                throw netResult.statusText;
            }
            const text = netResult.text;
            return this.processNetSuccess(text);
        } catch (err) {
            throw this.processNetFail(err);
        }
    };

    processNetSuccess = (text): WebsiteAliveResult => {
        const included = text.includes("illustrative");
        if (included) {
            return { success: true, status: "ok" };
        }
        return { success: false, status: "missing text" };
    };

    processNetFail = (err): WebsiteAliveResult => {
        return { success: false, status: err };
    };
}
```

Юнит-тесты для этого класса могут создать экземпляр фейкового адаптера сети и внедрить его через конструктор. В листинге 6.16 `substitute.js` используется для создания фейкового объекта, соответствующего новому интерфейсу.

Листинг 6.16. Юнит-тесты для объектно-ориентированного WebsiteVerifier

```
const makeStubNetworkWithResult = (fakeResult: NetworkAdapterFetchResults) => {  
    const stubNetwork = Substitute.for<INetworkAdapter>();  
    stubNetwork.fetchUrlText(Arg.any())  
        .returns(Promise.resolve(fakeResult));  
    return stubNetwork;  
};  
describe("unit test website verifier", () => {  
    test("with good content, returns true", async () => {  
        const stubSyncNetwork = makeStubNetworkWithResult({  
            ok: true,  
            text: "illustrative",  
        });  
        const webVerifier = new WebsiteVerifier(stubSyncNetwork);  
  
        const result = await webVerifier.isWebsiteAlive();  
        expect(result.success).toBe(true);  
        expect(result.status).toBe("ok");  
    });  
  
    test("with bad content, returns false", async () => {  
        const stubSyncNetwork = makeStubNetworkWithResult({  
            ok: true,  
            text: "unexpected content",  
        });  
        const webVerifier = new WebsiteVerifier(stubSyncNetwork);  
  
        const result = await webVerifier.isWebsiteAlive();  
        expect(result.success).toBe(false);  
        expect(result.status).toBe("missing text");  
    });  
});
```

Эта разновидность инверсии контроля IoC (Inversion of Control) и внедрения зависимостей DI (Dependency Injection) работает вполне эффективно. В объектно-ориентированном мире внедрение через конструктор с интерфейсами используется очень часто и во многих случаях может предоставить обоснованное и простое в сопровождении решение для отделения зависимостей от логики.

6.3. ТАЙМЕРЫ

Таймеры (например, `setTimeout`) представляют собой проблему, специфическую для JavaScript. Они являются частью предметной области и нередко используются в коде, нравится нам это или нет. Вместо того чтобы выделять адаптеры и точки входа, иногда проще отключить эти функции и поискать обходное решение. Мы рассмотрим два способа:

- прямое исправление ошибок во время исполнения программы (monkey-patching) в функции;
- использование Jest и других фреймворков для отключения этих функций и контроля над ними.

6.3.1. Создание стабов для таймеров

Исправление ошибок во время исполнения программы, или манки-патчинг, — механизм, позволяющий программам локально расширять или изменять системное ПО (так, чтобы изменения затронули только работающий экземпляр программы). Такие языки программирования и исполнительные среды, как JavaScript, Ruby и Python, позволяют реализовать манки-патчинг достаточно просто. У компилируемых языков с сильной типизацией (таких, как C# и Java) это делается намного сложнее. Манки-патчинг более подробно рассматривается в приложении к книге.

Приведу один из возможных способов для JavaScript. Начнем со следующего фрагмента кода, в котором используется метод `setTimeout`.

Листинг 6.17. Код с функцией `setTimeout`, которую мы хотим исправить во время исполнения программы

```
const calculate1 = (x, y, resultCallback) => {
  setTimeout(() => { resultCallback(x + y); },
  5000);
};
```

Функцию `setTimeout` можно изменить во время исполнения так, чтобы она стала синхронной, буквально назначая прототип этой функции в памяти, как показано в листинге 6.18.

Листинг 6.18. Простой паттерн манки-патчинга

```
const Samples = require("./timing-samples");

describe("monkey patching ", () => {
  let originalTimeout;
  beforeEach(() => (originalTimeout = setTimeout));
  afterEach(() => (setTimeout = originalTimeout));
  test("calculate1", () => {
    setTimeout = (callback, ms) => callback();
    Samples.calculate1(1, 2, (result) => {
      expect(result).toBe(3);
    });
  });
});
```

Так как все происходит синхронно, нам не нужно использовать `done()` для ожидания активизации обратного вызова. `setTimeout` заменяется чисто синхронной реализацией, которая активирует полученный обратный вызов немедленно.

Единственный недостаток такого решения заключается в том, что он требует шаблонного кода и обычно повышает риск ошибок, так как вы должны помнить о необходимости выполнить правильные завершающие действия. Посмотрим, какие фреймворки предоставляет Jest для таких ситуаций.

6.3.2. Фейки `setTimeout` средствами Jest

Jest предоставляет три функции для работы с основными типами таймеров в JavaScript:

- `jest.useFakeTimers` — создает фейки для всех функций таймеров (таких, как `setTimeout`);
- `jest.resetAllTimers` — сбрасывает все фейковые таймеры и восстанавливает настоящие;
- `jest.advanceTimersToNextTimer` — вызывает срабатывание любого фейкового таймера, чтобы сработали обратные вызовы.

В совокупности эти функции вместо вас заботятся о большей части шаблонного кода.

В листинге 6.19 приведен тот же тест, что и в листинге 6.18, но на этот раз с использованием вспомогательных функций Jest.

Листинг 6.19. Создание фейка для `setTimeout` средствами Jest

```
describe("calculate1 - with jest", () => {
  beforeEach(jest.clearAllTimers);
  beforeEach(jest.useFakeTimers);
```

```
test("fake timeout with callback", () => {
  Samples.calculate1(1, 2, (result) => {
    expect(result).toBe(3);
  });
  jest.advanceTimersToNextTimer();
});
});
```

Обратите внимание: здесь также не нужно вызывать `done()`, потому что все происходит синхронно. В то же время необходимо использовать `advanceTimersToNextTimer`, потому что без него фейковая версия `setTimeout` останется навсегда. `advanceTimersToNextTimer` также помогает в ситуациях, когда тестируемый модуль планирует вызов `setTimeout`, обратный вызов которого рекурсивно планирует другой вызов `setTimeout` (а это означает, что планирование никогда не прервется). В таких сценариях полезно иметь возможность перемещаться вперед во времени, шаг за шагом.

С `advanceTimersToNextTimer` можно переместить все таймеры на заданное количество шагов, чтобы смоделировать прохождение шагов, которые приведут к срабатыванию следующего обратного вызова таймера, ожидающего в очереди.

Тот же паттерн хорошо работает с `setInterval`, как показано в листинге 6.20.

Листинг 6.20. Функция, использующая setInterval

```
const calculate4 = (getInputsFn, resultFn) => {
  setInterval(() => {
    const { x, y } = getInputsFn();
    resultFn(x + y);
  }, 1000);
};
```

В данном случае функция получает в параметрах два обратных вызова: один предоставляет входные данные для вычислений, а другой передает их результат. Вызов `setInterval` используется для непрерывного получения новых входных данных и вычисления их результатов.

В листинге 6.21 приведен тест, который увеличивает счетчик таймера, дважды назначает интервал и ожидает, что результаты обоих вызовов совпадут.

Листинг 6.21. Увеличение счетчика фейковых таймеров в юнит-тесте

```

Samples.calculate4(inputFn, (result) => results.push(result));

jest.advanceTimersToNextTimer();
jest.advanceTimersToNextTimer(); | setInterval
                                | вызывается дважды

expect(results[0]).toBe(3);
expect(results[1]).toBe(5);
});

});

```

В этом примере мы убеждаемся в том, что новые значения правильно вычисляются и сохраняются. Заметим, что тот же тест можно было бы написать только с одним вызовом и одним `expect` примерно с такой же степенью достоверности, как у более сложного теста, но я предпочитаю добавить дополнительную проверку в тех случаях, когда мне требуется более высокая степень достоверности.

6.4. РАСПРОСТРАНЕННЫЕ СОБЫТИЯ

Говоря об асинхронном юнит-тестировании, невозможно не упомянуть простейший поток событий. Надеюсь, тема асинхронного юнит-тестирования к настоящему моменту уже не кажется вам настолько сложной, но мне хочется рассмотреть события отдельно от всего остального.

6.4.1. Генераторы событий

Чтобы избежать любого возможного недопонимания, приведу четкое и лаконичное определение генератора событий (event emitter) из учебника DigitalOcean «Using Event Emitters in Node.js».

Генераторы событий — это объекты в Node.js, которые выдают события, отправляя сообщение, сигнализирующее о завершении действия. Разработчики JavaScript могут написать код, который прослушивает события от генератора событий, что позволяет им выполнять функции при каждом срабатывании этих событий. В этом контексте события состоят из идентифицирующей строки и любых данных, которые должны передаваться слушателям.

Рассмотрим класс `Adder` из листинга 6.22, который выдает событие при каждом суммировании.

Листинг 6.22. Простой класс Adder с генератором событий

```

const EventEmitter = require("events");

class Adder extends EventEmitter {
  constructor() {

```

```

        super();
    }

    add(x, y) {
        const result = x + y;
        this.emit("added", result);
        return result;
    }
}

```

Самый простой способ написать юнит-тест, который проверяет, что событие было выдано, — в прямом смысле подписать на событие в тесте и убедиться в том, что оно срабатывает при вызове функции `add`.

Листинг 6.23. Тестирование генераторов событий путем подписки на них

```

describe("events based module", () => {
    describe("add", () => {
        it("generates addition event when called", (done) => {
            const adder = new Adder();
            adder.on("added", (result) => {
                expect(result).toBe(3);
                done();
            });
            adder.add(1, 2);
        });
    });
});

```

При помощи `done()` мы убеждаемся в том, что событие действительно сработало. Если бы мы не использовали `done()`, а событие не было бы сгенерировано, то наш тест прошел бы, потому что код, зарегистрированный при подписке, не выполнился. Добавляя `expect(x).toBe(y)`, мы также проверяем значения, переданные в параметрах события, наряду с неявным тестированием того, что событие сработало.

6.4.2. События click

Как насчет этих раздражающих UI-событий — например, `click`? Как протестировать правильность их связывания в наших скриптах? Возьмем простую веб-страницу и ассоциированную с ней логику из листингов 6.24 и 6.25.

Листинг 6.24. Простая веб-страница с функциональностью JavaScript click

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>File to Be Tested</title>
    <script src="index-helper.js"></script>

```

```
</head>
<body>
  <div>
    <div>A simple button</div>
    <Button data-testid="myButton" id="myButton">Click Me</Button>
    <div data-testid="myResult" id="myResult">Waiting...</div>
  </div>
</body>
</html>
```

Листинг 6.25. Логика веб-страницы в JavaScript

```
window.addEventListener("load", () => {
  document
    .getElementById("myButton")
    .addEventListener("click", onMyButtonClick);

  const resultDiv = document.getElementById("myResult");
  resultDiv.innerText = "Document Loaded";
});

function onMyButtonClick() {
  const resultDiv = document.getElementById("myResult");
  resultDiv.innerText = "Clicked!";
}
```

Эта очень простая логика проверяет, что щелчок по кнопке записывает специальное сообщение. Как это можно протестировать?

Начнем с антипаттерна: можно подписаться на событие `click` в своих тестах и убедиться в том, что оно сработало... только пользы от этого будет немного. Нас интересует не то, сработало ли событие, а сделало ли оно что-то полезное.

Более эффективное решение: можно инициировать событие `click` и убедиться в том, что оно изменило нужное значение внутри страницы, — этот результат будет приносить реальную пользу. На рис. 6.8 показано, как это делается.

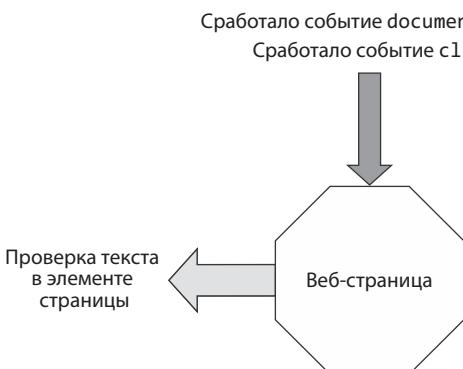


Рис. 6.8. Click как точка входа, элемент как точка выхода

В листинге 6.26 показано, как может выглядеть тест.

Листинг 6.26. Инициирование события click и проверка текста элемента

```
/**  
 * @jest-environment jsdom      ← Применение среды jsdom, имитирующей  
 */  
 //((Приведенное выше необходимо для оконных событий)  
 const fs = require("fs");  
 const path = require("path");  
 require("./index-helper.js");  
  
 const loadHtml = (fileRelativePath) => {  
   const filePath = path.join(__dirname, "index.html");  
   const innerHTML = fs.readFileSync(filePath);  
   document.documentElement.innerHTML = innerHTML;  
 };  
  
 const loadHtmlAndGetUIElements = () => {  
   loadHtml("index.html");  
   const button = document.getElementById("myButton");  
   const resultDiv = document.getElementById("myResult");  
   return { window, button, resultDiv };  
 };  
  
 describe("index helper", () => {  
   test("vanilla button click triggers change in result div", () => {  
     const { window, button, resultDiv } = loadHtmlAndGetUIElements();  
     window.dispatchEvent(new Event("load")); ← Имитация события document.load  
  
     button.click(); ← Инициирует click  
  
     expect(resultDiv.innerText).toBe("Clicked!"); ← Проверяет, что элемент  
   });  
});
```

В этом примере я выделил два вспомогательных метода, `loadHtml` и `loadHtmlAndGetUIElements`, чтобы тесты были более чистыми и читабельными, — а следовательно, в будущем у меня будет меньше хлопот с изменением тестов при изменении позиции или идентификатора UI-элемента.

В самом тесте имитируется событие `document.load`, чтобы тестируемый скрипт мог начать выполнение, а затем инициировать событие, будто пользователь щелкнул по кнопке. Наконец, тест проверяет, что элемент в нашем документе действительно изменился; это означает, что наш код успешно подписался на событие и выполнил свою работу.

Заметим, что нас не интересует логика во вспомогательном файле. Мы просто полагаемся на наблюдаемые изменения состояния в пользовательском интерфейсе, которые служат итоговой точкой выхода. Это снижает уровень связывания

в тестах; таким образом, при изменении тестируемого кода с меньшей вероятностью придется изменять тест, если только наблюдаемая функциональность действительно не изменилась.

6.5. ИСПОЛЬЗОВАНИЕ DOM TESTING LIBRARY

Наш тест содержит большое количество шаблонного кода, в основном предназначенного для поиска элементов и проверки их содержимого. Я рекомендую обратиться к исходному коду библиотеки DOM Testing Library, написанной Кентом Доддсом (Kent C. Dodds) (<https://github.com/kentcdodds/dom-testing-library-with-anything>). У этой библиотеки имеются версии, совместимые с большинством современных фронтенд-фреймворков JavaScript, таких как React, Angular и Vue.js. Мы будем использовать ее базовую версию, которая называется DOM Testing Library.

В этой библиотеке мне нравится то, что она старается предоставить возможность писать тесты ближе к точке зрения пользователя, взаимодействующего с веб-страницей. Вместо того чтобы использовать идентификаторы для элементов, мы запрашиваем информацию по тексту элемента; инициирование событий реализовано немного чище; а запросы и ожидание появления или исчезновения элементов также реализованы чище и скрыты под синтаксическим сахаром. Все это весьма полезно, особенно при использовании в многочисленных тестах.

Вот как наш тест выглядит с этой библиотекой:

Листинг 6.27. Использование DOM Testing Library в простом teste

```
const { fireEvent, findByText, getByText }           Импортирует некоторые
      = require("@testing-library/dom");               библиотечные API

const loadHtml = (fileRelativePath) => {
  const filePath = path.join(__dirname, "index.html");
  const innerHTML = fs.readFileSync(filePath);
  document.documentElement.innerHTML = innerHTML;
  return document.documentElement;
};

const loadHtmlAndGetUIElements = () => {
  const docElem = loadHtml("index.html");
  const button = getByText(docElem, "click me", { exact: false });
  return { window, docElem, button };
};
```



Библиотечные API требуют элемента `document` как основы для выполнения большей части работы

```

describe("index helper", () => {
  test("dom test lib button click triggers change in page", () => {
    const { window, docElem, button } = loadHtmlAndGetUIElements();
    fireEvent.load(window); ← Использует fireEvent API библиотеки
    fireEvent.click(button); ← для упрощения диспетчеризации событий
    //Ожидать до true или тайм-аут продолжительностью в 1 секунду
    expect(findByText(docElem, «clicked», { exact: false })).toBeTruthy(); ←
  });
}); ← Этот запрос ожидает, пока элемент
      не будет найден, или завершается
      по тайм-ауту через 1 секунду

```

Обратите внимание на то, как библиотека позволяет использовать для получения элементов обычный текст элементов страницы (вместо их идентификаторов или идентификаторов тестов). Это один из способов делать так, чтобы происходящее выглядело более естественно и воспринималось с точки зрения пользователя. Чтобы тест оставался более стабильным с течением времени, мы используем флаг `exact: false`, чтобы не беспокоиться о несоответствии регистра или пропущенных буквах в начале или конце строк. Тем самым снижается необходимость в модификации теста для небольших и менее важных изменений текста.

ИТОГИ

- Тестирование асинхронного кода напрямую приводит к созданию ненадежных (*flaky*) тестов, выполнение которых занимает много времени. Для решения этих проблем можно использовать два способа: выделение точки входа или выделение адаптера.
- Под выделением точки входа понимается выделение чистой логики в отдельные функции, которые рассматриваются как точки входа ваших тестов. Извлеченная точка входа может либо получать обратный вызов как аргумент, либо возвращать значение. Для простоты старайтесь использовать возвращаемые значения вместо обратных вызовов.
- Выделение адаптера подразумевает выделение зависимости, асинхронной по своей природе, и ее абстрагирование для замены чем-то синхронным. Адаптер может иметь разные типы:
 - *модульный* — с созданием стаба для всего модуля (файла) и заменой в нем отдельных функций;
 - *функциональный* — с внедрением функции или значения в тестируемую систему. Внедренное значение может заменяться стабом в тестах;
 - *объектно-ориентированный* — когда вы используете интерфейс в рабочем коде и создаете стаб, который реализует этот интерфейс в коде теста.

- Таймеры (например, `setTimeout` и `setInterval`) можно заменять либо напрямую через манки-патчинг, либо при помощи Jest или другого фреймворка для их отключения и контроля.
- События лучше тестировать посредством проверки конечного результата, который они производят, — изменений в документе HTML, видимых пользователю. Это можно делать либо напрямую, либо с использованием таких библиотек, как DOM Testing Library.

Часть 3

Код тестов

В этой части рассматриваются способы управления, организации и достижения высокого качества юнит-тестов в реальных проектах.

В главе 7 говорится о достоверности тестов. В ней объясняется, как писать тесты, которые достоверно сообщают о наличии или отсутствии ошибок. Также будут рассмотрены различия между истинными и ложными сбоями тестов.

Глава 8 посвящена главному критерию хороших юнит-тестов — простоте сопровождения, а также методам ее обеспечения. Чтобы тесты были полезными в долгосрочной перспективе, они не должны требовать значительных усилий по сопровождению; в противном случае их перестанут запускать.

7

Достоверные тесты

В ЭТОЙ ГЛАВЕ

- ✓ Как определить, что результату теста можно доверять
- ✓ Обнаружение недостоверных непроходящих тестов
- ✓ Обнаружение недостоверных проходящих тестов
- ✓ Проблема ненадежности тестов

Как бы вы ни структурировали свои тесты, сколько бы их ни было, они ничего не стоят, если вы им не доверяете, не можете прочесть или сопровождать. Чтобы тесты, написанные вами, могли считаться хорошими, они должны обладать тремя свойствами.

- *Достоверность* — разработчики хотят, чтобы запускаемые тесты были достоверными, а в их результатах можно было быть уверенным. Достоверные тесты не содержат ошибок и тестируют именно то, что нужно.
- *Простота в сопровождении* — тесты, которые невозможно нормально сопровождать, превращаются в настоящий кошмар, потому что они могут нарушать дедлайны проектов или просто откладываются, когда проект идет по более сжатому графику. Разработчики просто прекращают сопровождение и исправление ошибок в тестах, модификация которых занимает слишком

много времени или которые должны изменяться слишком часто при незначительных изменениях в рабочем коде.

- *Читабельность* — относится не только к возможности прочитать тест, но и найти проблему, если с тестом что-то не так. Без читабельности два остальных столпа хорошего теста довольно быстро рушатся: сопровождение тестов усложняется, и вы уже не можете им доверять, потому что не понимаете их.

В этой и следующих двух главах представлены некоторые практики, которые относятся к каждому из этих требований и которые можно использовать при реview тестов. Соблюдение всех трех гарантирует, что ваше время было потрачено не зря. Стоит нарушить одно из них, и усилия многих людей могут пойти прахом.

Достоверность — первое из трех свойств, или требований, по которым я оцениваю качество тестов, поэтому будет уместно начать с него. Если мы не доверяем тестам, какой смысл запускать их? Какой смысл исправлять их или код, если они не проходят? Какой смысл тратить время на их сопровождение?

7.1. КАК ОПРЕДЕЛИТЬ, ЧТО ТЕСТУ МОЖНО ДОВЕРЯТЬ

Что означает «доверие» со стороны разработчика в контексте теста? Вероятно, это проще объяснить, основываясь на том, что вы делаете (или не делаете), если тест проходит (или не проходит).

Скорее всего, вы не доверяете тесту, если:

- он не проходит и вас это не беспокоит (вы полагаете, что это непрошедший позитивный тест);
- вы чувствуете, что результаты теста можно спокойно игнорировать либо из-за того, что он проходит в отдельных случаях, либо вы считаете, что он неактуален или работает с ошибками;
- он проходит, но вас это беспокоит (вы считаете, что это прошедший негативный тест);
- вы все еще ощущаете необходимость в ручной отладке или тестировании кода «на всякий случай».

Тесту можно доверять, если:

- тест не проходит и вас по-настоящему беспокоит, что в программе что-то сломалось. Вы не двигаетесь дальше и не успокаиваете себя, что тест ошибочен;
- тест проходит и вы чувствуете облегчение, а не необходимость что-то тестировать или отлаживать вручную.

В нескольких ближайших разделах мы рассмотрим сбои тестов как способ выявления недостоверных тестов, затем проанализируем код проходящих тестов и посмотрим, как обнаружить код недостоверных тестов. Наконец, мы рассмотрим несколько общепринятых практик, которые могут повысить достоверность.

7.2. ПОЧЕМУ ТЕСТЫ НЕ ПРОХОДЯТ

В идеале тесты (любые, не только юнит-тесты) должны не проходить только по убедительной причине. Конечно, такой причиной должен быть настоящий баг, обнаруженный в рабочем коде.

К сожалению, тесты могут проваливаться и по другим причинам. Можно предположить, что непрохождение теста по любой другой причине, кроме указанной выше, должно стать сигналом «недостоверности», но не все тесты проваливаются одинаково. Если вы будете понимать, почему тесты могут не проходить, это поможет составить план того, что следует делать в каждом случае.

Некоторые причины, по которым тесты могут не проходить:

- в рабочем коде была обнаружена реальная ошибка;
- тест не проходит из-за ошибки в самом тесте;
- тест устарел из-за изменения функциональности;
- тест конфликтует с другим тестом;
- тест ненадежен.

Кроме первого пункта, все остальные причины указывают на то, что тесту в его текущей форме не следует доверять. Давайте последовательно разберем все эти пункты.

7.2.1. В рабочем коде была обнаружена реальная ошибка

Первая возможная причина — наличие ошибки в рабочем коде. И это хорошо! Тесты для этого и создаются. Переходим к другим возможным причинам.

7.2.2. Тест не проходит из-за ошибки в самом teste

Рабочий код может быть правильным, но это ничего не значит, если в самом teste допущена ошибка, из-за которой он не проходит. Может быть, в проверке задействован неправильный ожидаемый результат точки выхода или тестируемая система используется некорректно. А может, вы неправильно настроили контекст для теста или не поняли, что именно должно тестироваться.

Как бы то ни было, тест с ошибками весьма опасен; ошибка в тесте также может привести к тому, что он *пройдет* и вы не будете знать, что в действительности происходит. О тестах, которые проходят, хотя и не должны, будет рассказано далее в этой главе.

Как распознать тест с ошибкой

У вас имеется тест, который не проходит, но вы уже провели отладку рабочего кода и не обнаружили никаких ошибок. В таком случае можно начать подозревать тест. Обходного пути не существует. Придется медленно и терпеливо отлаживать код теста.

Некоторые потенциальные причины ложных сбоев в тестах:

- проверка по неправильному критерию или неправильной точке выхода;
- внедрение неправильного значения в точку входа;
- неправильная активация точки входа.

Это может быть еще какая-нибудь мелкая ошибка, одна из тех, что мы соверша-ем, когда пишем код в 2 часа ночи. (Кстати, не самая жизнеспособная стратегия программирования. Не надо так делать.)

Что делать, когда вы нашли ошибку в тесте?

Если вы нашли тест с ошибкой, не переживайте. Возможно, это происходит уже в миллионный раз, и вы впадаете в панику и начинаете думать: «Наши тесты никуда не годятся». Не исключено, что вы правы, но это не повод для паники. Исправьте ошибку, запустите тест и посмотрите, проходит ли он.

Если тест проходит, не торопитесь радоваться! Перейдите к рабочему коду и включите очевидную ошибку, которая должна обнаруживаться исправленным тестом. Например, измените логическое значение, чтобы оно всегда было равно `true`. Или `false`. Затем снова запустите тест и убедитесь в том, что он не проходит. Если тест проходит, значит, в нем все еще содержится ошибка. Исправляйте тест, пока он не обнаружит внесенную ошибку в рабочем коде.

Когда вы будете уверены в том, что тест не проходит из-за очевидной проблемы с исходным кодом, исправьте только что внесенную ошибку и снова запустите тест. Если тест теперь проходит, работа завершена. Вы видели, что тест проходит там, где должен проходить, и не проходит там, где проходить не должен. Сделайте коммит кода и продолжайте работать дальше.

Если тест все еще не проходит, возможно, он содержит другую ошибку. Повторите весь процесс, пока не убедитесь в том, что тест проходит и проваливается именно там, где должен. Если тест все еще не проходит, возможно, ошибка все же кроется в рабочем коде. Если так — считайте, что вам повезло!

Как избежать будущих ошибок в тестах

Один из лучших известных мне способов обнаружения и предотвращения ошибок в тестах — методология разработки через тестирование, о которой я кратко рассказал в главе 1. Я также применяю ее в реальной работе.

Разработка через тестирование (TDD) позволяет увидеть оба состояния теста: то, что он не проходит там, где не должен проходить (исходное состояние, с которого мы начинаем), и то, что он проходит, когда должен проходить (когда будет написан тестируемый рабочий код, обеспечивающий прохождение теста). Если тест по-прежнему не проходит, значит, мы обнаружили ошибку в рабочем коде. Если тест проходит изначально, значит, ошибка присутствует в teste.

Другой хороший способ сокращения вероятности ошибок в тестах — исключение из них логики. Подробнее об этом рассказано в разделе 7.3.

7.2.3. Тест устарел из-за изменения в функциональности

Тест может не проходить, если он перестал быть совместимым с текущей тестируемой функциональностью. Допустим, у вас имеется функциональность входа, и в предыдущей версии для входа необходимо было предоставить имя пользователя и пароль. В новой версии старая система входа была заменена двухфакторной аутентификацией. Существующий тест перестанет проходить, потому что он не передает правильные параметры функциям входа.

Что можно сделать сейчас?

Возможны два варианта:

- адаптировать тест к новой функциональности;
- написать новый тест для новой функциональности и удалить старый тест, потому что он перестал быть актуальным.

Предотвращение подобных ситуаций в будущем

Все меняется. Тесты будут устаревать, и вряд ли этого можно избежать. В следующей главе речь пойдет об изменениях, относящихся к сопровождению тестов и тому, как тесты справляются с изменениями в приложении.

7.2.4. Тест конфликтует с другим тестом

Допустим, у вас имеются два теста: один не проходит, а другой проходит. Также допустим, что они не могут проходить одновременно. Обычно вы замечаете только проваливающийся тест, потому что проходящий... просто проходит.

Например, тест может не проходить, потому что он неожиданно конфликтует с новым поведением. С другой стороны, конфликтующий тест может ожидать

нового поведения, но не находит его. Простейший пример — первый тест убеждается в том, что при вызове функции с двумя параметрами возвращается результат «3», тогда как второй тест ожидает, что та же функция вернет «4».

Что можно сделать сейчас?

Основная причина заключается в том, что один из тестов перестал быть актуальным. Это означает, что его необходимо удалить. Какой тест нужно удалить? Этот вопрос следует задать владельцу продукта, потому что ответ связан с тем, какое поведение считается правильным и ожидается от приложения.

Предотвращение подобных ситуаций в будущем

На мой взгляд, это здоровая динамика в коде, и я не пытаюсь предотвращать ее.

7.2.5. Ненадежность теста

Тест может вести себя непоследовательно: то проходить, то проваливаться. Даже если тестируемый рабочий код не изменился, тест может вдруг провалиться без какой-либо очевидной причины, потом снова пройти, а после этого снова провалиться. Такие тесты называются ненадежными (flaky).

Ненадежные тесты заслуживают особого внимания. Они будут рассмотрены в разделе 7.5.

7.3. ИЗБЕГАНИЕ ЛОГИКИ В ЮНИТ-ТЕСТАХ

Вероятность присутствия ошибок в ваших тестах возрастает почти экспоненциально при включении в них новой логики. Я видел, как многие тесты, которые должны были быть простыми, превращаются в динамических монстров, генерирующих случайные числа, создающих потоки и выполняющих запись в файлы — по сути, небольшие тестовые машины. К сожалению, так как они были «тестами», автор не задумывался о том, могут ли они содержать ошибки и насколько просто их будет сопровождать. На отладку и верификацию этих тестов-монстров уходило больше времени, чем они могли сэкономить.

Но все монстры рождаются маленькими. Часто опытный разработчик в компании смотрит на тест и начинает думать: «А если включить в функцию цикл и генерировать случайные числа как входные данные? Конечно, так я смогу найти гораздо больше ошибок!» И это правда... особенно в ваших собственных тестах.

Ошибки в тестах — один из самых раздражающих моментов для разработчиков, потому что они почти никогда не ищут причину непрохождения теста в нем самом. Я не говорю, что тесты с логикой не обладают никакой ценностью.

Собственно, я и сам пишу такие тесты в некоторых особых ситуациях. Тем не менее я стараюсь избегать этой практики, насколько это возможно.

Если что-то из перечисленного ниже встречается в ваших юнит-тестах, значит, ваш тест содержит логику, которую я обычно рекомендую сократить или полностью удалить:

- команды `switch`, `if` или `else`;
- циклы `foreach`, `for` или `while`;
- конкатенация (знак + и т. д.);
- `try`, `catch`.

7.3.1. Логика в утверждениях: создание динамических ожидаемых значений

Для начала рассмотрим простой пример с конкатенацией.

Листинг 7.1. Тест, содержащий логику

```
describe("makeGreeting", () => {
  it("returns correct greeting for name", () => {
    const name = "abc";
    const result = trust.makeGreeting(name);
    expect(result).toBe("hello" + name);   ← | Логика в утверждении
  });
});
```

Чтобы вы поняли, какая проблема скрывается в этом тесте, в листинге 7.2 приведен тестируемый код. Обратите внимание на то, что знак + встречается в обоих листингах.

Листинг 7.2. Тестируемый код

```
const makeGreeting = (name) => {
  return "hello" + name;   ← | Такая же логика,
};                                         как в рабочем коде
```

Обратите внимание на то, как алгоритм (очень простой, но все равно алгоритм) соединения имени со строкой «hello» повторяется как в тесте, так и в тестируемом коде.

```
expect(result).toBe("hello" + name);   ← | Наш тест
return "hello" + name;   ← | Тестируемый код
```

Моя претензия к этому тесту заключается в том, что тестируемый алгоритм повторяется в самом teste. Это означает, что если в алгоритме присутствует ошибка, то тест будет содержать *ту же ошибку*. Тест не обнаружит ошибку, а будет ожидать неправильный результат от тестируемого кода.

В данном случае неправильный результат связан с пропущенным пробелом между объединяемыми словами, но, надеюсь, вы видите, как эта проблема может стать намного сложнее с более сложным алгоритмом.

Здесь проявляется проблема с достоверностью. Мы не верим, что тест сообщит нам правду, потому что его логика повторяет тестируемую логику. Тест может пройти, даже если в коде присутствует ошибка, так что мы не можем доверять результату теста.

ВНИМАНИЕ! Избегайте динамического построения ожидаемого значения в своих утверждениях; используйте фиксированные значения там, где это возможно.

Более достоверная версия этого теста могла бы выглядеть так, как показано в листинге 7.3.

Листинг 7.3. Более достоверный тест

```
it("returns correct greeting for name 2", () => {
    const result = trust.makeGreeting("abc");
    expect(result).toBe("hello abc"); ← Использует фиксированное значение
});
```

Так как входные данные в этом тесте очень просты, написать фиксированное ожидаемое значение несложно. Именно так я обычно рекомендую поступать: сделайте тестовые входные данные настолько простыми, чтобы вы могли легко создать фиксированную версию ожидаемого значения. Заметим, что это относится в основном к юнит-тестам. В тестах более высокого уровня сделать это несколько сложнее; это еще одна причина, по которой к таким тестам следует относиться с большей осторожностью. Часто они создают ожидаемые результаты динамически, а этого следует избегать везде, где только можно.

«Но послушай, Рой, — скажете вы. — Мы же повторяемся: строка "abc" встречается дважды. В предыдущем teste нам этого удалось избежать». Если приходится выбирать, то достоверность важнее простоты в сопровождении. Какой прок от теста с отличным сопровождением, если ему нельзя доверять? О дублировании кода в тестах можно прочитать в статье Владимира Хорикова «DRY vs. DAMP in Unit Tests» (<https://enterprisecraftsmanship.com/posts/dry-damp-unit-tests/>).

7.3.2. Другие формы логики

А вот противоположная ситуация: динамическое создание входных данных (в цикле) заставляет нас динамически решать, каким должен быть ожидаемый вывод. Допустим, требуется протестировать код из листинга 7.4.

Листинг 7.4. Функция поиска имен

```
const isName = (input) => {
  return input.split(" ").length === 2;
};
```

В листинге 7.5 представлен очевидный антипаттерн в teste.

Листинг 7.5. Циклы и if в teste

```
describe("isName", () => {
  const namesToTest = ["firstOnly", "first second", ""];
  it("correctly finds out if it is a name", () => {
    namesToTest.forEach((name) => {
      const result = trust.isName(name);
      if (name.includes(" ")) {
        expect(result).toBe(true);
      } else {
        expect(result).toBe(false);
      }
    });
  });
});
```

Обратите внимание на то, как мы используем несколько вариантов ввода для теста. Это заставляет нас перебирать эти варианты в цикле, что само по себе усложняет тест. Помните: в циклах тоже могут присутствовать ошибки.

Кроме того, поскольку мы определили несколько сценариев для значений (с пробелами и без), инструкция `if/else` должна знать, чего ожидает проверка, а в `if/else` тоже могут присутствовать ошибки. Мы также повторяем часть рабочего алгоритма, что возвращает нас к предыдущему примеру с конкатенацией и его проблемами.

Наконец, так как тест должен учитывать разные сценарии и ожидаемые результаты, ему можно присвоить только обобщенное имя вида «оно работает». Это плохо влияет на читабельность.

В общем, тест плох едва ли не по всем критериям. Лучше разделить его на два или три теста, каждый из которых имеет собственный сценарий и имя. Это позволит использовать фиксированные входные данные и проверки, а также удалить из кода циклы и логику `if/else` из кода. Любые более сложные решения создают следующие проблемы.

- Тест труднее прочитать и понять.
- Тест труднее воспроизвести. Например, представьте, что многопоточный тест или тест со случайными числами неожиданно не проходит.
- Тест с большей вероятностью содержит ошибки или проверяет неправильное условие.

- Выбор имени для теста усложняется, потому что он решает сразу несколько задач.

Как правило, тесты-монстры заменяют исходные более простые тесты, и это усложняет поиск ошибок в рабочем коде. Если вам все же приходится создавать такого монстра, он должен быть добавлен как новый тест, а не как замена для существующих. Кроме того, он должен находиться в проекте (или каталоге), имя которого явно указывает на то, что он содержит другие тесты вместо юнит-тестов. Я называю их «интеграционными» или «сложными» тестами и стараюсь свести их количество к допустимому минимуму.

7.3.3. Еще больше логики

Логика может присутствовать не только в тестах, но и во вспомогательных методах тестов, вручную написанных фейках и вспомогательных классах тестов. Помните: каждый фрагмент логики, который вы добавляете в указанных местах, существенно затрудняет чтение кода и повышает вероятность ошибок.

Если вы обнаружили, что по какой-то причине в набор тестов приходится включать сложную логику (хотя обычно я поступаю так с интеграционными, а не с юнит-тестами), по крайней мере, убедитесь в том, что вы включили в тестовый проект пару тестов для логики вспомогательных методов. Это избавит вас от многих неприятностей в будущем.

7.4. ЛОЖНОЕ ЧУВСТВО ДОВЕРИЯ К ПРОХОДЯЩИМ ТЕСТАМ

Мы рассмотрели непроходящие тесты как способ обнаружения тестов, которым не следует доверять. Но как насчет тихих, «зеленых» тестов, которые вас окружают? Следует ли доверять им? Как насчет теста, для которого необходимо провести код-ревью перед сохранением в главной ветви? На что следует обращать внимание?

Будем использовать термин «ложное доверие» для описания теста, которому на самом деле доверять не следует, но вы этого еще не знаете. Возможность рецензировать тесты и выявлять потенциальные проблемы ложного доверия чрезвычайно ценна, потому что вы не только можете сами исправить такие тесты, но и влияете на доверие всех остальных, кто когда-либо будет читать или запускать ваши тесты. Перечислю некоторые причины, снижающие мое доверие к тестам, даже если они проходят.

- Тест не содержит утверждений.
- Я не могу понять тест.

- Юнит-тесты смешиваются с ненадежными интеграционными тестами.
- Тест проверяет несколько условий или точек выхода.
- Тест продолжает изменяться.

7.4.1. Тесты, которые ничего не проверяют

Все согласятся с тем, что от теста, который ничего не проверяет на истинность или ложность, проку немного, верно? Он требует затрат времени на сопровождение, рефакторинг и чтение и иногда отвлекает от основной работы, если его приходится изменять из-за изменений API в рабочем коде.

Если вы видите тест без утверждений, учтите, что скрытые утверждения могут выполняться в вызовах функций. Это создает проблемы с читабельностью, если имя функции не было выбрано так, чтобы указывать на это обстоятельство. Иногда люди также пишут тесты, которые выполняют фрагмент кода, просто чтобы убедиться, что этот код не выдает исключение. Такие тесты приносят некоторую практическую пользу, и, если вы решили написать такой тест, убедитесь в том, что данный факт отражен в имени теста формулировкой вроде «не выдает исключение». Можно выразиться еще определеннее: многие тестовые API предоставляют возможность указать, что что-то не выдает исключение. Вот как это можно сделать в Jest:

```
expect(() => someFunction()).not.toThrow(error)
```

Если у вас есть такие тесты, постарайтесь, чтобы их было как можно меньше. Я не рекомендую использовать эту возможность как стандартную: она должна применяться только в немногих особых случаях.

Иногда люди просто забывают написать утверждение из-за отсутствия опыта. Подумайте, нельзя ли добавить недостающее утверждение или удалить тест, если он не имеет никакой ценности. Разработчики также могут активно писать тесты, чтобы достичь некоего воображаемого показателя тестового покрытия, установленного руководством. Такие тесты не служат никакой реальной цели. Они пишутся только для того, чтобы начальство отстало и инженеры могли заняться настоящим делом.

СОВЕТ Тестовое покрытие вообще не должно быть самостоятельной целью. Оно не гарантирует «качества кода». Более того, часто оно заставляет разработчиков писать бессмысленные тесты, которые приводят к еще большим тратам времени на сопровождение. Вместо этого лучше измерять «упущенные ошибки», «время исправления» и другие метрики, обсуждаемые в главе 11.

7.4.2. Тесты непонятны

Это серьезная проблема, которая будет подробно рассмотрена в главе 9. Возможны разные причины.

- Тесты с неудачно выбранными именами.
- Тесты слишком длинные или содержат запутанный код.
- Тесты содержат невнятные имена переменных.
- Тесты содержат скрытую логику или предположения, которые не удается понять с ходу.
- Неоднозначные результаты тестов (ни прохождение, ни сбой).
- Тестовые сообщения, не предоставляющие достаточной информации.

Если вы не понимаете тест, который проходит или проваливается, вы не знаете, нужно вам предпринимать какие-то действия или нет.

7.4.3. Смешение юнит-тестов с ненадежными интеграционными тестами

Говорят, ложка дегтя портит бочку меда. Это относится и к ненадежным тестам, смешиваемым с надежными. Интеграционные тесты окажутся ненадежными с гораздо большей вероятностью, чем юнит-тесты, потому что они имеют больше зависимостей. Если вы увидите, что в одном каталоге или в одной команде запуска тестов юнит-тесты смешиваются с интеграционными, это должно стать для вас тревожным сигналом.

Люди обычно выбирают пути наименьшего сопротивления; так же они действуют и в программировании. Допустим, разработчик выполняет все тесты, и один из них не проходит. Если можно обвинить в этом пропущенный параметр конфигурации или проблему с сетью вместо того, чтобы тратить время на исследования и исправление реальной проблемы, то инженер так и сделает. Это особенно часто происходит под давлением жестких сроков или если программист перегружен работой и уже отстает от графика.

Проще всего обвинить любой непроходящий тест в ненадежности. Если надежные и ненадежные тесты смешиваются друг с другом, такое будет происходить очень часто и позволит проигнорировать проблему и поработать над чем-то более интересным. Поскольку человеческий фактор убрать нельзя, лучше исключить возможность списать неудачу на ненадежность. Что для этого нужно сделать? Постарайтесь создать *безопасную зеленую зону*, храня интеграционные и юнит-тесты в разных местах.

Безопасная зеленая зона должна содержать только надежные, быстрые тесты, когда разработчики знают, что они могут получить последнюю версию кода,

выполнить все тесты в этом пространстве имен или каталоге и все тесты должны быть зелеными, то есть пройти (если рабочий код не изменился). Если какие-либо тесты в безопасной зеленой зоне не проходят, у программиста появляется очень серьезная причина для беспокойства.

Дополнительное преимущество такого разделения заключается в том, что разработчики с большей вероятностью будут чаще запускать юнит-тесты, так как без интеграционных тестов их выполнение ускоряется. Лучше иметь хоть какую-то обратную связь, чем не иметь ее вовсе, верно? Автоматизированный конвейер сборки должен позаботиться о запуске всех «недостающих» тестов обратной связи, которые разработчики не хотят или не могут выполнять на своих локальных машинах.

7.4.4. Тестирование нескольких точек выхода

Понятие *точки выхода*, или *потенциальной проблемы* (*concern*)¹, было рассмотрено в главе 1. Это один конечный результат единицы работы: возвращаемое значение, изменение состояния системы или вызов стороннего объекта.

Ниже приведен простой пример функции с двумя точками выхода (то есть двумя потенциальными проблемами). Функция возвращает значение и активирует переданную функцию обратного вызова.

```
const trigger = (x, y, callback) => {
  callback("I'm triggered");
  return x + y;
};
```

Можно написать тест, который будет проверять обе точки выхода одновременно.

Листинг 7.6. Проверка двух точек выхода в одном teste

```
describe("trigger", () => {
  it("works", () => {
    const callback = jest.fn();
    const result = trigger(1, 2, callback);
    expect(result).toBe(3);
    expect(callback).toHaveBeenCalledWith("I'm triggered");
  });
});
```

Первая причина, по которой проверка более чем одной точки выхода в teste может дать обратный эффект, — это нежелательные последствия для имени теста. Читабельность будет рассматриваться в главе 9, а пока ограничимся небольшим замечанием об именах: правильный выбор имен тестов чрезвычайно важен как

¹ На сленгге называют также «консерн». — Примеч. ред.

для отладки, так и для целей документирования. Я трачу немало времени на выбор хороших имен для тестов и не стыжусь признаться в этом.

Именование теста может показаться простым делом, но, если вы тестируете более одной сущности, выбрать для теста хорошее имя, указывающее, что именно тестируется, достаточно сложно. Часто получаются очень обобщенные имена, которые заставляют других людей вчитываться в код теста. Когда вы тестируете всего одну точку выхода, выбрать имя для теста проще. Но постойте, это еще не все.

К сожалению, в большинстве фреймворков юнит-тестов при неудачной проверке выдается особая разновидность исключения, которая перехватывается программой запуска фреймворка. Когда тестовый фреймворк перехватывает это исключение, это означает, что тест не прошел. Многие исключения в различных языках намеренно не позволяют коду продолжить выполнение. Таким образом, если строка

```
expect(result).toBe(3);
```

нарушит проверяемое условие, то следующая строка вообще не выполняется:

```
expect(callback).toHaveBeenCalledWith("I'm triggered");
```

Выход из тестового метода происходит в той же строке, в которой выдается исключение. Эти утверждения можно и нужно считать разными требованиями, и они также могут (и вероятно, в данном случае должны) быть реализованы по отдельности и последовательно, одна за другой.

Непрохождение проверки можно сравнить с симптомами болезни. Чем больше симптомов вы найдете, тем проще будет диагностировать болезнь. После сбоя последующие утверждения не выполняются, и вы можете упустить другие возможные симптомы, которые могут предоставить ценную информацию для целенаправленного поиска и выявления проблемы. Проверка нескольких требований в одном юнит-тесте повышает сложность с минимальной отдачей. Проверки дополнительных точек выхода должны происходить в отдельных, автономных юнит-тестах, чтобы вы видели, где именно произошел сбой.

Разобьем этот тест на два отдельных.

Листинг 7.7. Проверка двух точек выхода в отдельных тестах

```
describe("trigger", () => {
  it("triggers a given callback", () => {
    const callback = jest.fn();
    trigger(1, 2, callback);
    expect(callback).toHaveBeenCalledWith("I'm triggered");
  });

  it("sums up given values", () => {
```

```
const result = trigger(1, 2, jest.fn());
expect(result).toBe(3);
});
});
```

Точки выхода четко разделены, и каждый тест может проваливаться отдельно от другого.

Иногда вполне нормально проверять несколько утверждений в одном тесте, если они не представляют несколько точек выхода. Для примера возьмем следующую функцию и связанный с ней тест. Функция `makePerson` должна строить новый объект `person` с некоторыми свойствами.

Листинг 7.8. Использование нескольких утверждений для одной точки выхода

```
const makePerson = (x, y) => {
  return {
    name: x,
    age: y,
    type: "person",
  };
};

describe("makePerson", () => {
  it("creates person given passed in values", () => {
    const result = makePerson("name", 1);
    expect(result.name).toBe("name");
    expect(result.age).toBe(1);
  });
});
```

В нашем teste проверка выполняется одновременно по имени и возрасту, потому что они являются частями одной точки выхода (построение объекта `person`). Если первая проверка завершится неудачей, скорее всего, вторая уже не представляет интереса, так как что-то пошло не так во время исходного построения объекта.

СОВЕТ Критерий для разбиения теста: если первая проверка не проходит, то продолжает ли вас интересовать результат второй проверки? Если продолжает, то, вероятно, вам стоит разделить тест на два отдельных.

7.4.5. Тесты, которые продолжают изменяться

Если тест использует текущую дату и время как часть своего выполнения или проверок, можно утверждать, что при каждом запуске вы выполняете другой тест. То же можно сказать о тестах, использующих случайные числа, имена машин и вообще все, что может получать текущее значение из-за пределов среды теста. Появляется высокая вероятность того, что результаты будут

непоследовательными, а значит, тест будет ненадежным. Для нас, разработчиков, ненадежность тестов снижает доверие к их результатам (о чем я расскажу в следующем разделе).

У динамически генерируемых значений есть еще одна огромная потенциальная проблема: если мы не знаем заранее, какой ввод получит система, ожидаемый вывод также придется вычислять, а это может привести к ошибочному тесту, зависящему от повторения рабочей логики, как упоминалось в разделе 7.3.

7.5. НЕНАДЕЖНЫЕ ТЕСТЫ

Термином *ненадежные (flaky) тесты* описываются тесты, которые при отсутствии изменений в коде выдают непоследовательные результаты. Это может происходить очень часто или очень редко, но все равно происходит.

На рис. 7.1 показаны возможные источники ненадежности. Диаграмма основана на количестве реальных зависимостей у тестов. Также на это можно взглянуть иначе: сколько «подвижных частей» имеет тест? В этой книге нас в основном интересует нижняя треть диаграммы: юнит-тесты и компонентные тесты. Тем не менее я также хочу затронуть высокоуровневую ненадежность и дать рекомендации относительно того, что следует изучить подробнее.

На самом нижнем уровне тесты полностью контролируют свои зависимости, а следовательно, не имеют «подвижных частей»: либо потому что они заменяют их фейками, либо потому что выполняются полностью в памяти и могут настраиваться. Мы делали это в главах 3 и 4. Пути выполнения в коде полностью детерминированы, потому что все исходные состояния и ожидаемые возвращаемые значения от разных зависимостей могут быть определены заранее. Путь почти статичен: если он не возвращает ожидаемый результат, значит, что-то важное могло измениться на пути выполнения или в логике рабочего кода.

Когда мы начинаем подниматься к более высоким уровням, тесты создают все больше стабов и моков и начинают использовать все больше реальных зависимостей: баз данных, сетей, конфигурации и т. д. В свою очередь, это означает увеличение числа «подвижных частей», которые мы не контролируем, и может привести к изменению пути выполнения, возвращению неожиданных результатов или невозможности выполнения.

На верхнем уровне фейковых зависимостей нет. Все, от чего зависят наши тесты, полностью реально, включая сторонние сервисы, уровни сети и безопасности, конфигурацию и т. д. Тесты этого типа обычно требуют настройки среды, чтобы она была как можно ближе к рабочему сценарию, если только они не выполняются прямо в продакшне.

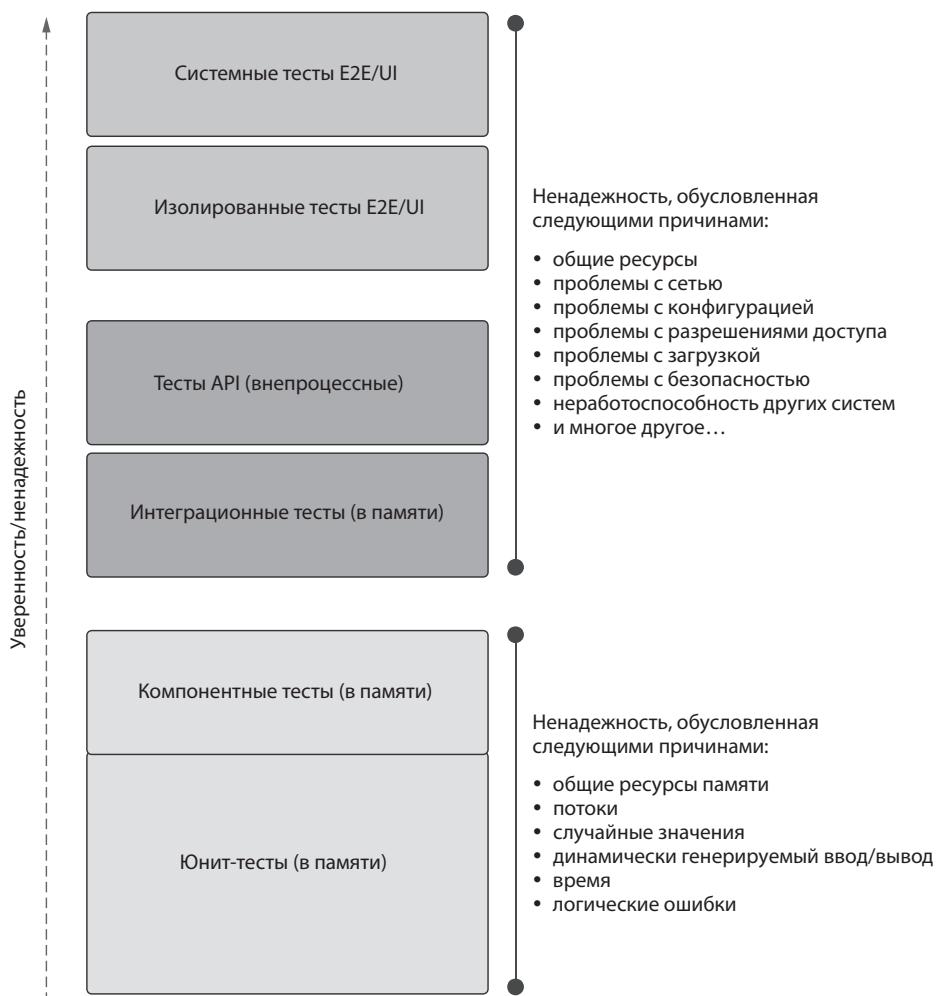


Рис. 7.1. Чем выше уровень тестов, тем больше реальных зависимостей они используют. Это дает нам уверенность в общей правильности работы системы, но снижает надежность тестов

Чем выше мы поднимаемся по диаграмме тестов, тем больше у нас уверенности в том, что код работает (если вы доверяете результатам тестов). К сожалению, чем выше мы поднимаемся, тем больше вероятность того, что тесты станут не-надежными из-за количества задействованных «подвижных частей».

Можно предположить, что тесты на нижнем уровне не должны иметь никаких проблем с надежностью, потому что здесь нет «подвижных частей», порождающих эту ненадежность. Теоретически это так, но на практике люди все равно

умудряются добавить подвижные части в тесты более низких уровней: с использованием текущей даты и времени, имени машины, сети, файловой системы и еще многое, что может подорвать надежность тестов.

Иногда тесты не проходят, хотя вы не прикасались к исходному коду. Примеры:

- тесты не проходят при каждом третьем запуске;
- тест не проходит только каждый N-й раз, где значение N неизвестно;
- тест не проходит при нарушении различных внешних условий, например доступности сети или базы данных, недоступности других API, конфигурации среды и т. д.

Добавьте еще к этому то, что каждая зависимость, используемая тестом (сеть, файловая система, потоки и т. д.), обычно повышает время выполнения. Вызовы к сети и базе данных занимают время. То же можно сказать об ожидании завершения потоков, чтении конфигураций и ожидании выполнения асинхронных задач.

Также вам потребуется больше времени для определения того, почему тест не проходит. Отладка теста или чтение больших объемов логов занимает умопомрачительно много времени и начинает постепенно затягивать вашу душу в пропасть «пора обновить мое резюме».

7.5.1. Что делать при обнаружении ненадежного теста?

Важно понимать, что ненадежные тесты могут дорого обойтись организации. В идеале в долгосрочной перспективе следует стремиться к нулевому проценту таких тестов. Вот несколько способов сократить затраты, связанные с ними;

- *Определите* смысл термина «ненадежный» для вашей организации. Например, запустите свой набор тестов 10 раз без изменений в рабочем коде и подсчитайте количество тестов с непоследовательными результатами (то есть тестов, которые не прошли все 10 раз и не провалились все 10 раз).
- Поместите любые тесты, считающиеся ненадежными, в специальную категорию или каталог для тестов, которые должны запускаться отдельно. Я рекомендую удалить все ненадежные тесты из обычной сборки, предназначеннной для поставки, чтобы они не создавали шума, и временно ограничить их собственным «карантинным» конвейером сборки. Затем вы перебираете все ненадежные тесты и играете в свою любимую игру «исправить, преобразовать или уничтожить».
 - *Исправить* — если возможно, сделайте тест надежным за счет контроля за его зависимостями. Например, если тест требует наличия данных в базе данных, вставьте эти данные как часть теста.
 - *Преобразовать* — устраните ненадежность переводом теста на более низкий уровень за счет удаления одной или нескольких зависимостей

и контроля над ними. Например, смоделируйте конечную точку сети стабом вместо использования реальной конечной точки.

- **Уничтожить** — серьезно поразмыслите над тем, достаточна ли ценность теста, чтобы продолжать его запуски и оплачивать все затраты на его сопровождение. Иногда лучшее, что можно сделать со старыми ненадежными тестами, — просто похоронить и забыть. Иногда они уже покрываются новыми, более эффективными тестами, а старые остаются обычным «техническим долгом», от которого можно избавиться. К сожалению, многие технические менеджеры неохотно отказываются от них из-за иллюзии невозвратных издержек: в старые тесты вложено так много труда, что их удаление кажется недопустимым расточительством. Однако на этой стадии сохранение ненадежных тестов может привести к большим затратам, чем их удаление, поэтому я настоятельно рекомендую рассмотреть такую возможность.

7.5.2. Предотвращение ненадежности в высокоуровневых тестах

Если вы хотите предотвратить ненадежность в высокоуровневых тестах, лучше всего убедиться в том, что они будут воспроизводиться в любой среде после любого развертывания. Для этого могут понадобиться следующие меры.

- Отмена всех изменений, внесенных вашими тестами во внешние общие ресурсы.
- Независимость от других тестов, изменяющих внешнее состояние.
- Определенная степень контроля над внешними системами и зависимостями: позаботьтесь о том, чтобы у вас была возможность воссоздавать их по своему усмотрению (поиските в интернете «инфраструктура как код»), создайте пустышки, которые вы можете контролировать, или специальные тестовые учетные записи и надейтесь на то, что они останутся в безопасности.

В последнем пункте контроль за внешними зависимостями может быть усложнен и даже невозможен при использовании внешних систем, находящихся под управлением других компаний. В таких ситуациях стоит рассмотреть следующие варианты.

- Удалите некоторые высокоуровневые тесты, если их сценарии уже покрываются низкоуровневыми тестами.
- Преобразуйте часть высокоуровневых тестов в набор низкоуровневых.
- Если вы пишете новые тесты, рассмотрите возможность использования конвейерно-ориентированной стратегии тестирования с тестовыми рецептами (вроде того, который будет рассматриваться в главе 10).

ИТОГИ

- Если вы не доверяете тесту, когда он не проходит, это может привести к игнорированию реальной ошибки. Если вы не доверяете тесту, когда он проходит, все кончится большим количеством ручной отладки и тестиирования. Предполагается, что количество случаев обоих типов сокращается при помощи хороших тестов. Но если не сокращать эти случаи и при этом тратить время на написание тестов, которым вы не доверяете, то зачем их вообще писать?
- Тесты могут не проходить по многим причинам: реальная ошибка присутствует в рабочем коде, ошибка в тесте приводит к ложному сбою, тест устарел из-за изменения функциональности, тест конфликтует с другим или ненадежен. Допустимой может считаться только первая причина. Все остальные указывают на то, что тесту доверять нельзя.
- Избегайте в тестах сложности, например динамического построения ожидаемых значений или дублирования логики из рабочего кода. Такая сложность повышает вероятность появления ошибок в тестах и время, необходимое для их понимания.
- Если тест не содержит утверждений, если вы не можете понять, что он делает, если он работает в сочетании с ненадежными тестами (даже если сам тест надежен), если он проверяет несколько точек выхода или продолжает изменяться, такому тесту нельзя доверять в полной мере.
- Ненадежные тесты проходят или проваливаются непредсказуемым образом. Чем выше уровень теста, тем больше реальных зависимостей в нем используется; это дает уверенность в общей правильности работы системы, но приводит к более низкому уровню надежности. Чтобы лучше выявлять ненадежные тесты, поместите их в специальную категорию или папку и запускайте отдельно от других.
- Для снижения уровня ненадежности тестов либо исправьте их, либо преобразуйте ненадежные высокоуровневые тесты в более надежные низкоуровневые, либо удалите их.

8

Простота в сопровождении

В ЭТОЙ ГЛАВЕ

- ✓ Фундаментальные причины непрохождения тестов
- ✓ Каким должен быть код тестов, чтобы реже его изменять
- ✓ Достижение простоты в сопровождении проходящих тестов

Тесты позволяют ускорить разработку, если только они не замедляют ее из-за всех необходимых изменений. Если можно избежать изменения существующих тестов при изменении рабочего кода, появляется надежда на то, что наши тесты помогают, а не вредят. В этой главе мы поговорим о сопровождаемости тестов.

Тесты, которые трудно сопровождать, могут привести к нарушению дедлайнов работы над проектами. Когда проект идет по более жесткому графику, такие тесты часто откладывают в сторону. Разработчики просто перестают сопровождать и исправлять те, изменение которых требует слишком много времени или которые нужно изменять слишком часто после незначительных преобразований в рабочем коде.

Простота в сопровождении определяется тем, насколько часто мы вынуждены изменять тесты, и нам хотелось бы свести к минимуму количество таких случаев. Это желание заставляет нас задаться следующими вопросами о фундаментальных причинах происходящего.

- Когда мы замечаем, что тест не проходит, а следовательно, может требовать изменений?
- Почему тесты не проходят?
- Какие случаи непрохождения теста заставляют нас изменить его?
- Когда мы решаем изменить тест, даже если ничто не вынуждает делать этого?

В этой главе представлена подборка практик для достижения большей простоты в сопровождении. Она пригодится также при код-ревью тестов.

8.1. ИЗМЕНЕНИЯ ИЗ-ЗА НЕПРОХОДЯЩИХ ТЕСТОВ

Непроходящий тест обычно является первым признаком потенциальных проблем с сопровождаемостью. Конечно, нельзя исключать, что вы нашли реальную ошибку в рабочем коде, но, если это не так, по каким другим причинам тест мог не пройти? Я буду называть первые ситуации *истинными сбоями*, а провалы тестов, происходящие по другим причинам, кроме обнаружения ошибок в тестируемом рабочем коде, — *ложными сбоями*.

Если вы захотите измерить степень сопровождаемости тестов, можно начать с оценки количества ложных сбоев и причин для каждого сбоя за некоторый промежуток времени. Одна из таких причин уже рассматривалась в главе 7: когда тест содержит ошибку. Теперь давайте обсудим другие возможные причины для ложных сбоев.

8.1.1. Тест неактуален или конфликтует с другим тестом

Конфликт может возникнуть при включении в рабочий код новой функциональности, напрямую конфликтующей с одним или несколькими существующими тестами. Вместо того чтобы обнаружить ошибку, тест может обнаружить конфликтующие или новые требования. Также могут появиться проходящие тесты, ориентированные на новые ожидания относительно того, как должен вести себя рабочий код.

Либо существующий непроходящий тест перестал быть актуальным, либо новые требования неверны. Если исходить из правильности требований, вероятно, неактуальный тест можно просто удалить.

Заметим, что у правила об «удалении тестов» есть одно частое исключение, касающееся работы с *переключателями функциональности* (feature toggle). Переключатели функциональности будут рассмотрены в главе 10, когда мы будем говорить о стратегии тестирования.

8.1.2. Изменения в API рабочего кода

Тест может не проходить при изменении тестируемого рабочего кода, в результате которого функция или объект должны использоваться по-другому (даже если они предоставляют ту же функциональность). Такие ложные сбои относятся к категории «избегать, насколько это возможно».

Рассмотрим класс `PasswordVerifier`, конструктор которого должен получать два параметра (см. листинг 8.1):

- массив правил (каждый элемент представляет собой функцию, которая получает входные данные и возвращает логическое значение);
- интерфейс `ILogger`.

Листинг 8.1. Password Verifier с двумя параметрами конструктора

```
export class PasswordVerifier {
    ...
    constructor(rules: ((input) => boolean)[], logger: ILogger) {
        this._rules = rules;
        this._logger = logger;
    }
    ...
}
```

Также можно написать пару тестов наподобие приведенного в листинге 8.2.

Листинг 8.2. Тесты без фабричных функций

```
describe("password verifier 1", () => {
    it("passes with zero rules", () => {
        const verifier = new PasswordVerifier([], { info: jest.fn() });
        const result = verifier.verify("any input");
        expect(result).toBe(true);
    });
    it("fails with single failing rule", () => {
        const failingRule = (input) => false;
        const verifier =
            new PasswordVerifier([failingRule], { info: jest.fn() });
        const result = verifier.verify("any input");
        expect(result).toBe(false);
    });
});
```

Тест использует существующий API

Если взглянуть на эти тесты с точки зрения сопровождаемости, то в них кроется потенциал для изменений, которые вам, вероятно, придется внести в будущем.

Обычно код живет долго

Предположим, что код, написанный вами, проживет в кодовой базе не менее 4–6 лет, а иногда и десятилетие. Насколько вероятно, что за это время дизайн `PasswordVerifier` изменится? Даже простые изменения — такие, как передача дополнительных параметров конструктору или изменения типов параметров, — становятся более вероятными в долгосрочной перспективе.

Примеры изменений, которые могут произойти с `PasswordVerifier` в будущем:

- добавление или удаление параметров конструктора `PasswordVerifier`;
- изменение типа одного из параметров `PasswordVerifier`;
- изменение количества функций `ILogger` или их сигнатур.;
- изменение схемы использования: вместо явного создания нового экземпляра `PasswordVerifier` содержащиеся в нем функции будут вызываться напрямую.

Если произойдет что-то из перечисленного, сколько тестов вам придется изменить? Разумно заранее внести изменения во все тесты, в которых создается экземпляр `PasswordVerifier`, и предотвратить возможные проблемы.

Допустим, в будущем наши опасения оправдались: что-то изменилось в API рабочего кода. Предположим, изменилась сигнатура конструктора и вместо `ILogger` в ней теперь используется `IComplicatedLogger` (листинг 8.3).

Листинг 8.3. Критичные изменения в конструкторе

```
export class PasswordVerifier2 {
  private _rules: ((input: string) => boolean)[];
  private _logger: IComplicatedLogger;

  constructor(rules: ((input) => boolean)[],
             logger: IComplicatedLogger) {
    this._rules = rules;
    this._logger = logger;
  }
  ...
}
```

На данный момент вам придется изменять все тесты, которые напрямую создают экземпляры `PasswordVerifier`.

Фабричные функции ослабляют связь с созданием тестируемого объекта

Существует простой прием, избавляющий вас от будущих неприятностей: создание тестируемого кода абстрагируется, тогда изменения в конструкторе достаточно централизованно внести всего в одном месте. Функция, единственным предназначением которой является создание и предварительная настройка

экземпляра объекта, обычно называется *фабричной* функцией или методом. Более совершенной формой этого паттерна (которая здесь рассматриваться не будет) является паттерн «мать объектов» (Object Mother).

Фабричные функции помогают уменьшить последствия указанной проблемы. Следующие два листинга показывают, как можно было бы изначально написать тесты до изменения сигнатуры и как можно было бы легко адаптироваться к изменению сигнатуры. В листинге 8.4 создание `PasswordVerifier` выделено в отдельную централизованную фабричную функцию. Я проделал то же самое с `fakeLogger`: теперь он создается в отдельной фабричной функции. Если в будущем произойдет одно из изменений, перечисленных выше, достаточно изменить только фабричные функции; никакие изменения в коде тестов скорее всего не потребуются.

Листинг 8.4. Рефакторинг с созданием фабричной функции

```
describe("password verifier 1", () => {
  const makeFakeLogger = () => {
    return { info: jest.fn() };
  };

  const makePasswordVerifier = (
    rules: ((input) => boolean)[],
    fakeLogger: ILogger = makeFakeLogger() => {
      return new PasswordVerifier(rules, fakeLogger);
    }
  );

  it("passes with zero rules", () => {
    const verifier = makePasswordVerifier([]);
    const result = verifier.verify("any input");
    expect(result).toBe(true);
  });
}

Централизованная точка для создания fakeLogger
Централизованная точка для создания PasswordVerifier
Использование фабричной функции для создания PasswordVerifier
```

В листинге 8.5 я провел рефакторинг тестов в соответствии с изменением сигнатуры. Обратите внимание: при этом изменяются только фабричные функции, но не сами тесты. Это управляемые изменения, с которыми можно жить в реальном проекте.

Листинг 8.5. Рефакторинг фабричных методов для новой сигнатуры

```
describe("password verifier (ctor change)", () => {
  const makeFakeLogger = () => {
    return Substitute.for<IComplicatedLogger>();
  };

  const makePasswordVerifier = (
    rules: ((input) => boolean)[],
    fakeLogger: IComplicatedLogger = makeFakeLogger() => {
      return new PasswordVerifier(rules, fakeLogger);
    }
}
```

```
    return new PasswordVerifier2(rules, fakeLogger);
};

// Тесты остаются неизменными
});
```

8.1.3. Изменения в других тестах

Недостаточная изоляция тестов — одна из главных причин «тестовой блокады»; я неоднократно наблюдал это явление в ходе консультирования и работы над юнит-тестами. Основная концепция, о которой следует помнить, — тесты всегда должны запускаться в собственном маленьком мире, в изоляции от других тестов, даже если они проверяют ту же функциональность.

Тест, который кричал: «Сбой!»

В одном проекте, в котором я участвовал, юнит-тесты вели себя странно, а с течением времени странностей становилось только больше. Сегодня тест не проходил, а через пару дней вдруг начинал проходить. Еще через день он снова не проходил, казалось бы рандомно, а в других случаях проходил даже при изменении кода с удалением или изменением поведения. Дошло до того, что в какой-то момент разработчики стали говорить друг другу: «Да ладно, это нормально. Если иногда проходит, значит, он проходит».

При должном исследовании выяснилось, что из теста вызывался другой (ненадежный) тест и, если второй тест не проходил, это нарушало работу первого.

На то, чтобы распутать эту мешанину, нам понадобилось три дня — после месяца, потраченного на поиск различных обходных решений. Когда тест наконец-то заработал правильно, оказалось, что в нашем коде были реальные ошибки, которые мы игнорировали, потому что в teste были собственные ошибки и проблемы. История мальчика, который кричал: «Волки!», оказалась актуальной и для разработки.

Если тесты не были достаточно хорошо изолированы, они начинают мешать друг другу, а вы начинаете жалеть, что связались с юнит-тестированием, и обещаете никогда этим больше не заниматься. Я уже с этим сталкивался. Разработчики не считают нужным искать проблемы в тестах, так что при возникновении проблемы может потребоваться много времени, чтобы выяснить, что же идет не так. Самый явный признак недостаточной изоляции тестов — то, что я называю «предопределенным порядком выполнения тестов».

Предопределенный порядок выполнения тестов

Предопределенный порядок выполнения тестов возникает тогда, когда тест предполагает, что предыдущий тест был (или, наоборот, не был) выполнен первым, потому что он зависит от некоего общего состояния, которое задается или сбрасывается другим тестом. Например, если один тест изменяет общую переменную в памяти или некоторый внешний ресурс (допустим, базу данных), а другой тест зависит от значения этой переменной после выполнения первого теста, между тестами возникает зависимость, основанная на порядке их выполнения.

Этот факт дополняется тем, что многие программы для запуска тестов (тест-раннери) не гарантируют (и не могут, и, наверное, не должны гарантировать!), что тесты будут выполняться в определенном порядке. Таким образом, если вы запустили все свои тесты сегодня, а через неделю запустили их в новой версии тест-раннера, они могут выполняться не в том порядке, как раньше.

Чтобы продемонстрировать суть проблемы, рассмотрим простую ситуацию. На рис. 8.1 изображен объект `SpecialApp`, использующий объект `UserCache`. Кэш пользователей существует в виде одного экземпляра (синглтона), совместно используемого для реализации кэширования в приложении, и по случайности также для всех тестов. На рис. 8.1 изображена реализация `SpecialApp`, кэш и интерфейс `IUserDetails`.

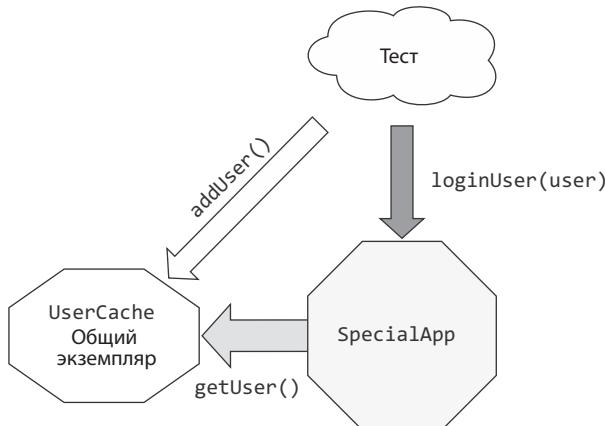


Рис. 8.1. Общий экземпляр UserCache

Листинг 8.6. Общий кэш и связанные с ним интерфейсы

```

export interface IUserDetails {
  key: string;
  password: string;
}

```

```
export interface IUserCache {  
    addUser(user: IUserDetails): void;  
    getUser(key: string);  
    reset(): void;  
}  
export class UserCache implements IUserCache {  
    users: object = {};  
    addUser(user: IUserDetails): void {  
        if (this.users[user.key] !== undefined) {  
            throw new Error("user already exists");  
        }  
        this.users[user.key] = user;  
    }  
    getUser(key: string) {  
        return this.users[key];  
    }  
    reset(): void {  
        this.users = {};  
    }  
}  
  
let _cache: IUserCache;  
export function getUserCache() {  
    if (_cache === undefined) {  
        _cache = new UserCache();  
    }  
    return _cache;  
}
```

В листинге 8.7 приведена реализация SpecialApp.

Листинг 8.7. Реализация SpecialApp

```
export class SpecialApp {  
    loginUser(key: string, pass: string): boolean {  
        const cache: IUserCache = getUserCache();  
        const foundUser: IUserDetails = cache.getUser(key);  
        if (foundUser?.password === pass) {  
            return true;  
        }  
        return false;  
    }  
}
```

Это упрощенная реализация для нашего примера, так что классу SpecialApp не стоит уделять слишком много внимания. Лучше взгляните на тесты в листинге 8.8.

Листинг 8.8. Тесты, которые должны выполняться в определенном порядке

```

describe("Test Dependence", () => {
  describe("loginUser with loggedInUser", () => {
    test("no user, login fails", () => {
      const app = new SpecialApp();
      const result = app.loginUser("a", "abc");
      expect(result).toBe(false);           | Требует, чтобы кэш
                                              | пользователей был пустым
    });

    test("can only cache each user once", () => {
      getUserCache().addUser({             ←
        key: "a",
        password: "abc",                 | Добавляет пользователя
                                              | в кэш
      });
      expect(() =>
        getUserCache().addUser({
          key: "a",
          password: "abc",
        })
      ).toThrowError("already exists");
    });

    test("user exists, login succeeds", () => {
      const app = new SpecialApp();
      const result = app.loginUser("a", "abc");
      expect(result).toBe(true);           | Требует, чтобы пользователь
                                              | присутствовал в кэше
    });
  });
});

```

Обратите внимание: первый и третий тесты зависят от второго теста. Первый тест требует, чтобы второй тест еще не был выполнен, потому что кэш должен быть пустым. В свою очередь, третий тест зависит от второго теста, который должен добавить в кэш ожидаемого пользователя. Если запустить только третий тест при помощи ключевого слова Jest `test.only`, то тест не пройдет.

```

test.only("user exists, login succeeds", () => {
  const app = new SpecialApp();
  const result = app.loginUser("a", "abc");
  expect(result).toBe(true);
});

```

Этот антипаттерн обычно встречается при попытке повторно использовать части тестов без выделения вспомогательных функций. Предполагается, что другой тест будет выполнен первым, избавляя нас от необходимости проводить инициализацию. И такой подход работает... до определенного времени.

Схема рефакторинга этой схемы состоит из нескольких шагов:

- выделения вспомогательной функции для добавления пользователя;

- повторного использования этой функции для нескольких тестов;
- сброса кэша пользователей между тестами.

Листинг 8.9 показывает, как провести рефакторинг тестов для предотвращения указанной выше проблемы.

Листинг 8.9. Рефакторинг тестов для устранения порядковой зависимости

```
const addDefaultUser = () => ← Выделяется вспомогательная
  getUserCache().addUser({           функция создания пользователя
    key: "a",
    password: "abc",
  });
const makeSpecialApp = () => new SpecialApp(); ← Выделяется фабричная
describe("Test Dependence v2", () => {           функция
  beforeEach(() => getUserCache().reset()); ← Сброс кэша между тестами
  describe("user cache", () => {
    test("can only add cache use once", () => {
      addDefaultUser(); ← Вызывает
      expect(() => addDefaultUser())           повторно
        .toThrowError("already exists");         используемые
      });                                     вспомогательные
    });
  });
  describe("loginUser with loggedInUser", () => {
    test("user exists, login succeeds", () => {
      addDefaultUser();
      const app = makeSpecialApp(); ← Новые
      const result = app.loginUser("a", "abc");
      expect(result).toBe(true);
    });
    test("user missing, login fails", () => {
      const app = makeSpecialApp();
      const result = app.loginUser("a", "abc");
      expect(result).toBe(false);
    });
  });
});
```

Здесь происходит последовательность событий. Сначала мы выделяем две вспомогательные функции: фабричную функцию `makeSpecialApp` и функцию `addDefaultUser`, пригодную для повторного использования. Затем создается очень важная функция `beforeEach`, которая сбрасывает кэш пользователей перед каждым тестом. Когда я использую подобные общие ресурсы, я почти всегда создаю функцию `beforeEach` или `afterEach`, которая возвращает ресурс к исходному состоянию перед или после запуска теста.

Первый и третий тесты теперь выполняются в собственной вложенной структуре `describe`. Кроме того, они оба используют фабричную функцию `makeSpecialApp`, и один из них использует `addDefaultUser`, чтобы его запуск не требовал выполнения до него другого теста. Второй тест также выполняется в собственной вложенной функции `describe` и повторно использует функцию `addDefaultUser`.

8.2. РЕФАКТОРИНГ ДЛЯ УЛУЧШЕНИЯ СОПРОВОЖДАЕМОСТИ

До настоящего момента мы рассматривали сбои тестов, которые заставляли нас вносить изменения в код. Теперь обсудим изменения, которые мы *сознательно* вносим для того, чтобы упростить сопровождение тестов с течением времени.

8.2.1. Избегайте тестирования приватных или защищенных методов

Этот раздел в большей степени относится к объектно-ориентированным языкам и TypeScript. Доступ к приватным или защищенным методам обычно ограничивается по веской причине, понятной их разработчику. Иногда это делается для скрытия подробностей реализации, чтобы она могла позднее изменяться без изменения наблюдаемого поведения. Также ограничение доступа может быть обусловлено причинами, связанными с безопасностью или правами интеллектуальной собственности.

Когда вы тестируете приватный метод, то проводите тестирование в условиях контракта, внутреннего для системы. Внутренние контракты динамичны, и они могут изменяться при рефакторинге системы. Когда они изменяются, тест может не пройти из-за того, что некоторая внутренняя работа выполняется по-другому, притом что общая функциональность системы осталась неизменной. Для целей тестирования вас должен интересовать только публичный контракт (наблюдаемое поведение). Тестирование функциональности приватных методов может привести к нарушению работоспособности тестов даже в том случае, если наблюдаемое поведение остается правильным.

На ситуацию можно взглянуть так: приватные методы не существуют в вакууме. Они должны кем-то вызываться, иначе никогда не будут активированы. Обычно существует публичный метод, который в ходе своего выполнения активирует приватный, а если нет, то где-то выше в цепочке вызовов всегда найдется активируемый публичный метод. Это означает, что приватный метод всегда является частью большей единицы работы или сценария использования в системе. Этот сценарий начинается с публичного API и завершается одним из трех конечных результатов: возвращаемым значением, изменением состояния или сторонним вызовом (или всеми тремя).

Таким образом, когда вы видите приватный метод, найдите в системе публичный сценарий использования, который этот метод активирует. Если вы протестирували только приватный метод и он сработал, это не означает, что остальные части системы правильно используют этот метод или обрабатывают предоставленные им результаты. Система может идеально работать внутри, но все эти идеальные внутренние механизмы будут неправильно использоваться через публичные API.

Если приватный метод заслуживает тестирования, иногда его стоит сделать публичным, статическим или по крайней мере внутренним и определить публичный контракт для любого кода, в котором он используется. В некоторых случаях архитектура получается более чистой, если поместить этот метод в другой класс. Вскоре мы рассмотрим подобные приемы.

Означает ли это, что в кодовой базе в итоге не должно быть приватных методов? Нет. В TDD вы обычно пишете тесты для публичных методов, а они позднее подвергаются рефакторингу с вызовом меньших, приватных методов. При этом тесты для публичных методов продолжают проходить.

Объявление методов публичными

Преобразование метода в публичный не обязательно плохо. В мире функционального программирования это вообще не создает проблем. Казалось бы, эта практика противоречит принципам объектно-ориентированного программирования, на которых многие из нас воспитывались, но это не всегда так.

Ваше желание протестировать метод может означать, что метод обладает известным поведением или контрактом относительно вызывающего кода. Объявляя его публичным, вы закрепляете этот контракт официально. Оставляя метод приватным, вы сообщаете всем разработчикам, которые придут после вас, что они могут изменить реализацию метода, не беспокоясь о неизвестном коде, который его использует.

Выделение методов в новые классы или модули

Если ваш метод содержит большой объем логики, которая может существовать самостоятельно, или использует специализированные переменные состояния в классе или модуле, актуальные только для этого метода, возможно, вам стоит выделить метод в новый класс или отдельный модуль с конкретной ролью в системе. После этого новый класс можно будет тестировать отдельно от других. В книге Майкла Физерса «Working Effectively with Legacy Code» приведены хорошие примеры этого приема, а книга Роберта Мартина «Clean Code»¹ (Pearson, 2008) поможет вам понять, почему так стоит поступать.

¹ Мартин Р. Чистый код. СПб.: Питер.

Преобразование приватных методов без состояния в публичные и статические

Если метод полностью лишен состояния, некоторые разработчики решают провести рефакторинг метода, делая его статическим (в языках, которые поддерживают данную возможность). Это существенно улучшает его тестируемость, а также явно указывает, что он является вспомогательным методом с известным публичным контрактом, определяемым по его имени.

8.2.2. Соблюдение принципа DRY в тестах

Дублирование в юнит-тестах может навредить разработчику не в меньшей, а то и в большей степени, чем дублирование в рабочем коде. Дело в том, что любое изменение в коде, содержащем дубликаты, заставляет вас также изменять все дубликаты. Когда вы имеете дело с тестами, повышается риск того, что разработчик просто решит обойти все эти проблемы и удалит или проигнорирует тесты вместо того, чтобы исправлять их.

Принцип DRY (Don't Repeat Yourself, то есть «не повторяйтесь») должен соблюдаться в коде тестов точно так же, как и в рабочем коде. Дублирование кода означает, что при изменении одного тестируемого аспекта вам также придется вносить изменения в других местах. Изменение конструктора или семантики использования класса может иметь серьезные последствия для тестов, содержащих большое количество дублируемого кода.

Как было показано в предыдущих примерах этой главы, использование вспомогательных функций может способствовать устраниению дублирования в тестах.

ВНИМАНИЕ! Устранение дублирования также может зайти слишком далеко и навредить читабельности. Об этом мы поговорим в следующей главе.

8.2.3. Нежелательность подготовительных методов

Мне не нравится функция `beforeEach` (также называемая *setup-функцией*, или *подготовительной функцией*), которая выполняется перед каждым тестом и часто используется для устранения дублирования. Я предпочитаю пользоваться вспомогательными функциями (хелперами). Подготовительными функциями слишком легко злоупотребить. Разработчики склонны использовать их для того, для чего они не предназначены, и в результате страдают читабельность и простота в сопровождении тестов.

Некоторые примеры злоупотреблений подготовительными методами со стороны разработчиков:

- инициализация в этих методах объектов, которые используются лишь в нескольких тестах из файла;

- наличие длинного и непонятного `setup`-кода;
- создание моков и фейковых объектов в подготовительном методе.

Кроме того, у подготовительных методов имеются ограничения, которые можно обойти при помощи простых вспомогательных методов.

- Они реально помогают только при инициализации данных.
- Они не всегда оказываются лучшими кандидатами на устранение дублирования. Удаление дублирования не всегда сводится к созданию и инициализации новых экземпляров объектов. Иногда оно связано с удалением дублирования в логике проверки или вынесением кода определенным образом.
- Они не могут иметь параметров или возвращаемых значений.
- Они не могут использоваться как фабричные методы, которые возвращают значения. Они выполняются перед выполнением тестов, поэтому характер их работы должен быть более общим. Тестам иногда приходится выдавать особые запросы или вызывать общий код с параметром для конкретного теста (например, получить объект и задать его свойству конкретное значение).
- Они должны содержать только код, относящийся ко всем тестам текущего класса; в противном случае метод будет труднее прочитать и понять.

Я почти полностью отказался от использования таких методов в своих тестах. Код тестов должен быть таким же простым и чистым, как и рабочий код. Впрочем, если ваш рабочий код выглядит ужасно, не используйте это как оправдание для написания нечитабельных тестов. Используйте фабричные и вспомогательные методы, и вы немного улучшите жизнь для поколения разработчиков, которым придется сопровождать ваш код через 5 или 10 лет.

ПРИМЕЧАНИЕ Пример перехода с использования `beforeEach` на вспомогательные функции представлен в разделе 8.2.3 (листинг 8.9), а также в главе 2.

8.2.4. Использование параметризованных тестов для устранения дублирования

Другая хорошая альтернатива для подготовительных методов, если все ваши тесты выглядят одинаково, — использование параметризованных тестов. Разные фреймворки тестирования в разных языках поддерживают параметризованные тесты, и, если вы используете Jest, в вашем распоряжении имеются встроенные функции `test.each` и `it.each`.

Параметризация помогает переместить логику подготовки, которая в противном случае осталась бы продублированной или находилась бы в блоке `beforeEach`, в часть подготовки тестов. Она также помогает избежать дублирования логики утверждения, как показано в листинге 8.10.

Листинг 8.10. Параметризованные тесты в Jest

```

const sum = numbers => {
  if (numbers.length > 0) {
    return parseInt(numbers);
  }
  return 0;
};

describe('sum with regular tests', () => {
  test('sum number 1', () => {
    const result = sum('1');
    expect(result).toBe(1);
  });
  test('sum number 2', () => {
    const result = sum('2');
    expect(result).toBe(2);
  });
});

describe('sum with parameterized tests', () => {
  test.each([
    ['1', 1],           ← Тестовые данные, используемые
    ['2', 2]            для сценария и утверждений
  ])('add ,for %s, returns that number', (input, expected) => {
    const result = sum(input);
    expect(result).toBe(expected); | Сценарий и утверждение без дублирования
  })
});

```

В первом блоке `describe` содержатся два теста, которые повторяют друг друга с разными входными значениями и ожидаемыми результатами. Во втором блоке `describe` используется вызов `test.each` для передачи массива массивов, в котором каждый подмассив перечисляет значения, необходимые для тестовой функции.

Параметризованные тесты могут сильно помочь с устранением дублирования между тестами. Тем не менее будьте осторожны и используйте этот метод только там, где многократно повторяется один сценарий и где изменяется только ввод и вывод.

8.3. ИЗБЕГАЙТЕ ИЗЛИШНЕЙ ДЕТАЛИЗАЦИИ

Тест с излишней детализацией содержит предположения относительно того, как конкретная тестируемая единица (рабочий код) должна реализовать свое внутреннее поведение, вместо того чтобы ограничиться проверкой правильности наблюдаемого поведения (точек выхода).

Некоторые случаи излишней детализации в юнит-тестах:

- тест проверяет чисто внутреннее состояние в тестируемом объекте;
- тест использует несколько моков;
- тест использует стабы в качестве моков;
- тест предполагает конкретный порядок точных совпадений строк там, где это не требуется.

Рассмотрим несколько примеров тестов с излишней детализацией.

8.3.1. Излишняя детализация внутреннего поведения с моками

Очень распространенный антипаттерн: проверять, что внутренняя функция в классе или модуле была вызвана, вместо того чтобы проверять точку выхода единицы работы. В листинге 8.11 приведена очередная реализация PasswordVerifier с вызовом внутренней функции, которая не должна представлять интерес для теста.

Листинг 8.11. Рабочий код с вызовом защищенной функции

```
export class PasswordVerifier4 {
    private _rules: ((input: string) => boolean)[];
    private _logger: IComplicatedLogger;

    constructor(rules: ((input) => boolean)[],
               logger: IComplicatedLogger) {
        this._rules = rules;
        this._logger = logger;
    }

    verify(input: string): boolean {
        const failed = this.findFailedRules(input); ←———— Вызов внутренней функции

        if (failed.length === 0) {
            this._logger.info("PASSED");
            return true;
        }
        this._logger.info("FAIL");
        return false;
    }

    protected findFailedRules(input: string) { ←———— Внутренняя функция
        const failed = this._rules
            .map((rule) => rule(input))
            .filter((result) => result === false);
        return failed;
    }
}
```

Обратите внимание: мы вызываем защищенную функцию `findFailedRules`, чтобы получить от нее результат, а затем выполняем вычисления с этим результатом.

Код теста приведен в листинге 8.12.

Листинг 8.12. Тест с излишней детализацией проверяет вызов защищенной функции

```
describe("verifier 4", () => {
  describe("overspecify protected function call", () => {
    test("checkfailedFules is called", () => {
      const pv4 = new PasswordVerifier4(
        [], Substitute.for<IComplicatedLogger>()
      );
      const failedMock = jest.fn(() => []);
      pv4["findFailedRules"] = failedMock;
      pv4.verify("abc");
      expect(failedMock).toHaveBeenCalled(); ← Проверка вызова внутренней
    });
  });
});
```

Создание мока для внутренней функции

← Проверка вызова внутренней функции. Не делайте так

Антипаттерн заключается в том, что мы предоставляем нечто, что не является точкой выхода. Мы проверяем, что в коде вызывается внутренняя функция, но что это на самом деле доказывает? Мы не проверяем правильность результата вычислений, а просто тестируем ради тестирования.

Если функция возвращает значение, обычно это указывает на то, что для нее нельзя создавать мок, потому что сам вызов функции не представляет точки выхода. Точной выхода становится значение, возвращаемое функцией `verify()`. Нас не должно интересовать даже то, существует ли вообще эта внутренняя функция.

Используя при проверке мок защищенной функции, которая не является точкой выхода, мы привязываем свою реализацию теста к внутренней реализации тестируемого кода, не приобретая каких-либо реальных преимуществ. Когда внутренние вызовы изменятся (а это непременно произойдет), нам также придется изменять все тесты, связанные с этими вызовами, а это занятие на любителя. О моках и их связи с хрупкостью тестов можно прочитать в главе 5 книги Владимира Хорикова «Unit Testing Principles, Practices, and Patterns»¹ (Manning, 2020).

¹ Хориков В. Принципы юнит-тестирования. СПб.: Питер.

Что делать вместо этого?

Взгляните на точку выхода. Реальная точка выхода зависит от типа теста, который вы хотите выполнить.

- *Тест на основе значения* — для тестов, основанных на значениях (которые я настоятельно рекомендую использовать там, где это возможно), нас интересует возвращаемое значение вызванной функции. В данном случае функция `verify` возвращает значение, поэтому она становится идеальным кандидатом для такого теста: `pv4.verify("abc")`.
- *Тест на основе состояния* — для теста, основанного на состоянии, мы ищем одноуровневую функцию (функцию, существующую на том же уровне области видимости, что и точка входа) или одноуровневое свойство, на которое влияет вызов функции `verify()`. Например, функции `firstname()` и `lastname()` могут считаться одноуровневыми. Здесь и следует выполнять проверку. В нашей кодовой базе вызов `verify()`, видимый на том же уровне, ни на что не влияет, так что данный пример не является хорошим кандидатом для такого типа тестирования.
- *Тесты со сторонними вызовами* — для таких тестов придется использовать `mok`, а для этого необходимо будет определить, где в коде находится точка вмешательства. Функция `findFailedRules` таковой не является, потому что она передает реальную информацию нашей функции `verify()`. В данном случае нет реальной сторонней зависимости, которую мы могли бы перехватить.

8.3.2. Излишняя детализация выводов

Распространенный антипаттерн — излишняя детализация порядка вывода и структуры набора возвращаемых значений. Обычно проще задать при проверке весь набор вместе со всеми элементами, но при таком подходе мы неявно берем на себя бремя исправления теста при изменении любой мелкой подробности этого набора. Вместо одной большой проверки следует разделить несколько аспектов верификации на меньшие неявные проверки.

В листинге 8.13 приведена функция `verify()`, которая получает несколько входных данных и возвращает список объектов-результатов.

Листинг 8.13. Функция, возвращающая список результатов

```
interface IResult {  
    result: boolean;  
    input: string;  
}  
  
export class PasswordVerifier5 {  
    private _rules: ((input: string) => boolean)[];  
    constructor(rules: ((input) => boolean)[]) {  
        this._rules = rules;  
    }  
  
    verify(...args: string[]): IResult[] {  
        const results: IResult[] = [];  
        for (const arg of args) {  
            const result = this._rules.find(rule => rule(arg));  
            if (result === undefined) {  
                results.push({ result: false, input: arg });  
            } else {  
                results.push({ result: true, input: arg });  
            }  
        }  
        return results;  
    }  
}
```

```

    this._rules = rules;
}

verify(inputs: string[]): IResult[] {
  const failedResults =
    inputs.map((input) => this.checkSingleInput(input));
  return failedResults;
}

private checkSingleInput(input: string): IResult {
  const failed = this.findFailedRules(input);
  return {
    input,
    result: failed.length === 0,
  };
}

```

Функция `verify()` возвращает массив объектов `IResult`, каждый из которых содержит поля `input` и `result`. В листинге 8.14 приведен тест, который неявно проверяет как порядок результатов, так и структуру каждого результата, а также значения результатов.

Листинг 8.14. Излишняя детализация результата

```

test("overspecify order and schema", () => {
  const pv5 =
    new PasswordVerifier5([input => input.includes("abc")]);

  const results = pv5.verify(["a", "ab", "abc", "abcd"]);

  expect(results).toEqual([
    { input: "a", result: false },
    { input: "ab", result: false },
    { input: "abc", result: true },
    { input: "abcd", result: true },
  ]);
});

```

Одна огромная проверка

Как этот тест может измениться в будущем? Возможных причин довольно много:

- при изменении длины массива `results`;
- когда в каждый объект-результат добавляется или из него удаляется свойство (даже если в teste эти свойства не используются);
- при изменении порядка вывода результатов (даже если он неважен для текущего теста).

Если в будущем произойдут какие-либо из этих изменений, а ваш тест сосредоточен на логике верификатора и структуре его вывода, сопровождение этого теста создаст массу проблем.

Часть этих проблем можно устраниить, проверяя только те части, которые важны для нас.

Листинг 8.15. Игнорирование схемы результата

```
test("overspecify order but ignore schema", () => {
  const pv5 =
    new PasswordVerifier5([(input) => input.includes("abc")]);

  const results = pv5.verify(["a", "ab", "abc", "abcd"]);

  expect(results.length).toBe(4);
  expect(results[0].result).toBe(false);
  expect(results[1].result).toBe(false);
  expect(results[2].result).toBe(true);
  expect(results[3].result).toBe(true);
});
```

Вместо полного ожидаемого вывода можно использовать для проверки значения конкретных свойств. Тем не менее это не решает проблему изменения порядка вывода результатов. Если он вас не интересует, можно просто проверить, содержит ли вывод конкретный результат, как показано в листинге 8.16.

Листинг 8.16. Игнорирование порядка вывода и схемы

```
test("ignore order and schema", () => {
  const pv5 =
    new PasswordVerifier5([(input) => input.includes("abc")]);

  const results = pv5.verify(["a", "ab", "abc", "abcd"]);

  expect(results.length).toBe(4);
  expect(findResultFor("a")).toBe(false);
  expect(findResultFor("ab")).toBe(false);
  expect(findResultFor("abc")).toBe(true);
  expect(findResultFor("abcd")).toBe(true);
});
```

Здесь функция `findResultFor()` помогает найти конкретный результат для заданного ввода. Теперь порядок вывода результатов может измениться, в них могут появиться новые значения, но тест не пройдет только в том случае, если изменится результат вычисления `true` или `false`.

Другой распространенный антипаттерн, который часто повторяется многими разработчиками, — проверка фиксированных строк в возвращаемом значении или свойствах единицы работы, тогда как на самом деле необходима лишь конкретная часть этой строки. Спросите себя: «Нельзя ли проверить, что строка *содержит* нужную подстроку, вместо проверки ее *равенства* чему-то?» В листинге 8.17 приведена версия Password Verifier, которая выводит сообщение с количеством правил, нарушенных в ходе верификации.

Листинг 8.17. Версия Password Verifier, возвращающая строковое сообщение

```
export class PasswordVerifier6 {
  private _rules: ((input: string) => boolean)[] = [];
  private _msg: string = "";
  constructor(rules: ((input) => boolean)[]) {
    this._rules = rules;
  }
  getMsg(): string {
    return this._msg;
  }
  verify(inputs: string[]): IResult[] {
    const allResults = inputs.map((input) => this.checkSingleInput(input));
    this.setDescription(allResults);
    return allResults;
  }
  private setDescription(results: IResult[]) {
    const failed = results.filter((res) => !res.result);
    this._msg = `you have ${failed.length} failed rules.`;
  }
}
```

В листинге 8.18 приведены два теста, в которых используется `getMsg()`.

Листинг 8.18. Излишняя детализация с применением равенства строк

```
describe("verifier 6", () => {
  test("over specify string", () => {
    const pv5 =
      new PasswordVerifier6([(input) => input.includes("abc")]);
    pv5.verify(["a", "ab", "abc", "abcd"]);
    const msg = pv5.getMsg();
    expect(msg).toBe("you have 2 failed rules.");
  });
});
```

//Более правильный способ записи этого теста

```
test("more future proof string checking", () => {
  const pv5 =
    new PasswordVerifier6([(input) => input.includes("abc")]);
  pv5.verify(["a", "ab", "abc", "abcd"]);
  const msg = pv5.getMsg();
  expect(msg).toMatch(/2 failed/);
});
```

Первый тест проверяет, что строка в точности равна другой строке. Такое решение часто порождает проблемы, потому что строки являются одной из форм пользовательского интерфейса. Разработчики часто слегка изменяют и дорабатывают их. Например, так ли важно для нас, что в конце строки стоит точка? Тест требует, чтобы это было важно, но вся суть проверки заключается в том, чтобы выводилось правильное число (особенно потому, что строки изменяются в зависимости от языка или культуры, но числа обычно остаются неизменными).

Второй тест просто ищет строку «`2 failed`» внутри сообщения. Тем самым обеспечивается большая устойчивость теста при будущих изменениях: строка может слегка измениться, но основное сообщение останется неизменным, и вам не придется изменять тест.

ИТОГИ

- Тесты растут и изменяются вместе с тестируемой системой. Если не уделять внимания сопровождаемости, тесты могут потребовать такого количества изменений, что результат не оправдывает затрат. Возможно, все кончится тем, что вы просто удалите тесты, а вся тяжелая работа по их созданию пойдет прахом. Чтобы тесты приносили пользу в долгосрочной перспективе, они должны не проходить только по причинам, которые для нас действительно имеют значение.
- *Истинным сбоем* называется непрохождение теста из-за того, что он обнаружил ошибку в рабочем коде. *Ложным сбоем* называется непрохождение теста по любой другой причине.
- Для оценки простоты сопровождения теста можно определять количество ложных сбоев тестов и причины каждого сбоя за некоторый промежуток времени.
- Ложный сбой теста может объясняться несколькими причинами: возможными конфликтами с другим тестом (в этом случае его следует просто удалить); изменениями в API рабочего кода (проблема частично устраняется использованием фабричных и вспомогательных методов); изменениями в других тестах (такие тесты следует изолировать друг от друга).
- Избегайте тестирования приватных методов. Приватные методы являются подробностями реализации, и тесты получатся хрупкими. Тесты должны проверять *наблюдаемое поведение* — то есть поведение, актуальное для пользователя. Иногда необходимость в тестировании приватного метода является признаком отсутствующей абстракции, что означает, что метод должен быть объявлен публичным или даже выделен в отдельный класс.

- Соблюдайте принцип DRY в своих тестах. Используйте вспомогательные методы для абстрагирования несущественных подробностей в частях подготовки (сетапа) и проверки. Это упростит ваши тесты без лишней привязки их друг к другу.
- Избегайте подготовительных методов — таких, как функция `beforeEach`. Вместо них лучше использовать вспомогательные методы. Другой возможный вариант — параметризация тестов и вынесение контента блока `beforeEach` в часть подготовки теста.
- Избегайте излишней детализации. Возможные примеры: проверка приватного состояния тестируемого кода, проверка вызовов стабов, проверка конкретного порядка элементов в наборе результатов или точных совпадений строк там, где они необязательны.

Часть 4

Проектирование и процесс

В завершающих главах рассматриваются проблемы, с которыми вы столкнетесь при внедрении юнит-тестирования в существующей организации или кодовой базе, и приемы, которые вам понадобятся для решения этих проблем.

Глава 9 посвящена читабельности тестов. Мы обсудим некоторые схемы именования тестов и их входные значения. Также будут рассмотрены лучшие практики структурирования тестов и написания более эффективных проверочных сообщений.

Глава 10 объясняет, как разработать стратегию тестирования. Вы узнаете, какие уровни тестирования предпочтительны при тестировании новой функциональности, мы обсудим распространенные антипаттерны на уровнях тестирования, а также обсудим стратегию рецептов тестирования.

В главе 11 будет рассматриваться непростая проблема реализации юнит-тестирования в организациях, а также показаны приемы, которые упростят вашу работу. Эта глава дает ответы на некоторые вопросы, часто встречающиеся на начальной стадии реализации юнит-тестирования.

В главе 12 описаны распространенные проблемы, связанные с унаследованным кодом, а также некоторые средства для работы с ним.

9

Читабельность

В ЭТОЙ ГЛАВЕ

- ✓ Схемы выбора имен юнит-тестов
- ✓ Написание читабельных тестов

Если написанный вами тест не будет читабельным, он станет практически бессмысленным для тех, кто будет использовать его в будущем. Читабельность — связующая нить между человеком, написавшим тест, и тем несчастным, которому придется читать его через несколько месяцев или лет. Тесты — это хронология проекта, которую вы рассказываете следующему поколению программистов. Они позволяют разработчику точно понять, как устроено приложение и с чего оно начиналось.

Вся суть этой главы заключается в том, чтобы разработчики, которые придут после вас, могли нормально сопровождать рабочий код и тесты, написанные вами. Они должны понимать, что делают и где им это нужно делать.

У читабельности есть несколько аспектов:

- выбор имен юнит-тестов;
- выбор имен переменных;
- отделение утверждений от действий;
- подготовка и завершение.

Давайте рассмотрим их один за другим.

9.1. ВЫБОР ИМЕН ЮНИТ-ТЕСТОВ

Стандарты именования важны. Они предоставляют удобные правила и шаблоны, которые описывают, что вы должны объяснить касательно своего теста. В каком бы порядке я их ни расставлял, какой бы конкретный фреймворк или язык ни использовал, я стараюсь убедиться в том, что в имени теста или структуре файла, в котором существует тест, присутствовали три важные информационные составляющие:

- точка входа в единицу работы (или имя проверяемой функциональности);
- сценарий, в котором тестируется точка входа;
- ожидаемое поведение точки выхода единицы работы.

Имя точки входа (или единицы работы) нужно для того, чтобы вы могли легко понять начальную область видимости тестируемой логики. Включение ее как первой части имени теста также упрощает навигацию и автозаполнение при вводе (если ваша IDE поддерживает эту возможность) в тестовом файле.

Сценарий, в котором проводится тестирование, определяет часть имени «с чем»: «Когда я вызываю точку входа X с null, она должна сделать Y».

Ожидаемое поведение точки выхода единицы работы — это место, в котором тест обычным языком объясняет, что должна сделать или вернуть единица работы или как она должна себя вести в зависимости от текущего сценария: «Когда я вызываю точку входа X с null, она должна сделать Y в видимости точки выхода единицы работы».

Для человека, читающего тест, эти три элемента должны находиться где-то на видном месте. Иногда все они могут инкапсулироваться в имени функции теста, а иногда их можно включить во вложенные структуры `describe`. В некоторых странах допустимо просто использовать строковое описание как параметр или аннотацию теста.

Примеры, приведенные в листинге 9.1, содержат одни и те же информационные составляющие, но представленные по-разному.

Листинг 9.1. Одна информация, разные представления

```
test('verifyPassword, with a failing rule, returns error based on
rule.reason', () => { ... }

describe('verifyPassword', () => {
  describe('with a failing rule', () => {
    it('returns error based on the rule.reason', () => { ... }

verifyPassword_withFailingRule_returnsErrorBasedonRuleReason()
```

Конечно, вы сможете предложить и другие способы структурирования. (Кто сказал, что нужно использовать нижние подчеркивания? Это всего лишь мое личное предпочтение, которое напоминает мне и другим, что информация состоит из трех частей.) Самое важное, что следует вынести из этого описания, — если удалить хотя бы одну из этих составляющих, то читателю кода для получения ответа придется читать код, содержащийся в тесте, и попусту тратить свое драгоценное время.

В листинге 9.2 приведены примеры тестов с недостающей информацией.

Листинг 9.2. Имена тестов с недостающей информацией

```
test('failing rule, returns error based on rule.reason', () => { ... }) ← Что здесь тестируется?
test('verifyPassword, returns error based on rule.reason', () => { ... }) ←
test('verifyPassword, with a failing rule', () => { ... }) ← Когда это должно
                                                               произойти?
                                                               ↑
                                                               что должно
                                                               случиться потом?
```

Ваша главная цель в отношении читабельности — освободить следующего разработчика от необходимости читать код вашего теста, чтобы понять, что же в нем тестируется.

Для включения всех этих информационных составляющих в имя теста есть и другая важная причина: как правило, имя — единственное, что выводится при сбое автоматизированного конвейера сборки. Вы видите имена непрощедших тестов в журнале неудачной сборки, но не видите ни комментариев, ни кода тестов. Если имена достаточно содержательны, возможно, вам не придется читать код или отлаживать тесты; не исключено, что вы поймете причину сбоя, просто читая журнал неудачной сборки. Это сэкономит вам драгоценное время, которое вы иначе потратили бы на отладку и чтение.

Хорошее имя теста также работает на концепцию исполняемой документации. Если вы можете предложить новому разработчику, который пришел в команду, почитать тесты, чтобы понять, как работает конкретный компонент или приложение, — это хороший признак читабельности. Если ему не удается понять поведение приложения или компонента по одним тестам — это тревожный сигнал.

9.2. МАГИЧЕСКИЕ ЗНАЧЕНИЯ И ИМЕНА ПЕРЕМЕННЫХ

Вам доводилось слышать термин «магические значения»? Звучит красиво, но на самом деле все наоборот. Правильнее было бы назвать их «ведьмовскими значениями», чтобы передать негативный эффект от их использования. Что

это такое, спросите вы? Это жестко фиксированные, недокументированные или малопонятные константы или переменные. Упоминание магии указывает на то, что эти значения работают, но вы понятия не имеете почему.

Возьмем следующий тест.

Листинг 9.3. Тест с магическими значениями

```
describe('password verifier', () => {
  test('on weekends, throws exceptions', () => {
    expect(() => verifyPassword('jh6Gu78!', [], 0)) ← | Магические
      .toThrowError("It's the weekend!");
  });
});
```

Этот тест содержит три магических значения. Легко ли будет понять человеку, который не написал этот тест и не знает тестируемого API, что означает `0`? А как насчет массива `[]`? Первый параметр этой функции выглядит как пароль, но даже в нем есть нечто магическое. Итак:

- `0` может означать все что угодно. Мне как читателю придется искать его в коде или перейти к сигнатуре вызываемой функции, чтобы понять, что этот параметр задает день недели;
- `[]` заставляет меня просмотреть сигнатуру вызываемой функции, чтобы понять, что функция ожидает получить массив правил проверки пароля; это значение указывает на то, что тест проверяет ситуацию без правил;
- `jh6Gu78!` — вроде бы очевидный пароль, но тут у читателя кода возникает большой вопрос: а почему именно конкретное значение? Чем так важен этот конкретный пароль? Очевидно, в этом тесте необходимо использовать именно это значение, а не какое-то другое, потому что оно задано предельно конкретно. На самом деле это не так, но читатель этого не знает. Скорее всего, он попробует использовать этот пароль в других тестах просто для надежности. Магические значения склонны размножаться в тестах.

В листинге 9.4 приведен тот же тест с исправленными магическими значениями.

Листинг 9.4. Исправление магических значений

```
describe("verifier2 - dummy object", () => {
  test("on weekends, throws exceptions", () => {
    const SUNDAY = 0, NO_RULES = [];
    expect(() => verifyPassword2("anything", NO_RULES, SUNDAY))
      .toThrowError("It's the weekend!");
  });
});
```

Помещая магические значения в переменные с содержательными именами, мы снимаем вопросы, которые могут возникнуть у людей при чтении нашего теста.

Что касается пароля, я просто изменил непосредственное значение, чтобы объяснить читателю, что в данном тесте оно *не играет* существенной роли.

Имена переменных и значения должны объяснять читателю не только то, что важно в тесте, но и то, что для него *неважно*.

9.3. ОТДЕЛЕНИЕ ПРОВЕРОК ОТ ДЕЙСТВИЙ

Ради читабельности и всего святого избегайте размещения проверок и вызовов методов в одной команде. Листинг 9.5 показывает, что я имею в виду.

Листинг 9.5. Отделение проверок от действий

```
expect(verifier.verify("any value")[0]).toContain("fake reason"); ← Плохой
const result = verifier.verify("any value");
expect(result[0]).toContain("fake reason"); | Хороший пример
```

Видите различия между этими двумя примерами? Первый пример намного труднее прочитать и понять в контексте реального теста из-за длины строки и вложения частей действия и проверки.

Кроме того, отладка второго примера будет намного проще, если вы захотите сосредоточиться на значении результата после вызова. Не пренебрегайте этим маленьким советом. Люди, пришедшие после вас, молча скажут вам спасибо за то, что ваши тесты им понятны и им не приходится чувствовать себя бестолковыми.

9.4. ПОДГОТОВКА И ЗАВЕРШЕНИЕ

Методами подготовки/настройки и завершения в юнит-тестах можно злоупотреблять до такой степени, что тесты или сами эти методы становятся нечитаемыми. С методами настройки ситуация обычно хуже, чем с завершающими.

В листинге 9.6 продемонстрировано одно возможное злоупотребление, которое встречается очень часто: использование подготовительной функции (`beforeEach`) для создания моков или стабов.

Листинг 9.6. Использование функции `beforeEach` для создания моков

```
describe("password verifier", () => {
  let mockLog;
  beforeEach(() => {
    mockLog = Substitute.for<IComplicatedLogger>(); ← Создание
  });
});
```

```

test("verify, with logger & passing, calls logger with PASS", () => {
  const verifier = new PasswordVerifier2([], mockLog);
  verifier.verify("anything");
  mockLog.received().info(
    Arg.is((x) => x.includes("PASSED")),
    "verify"
  );
});
});
});

```

Если вы будете создавать моки и стабы в подготовительном методе, это означает, что вам не придется делать это непосредственно в тесте. В свою очередь, это означает, что читатель вашего кода может даже не понимать, что в нем есть моки или чего тест ожидает от них.

Тест в листинге 9.6 использует переменную `mockLog`, которая инициализируется в функции `beforeEach` (подготовительный метод). Представьте, что в файле содержатся десятки и более таких тестов. Подготовительная функция располагается в начале файла, а вы застрияли над чтением теста где-то в конце. Вы сталкиваетесь с переменной `mockLog` и начинаете задавать вопросы: «Где она инициализируется? Как она ведет себя в teste?» и т. д.

Кроме того, использование нескольких моков или стабов в разных тестах одного файла может создать другую проблему: подготовительная функция превратится в свалку для разнообразных состояний, используемых вашими тестами. Она превращается в месиво с многочисленными параметрами; одни из них используются одним тестом, другие — другими. Понять такую подготовку и управлять ей становится все сложнее.

Инициализация моков в teste со всеми их ожиданиями читается намного лучше. Листинг 9.7 дает пример инициализации мока в каждом teste.

Листинг 9.7. Отказ от использования подготовительной функции

```

describe("password verifier", () => {
  test("verify, with logger & passing, calls logger with PASS", () => {
    const mockLog = Substitute.for<IComplicatedLogger>(); ← Когдa это должно
    const verifier = new PasswordVerifier2([], mockLog); произойти?
    verifier.verify("anything");

    mockLog.received().info(
      Arg.is((x) => x.includes("PASSED")),
      "verify"
    );
  });
});

```

Когда я смотрю на этот тест, все предельно ясно. Я вижу, где создается мок, каким поведением он обладает, и вообще все, что мне нужно знать.

Если вы заботитесь о читабельности, проведите рефакторинг и выделите создание мока во вспомогательную функцию, которая будет вызываться каждым тестом. Это позволит вам обойтись без создания обобщенной подготовительной функции, и вместо этого вы будете вызывать одну и ту же вспомогательную функцию из разных тестов.

Как видно из листинга 9.8, вы сохраняете всю читабельность, но сопровождать тест становится проще.

Листинг 9.8. Использование вспомогательной функции

```
describe("password verifier", () => {
  test("verify, with logger & passing, calls logger with PASS", () => {
    const mockLog = makeMockLogger(); ←
    const verifier = new PasswordVerifier2([], mockLog);
    verifier.verify("anything");

    mockLog.received().info(
      Arg.is((x) => x.includes("PASSED")),
      "verify"
    );
  });
});
```

Использование
вспомогательной
функции для
инициализации мока

Да, если следовать этой логике, вы увидите, что меня полностью устраивает *отсутствие* подготовительных функций в моих тестах. Я часто писал целые семейства тестов, в которых не было этих функций (вместо них использовались вызовы вспомогательных методов из каждого теста), ради сопровождаемости. И да, такие тесты легче читать и сопровождать.

ИТОГИ

- При именовании теста включите в него имя тестируемой единицы работы, текущий тестовый сценарий и ожидаемое поведение единицы работы.
- Не оставляйте в тестах магические значения. Либо упакуйте их в переменные с содержательными именами, либо включите описание в само значение, если оно представляет собой строку.
- Отделяйте утверждения от действий. Их слияние делает код более компактным, но существенно менее понятным.
- Страйтесь не использовать подготовительные методы (такие, как `beforeEach`). Добавьте вспомогательные методы, упрощающие фазу подготовки теста, и используйте их в каждом teste.

10

Разработка стратегии тестирования

В ЭТОЙ ГЛАВЕ

- ✓ Плюсы и минусы тестирования на различных уровнях
- ✓ Распространенные антипаттерны
- ✓ Стратегия рецептов тестирования
- ✓ Тесты, блокирующие и не блокирующие поставку
- ✓ Конвейеры поставки и информационные конвейеры
- ✓ Параллелизация тестов

Юнит-тесты представляют собой лишь одну из разновидностей тестов, которые вы можете (и должны!) писать. В этой главе мы обсудим, как юнит-тестирование укладывается в общую стратегию тестирования в масштабе организации. Как только мы начинаем рассматривать другие виды тестов, сразу же возникает целый ряд важных вопросов.

- На каком *уровне* должны тестироваться различные функциональности? (UI, бэкенд, API, единица работы и т. д.)
- Чем мы руководствуемся при принятии решения о том, на каком уровне тестировать функциональность? Стоит ли тестировать ее много раз на разных уровнях?

- Следует ли иметь больше функциональных сквозных тестов или больше юнит-тестов?
- Как оптимизировать скорость тестирования без потери достоверности?
- Кто должен писать тесты каждого вида?

Ответы на эти (и многие другие) вопросы образуют то, что я называю *стратегией тестирования*.

Для начала поговорим о том, какие бывают виды тестов.

10.1. ОСНОВНЫЕ ВИДЫ И УРОВНИ ТЕСТИРОВАНИЯ

В разных отраслях могут существовать разные виды и уровни тестирования. На рис. 10.1, который впервые рассматривался в главе 7, изображен довольно общий набор видов тестов, который, на мой взгляд, подходит для 90 % (если не более) организаций, которые я консультировал. Чем выше уровень, тем больше реальных зависимостей используется в тестах, что дает нам уверенность в правильности работы системы в целом. С другой стороны, такие тесты работают медленнее и менее надежны.

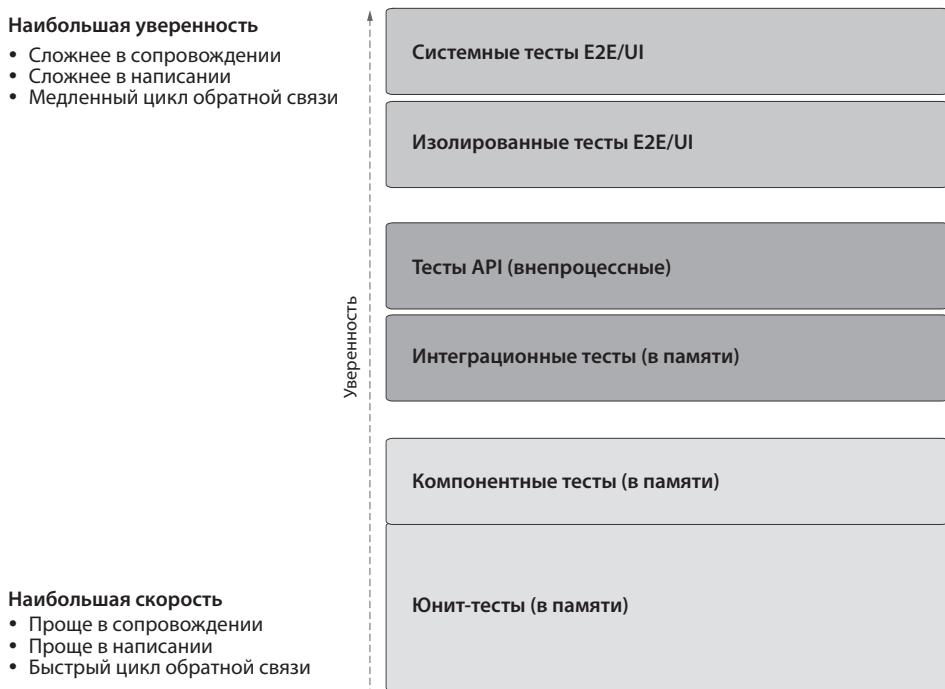


Рис. 10.1. Основные уровни тестирования ПО

Хорошая диаграмма, но что с ней делать? Она используется при дизайне фреймворка для принятия решения о том, какой тест написать. Есть несколько критериев (обстоятельств, которые делают нашу работу проще или сложнее), которые помогают мне решить, какой вид теста следует выбрать.

10.1.1. Критерии для выбора вида теста

Когда вы оказываетесь перед выбором более чем из двух вариантов, один из лучших способов принятия решения — определить *отличительные признаки* тестов, наилучшим образом подходящие для решения имеющейся задачи. Это такие свойства теста, в отношении которых существует консенсус, что они полезны (или, наоборот, их следует по возможности избегать). Они перечислены в таблице 10.1.

Все свойства тестов измеряются по шкале от 1 до 5. Как видите, у каждого уровня из рис. 10.1 есть свои достоинства и недостатки по каждому из критериев.

Таблица 10.1. Общие характеристики тестов

| Критерий | Шкала оценки | Примечания |
|-----------------------------|--------------|--|
| Сложность | 1–5 | Насколько сложно написать тест, прочитать или отладить его. Чем ниже, тем лучше |
| Ненадежность | 1–5 | Насколько вероятно, что тест не проходит из-за факторов, которые он не контролирует: кода других групп, сетей, баз данных, конфигурации и многого другого. Чем ниже, тем лучше |
| Уверенность при прохождении | 1–5 | Насколько прохождение теста повышает нашу уверенность в рабочем коде. Чем выше, тем лучше |
| Простота в сопровождении | 1–5 | Насколько часто тест должен изменяться и насколько он прост в изменении. Чем выше, тем лучше |
| Скорость выполнения | 1–5 | Насколько быстро завершается тест? Чем выше, тем лучше |

10.1.2. Юнит-тесты и компонентные тесты

Юнит-тесты и их разновидность, компонентные тесты, — виды тестов, которые рассматривались в этой книге до настоящего момента. Оба вида принадлежат к одной категории; единственное различие между ними заключается в том, что в компонентных тестах единица работы может содержать больше функций,

классов или компонентов. Иначе говоря, компонентные тесты содержат больше «материала» между точками входа и выхода.

Два примера демонстрируют эти различия.

- *Test A* – юнит-тест нестандартного UI-объекта (кнопки) в памяти. Вы можете создать экземпляр кнопки, щелкнуть на ней и увидеть, что она активирует ту или иную разновидность события щелчка.
- *Test B* – компонентный тест, который создает экземпляр компонента формы более высокого уровня и включает эту кнопку как часть своей структуры. Тест проверяет форму более высокого уровня, а кнопка играет небольшую роль как часть сценария.

При этом оба теста остаются юнит-тестами, работают в памяти и обладают полным контролем над всеми используемыми составляющими; в них нет зависимостей от файлов, баз данных, сетей, конфигурации или других зависимостей, которые мы не контролируем. Тест А является низкоуровневым юнит-тестом, а тест В – компонентным или высокоуровневым юнит-тестом.

Почему приходится проводить это различие? Потому что меня часто спрашивают, как бы я назвал тест с другим уровнем абстракции. Принадлежность теста к категории юнит-тестов / компонентных тестов определяется зависимостями, которые он содержит (или не содержит), а не используемым им уровнем абстракции. В таблице 10.2 приведены характеристики уровня юнит-тестов/компонентных тестов.

Таблица 10.2. Характеристики юнит-тестов / компонентных тестов

| | | |
|-----------------------------|-----|--|
| Сложность | 1/5 | Наименее сложные из всех типов тестов, что объясняется меньшей областью действия и тем фактом, что мы можем контролировать все части теста |
| Ненадежность | 1/5 | Наименее ненадежные из всех типов тестов, так как мы можем контролировать все части теста |
| Уверенность при прохождении | 1/5 | Хорошо, когда юнит-тест проходит, но это не вселяет особой уверенности в то, что наше приложение работает. Известно лишь, что работает небольшая его часть |
| Простота в сопровождении | 5–5 | Эти тесты создают меньше всего проблем с сопровождением из всех типов тестов, так как они относительно просто читаются и анализируются |
| Скорость выполнения | 5–5 | Самые быстрые из всех типов тестов, так как все работает в памяти без каких-либо жестких зависимостей от файлов, сети или баз данных |

10.1.3. Интеграционные тесты

Интеграционные тесты выглядят почти так же, как обычные юнит-тесты, но некоторые зависимости в них не заменяются стабами. Например, можно использовать реальную конфигурацию, реальную базу данных, реальную файловую систему или все сразу. Но чтобы активировать тест, мы все равно создаем экземпляр из рабочего кода в памяти и активируем функцию точки входа непосредственно с этим объектом. В таблице 10.3 приведены характеристики интеграционных тестов.

Таблица 10.3. Характеристики интеграционных тестов

| | | |
|-----------------------------|-------|---|
| Сложность | 2/5 | Эти тесты несколько (или значительно) более сложны в зависимости от количества зависимостей, для которых не создаются фейки в teste |
| Ненадежность | 2–3/5 | Эти тесты несколько (или значительно) более ненадежны в зависимости от количества используемых реальных зависимостей |
| Уверенность при прохождении | 2–3/5 | Прохождение интеграционного теста создает гораздо большую уверенность, потому что мы убеждаемся в том, что в коде успешно работает нечто неподконтрольное — например, база данных или конфигурационный файл |
| Простота в сопровождении | 3–4/5 | Эти тесты сложнее юнит-тестов из-за зависимостей |
| Скорость выполнения | 3–4/5 | Эти тесты несколько (или значительно) медленнее юнит-тестов из-за зависимостей от файловой системы, сети, базы данных или потоков |

10.1.4. Тесты API

На предшествующих, более низких, уровнях тесты не требовали развертывания тестируемого приложения или его корректного выполнения. На уровне тестов API приходится развертывать тестируемое приложение (по крайней мере, частично) и работать с ним по сети. В отличие от компонентных, интеграционных и юнит-тестов, тестов, которые можно отнести к категории тестов в памяти, тесты API являются внепроцессными. Тестируемая единица уже не создается непосредственно в памяти. Это означает, что в общей картине появляется новая зависимость: сеть, а также развертывание некоторых сетевых сервисов. В таблице 10.4 приведены характеристики тестов API.

Таблица 10.4. Характеристики тестов API

| | | |
|-----------------------------|-------|---|
| Сложность | 3/5 | Эти тесты несколько (или значительно) более сложны в зависимости от сложности развертывания, конфигурации и настройки API. Иногда в тест приходится включать схему API, что требует дополнительной работы и размышлений |
| Ненадежность | 3–4/5 | Сеть добавляет дополнительную ненадежность |
| Уверенность при прохождении | 3–4/5 | Прохождение тестов API создает еще большую уверенность. Мы можем рассчитывать на то, что после развертывания другие смогут вызывать наш API, доверяя правильности результатов |
| Простота в сопровождении | 2–3/5 | Сеть усложняет фазу подготовки и требует дополнительного внимания при изменении теста или добавлении/изменении API |
| Скорость выполнения | 2–3/5 | Сеть существенно замедляет выполнение тестов |

10.1.5. Изолированные тесты E2E/UI

На уровне изолированных сквозных (E2E, end-to-end) тестов, а также UI-тестов, или тестов пользовательского интерфейса, приложение тестируется с точки зрения пользователя. Я использую термин *изолированные*, чтобы указать, что тестируется *только* наше собственное приложение или сервис, без развертывания каких-либо других приложений или сервисов-зависимостей, которые могут понадобиться нашему. Такие тесты имитируют сторонние механизмы аутентификации, API других приложений, которые должны быть развернуты на том же сервере, а также любой код, который не является частью основного тестируемого приложения (включая приложения других отделов той же организации — для них также будут созданы фейки).

В таблице 10.5 приведены характеристики изолированных тестов E2E/UI.

Таблица 10.5. Характеристики изолированных тестов E2E/UI

| | | |
|--------------|-----|---|
| Сложность | 4/5 | Эти тесты намного сложнее предыдущих, так как в них приходится учитывать последовательность действий пользователя, изменения, обусловленные UI, а также фиксацию и анализ UI для интеграции и проверок. Частые ожидания и тайм-ауты |
| Ненадежность | 4/5 | Тест может замедляться, завершаться по тайм-ауту или не работать из-за большого количества задействованных зависимостей |

Таблица 10.5 (окончание)

| | | |
|-----------------------------|-------|---|
| Уверенность при прохождении | 4/5 | Прохождение таких тестов приносит большое облегчение. Мы обретаем значительную уверенность в работоспособности своего приложения |
| Простота в сопровождении | 1–2/5 | Увеличение числа зависимостей усложняет подготовку и требует большей осторожности при изменении теста или добавлении/изменении потоков операций. Такие тесты получаются длинными и обычно состоят из нескольких фаз |
| Скорость выполнения | 1–2/5 | Такие тесты могут быть очень медленными, так как они требуют операций с пользовательским интерфейсом, например аутентификации, кэширования, многостраничной навигации и т. д. |

10.1.6. Системные тесты E2E/UI

На уровне системных тестов E2E и UI фейков *нет*. Все работает настолько близко к развертыванию на продакшен, насколько это возможно: все приложения и сервисы-зависимости реальны, но они могут быть настроены иначе для наших тестовых сценариев. В таблице 10.6 приведены характеристики системных тестов E2E/UI.

Таблица 10.6. Характеристики системных тестов E2E/UI

| | | |
|-----------------------------|-----|--|
| Сложность | 5/5 | Тесты этого уровня сложнее всех в подготовке и написании из-за количества зависимостей |
| Ненадежность | 5/5 | Тесты могут не проходить по тысячам разных причин, часто по нескольким сразу |
| Уверенность при прохождении | 5/5 | Эти тесты дают наивысшую уверенность из-за объема кода, который проверяется при выполнении тестов |
| Простота в сопровождении | 1/5 | Тесты сложны в сопровождении из-за множества зависимостей и длинных потоков операций |
| Скорость выполнения | 1/5 | Тесты выполняются очень медленно, потому что они используют UI и реальные зависимости. Выполнение одного теста может занимать минуты и даже часы |

10.2. АНТИПАТТЕРНЫ В РАЗРАБОТКЕ СТРАТЕГИИ ТЕСТИРОВАНИЯ

Антипаттерны для разных уровней тестирования имеют не техническую, а скорее организационную природу. Вероятно, они вам уже встречались. Я как консультант могу сказать, что они чрезвычайно широко распространены.

10.2.1. Антипаттерн «только сквозные тесты»

Очень распространенная стратегия, применяемая во многих организациях, — использование в основном (и даже исключительно) тестов E2E (изолированных и системных). На рис. 10.2 показано, как это выглядит на диаграмме уровней тестов и их видов.

Почему это антипаттерн? Тесты на этом уровне выполняются очень медленно, сложны в сопровождении, создают трудности в отладке и крайне ненадежны. Эти затраты постоянны, тогда как пользы от каждого нового теста E2E становится все меньше.

Убывающая полезность тестов E2E

Первый тест E2E, который вы напишете, принесет наибольшую уверенность из-за количества кодовых путей, задействованных в этом сценарии, и из-за «клея» (кода, координирующего работу между вашим приложением и другими системами), который выполняется как часть этого теста.

Но как насчет второго теста E2E? Обычно он оказывается вариацией на тему первого, а это означает, что он может принести лишь незначительную долю той же пользы. Возможно, он отличается каким-нибудь полем со списком и добавлением других UI-элементов, но все зависимости, включая базу данных и сторонние системы, остаются прежними.

Дополнительная уверенность, которую вы получите от *второго* теста E2E, тоже составляет лишь небольшую долю той уверенности, которую вы получили от первого теста. Однако этого нельзя сказать о затратах на отладку, изменение, чтение и запуск этого теста; по сути, они будут такими же, как для предыдущего. Вы создаете большой объем дополнительной работы ради небольшой дополнительной уверенности, поэтому я и говорю, что польза от тестов E2E быстро убывает.

Если вам нужна вариация на тему первого теста, будет намного практичнее провести тестирование на более низком уровне. Из первого теста я уже знаю, как работает большая часть «клея» между уровнями. Нет необходимости тратиться еще на один тест E2E, если я могу проверить следующий сценарий на более низком уровне и обойтись намного меньшими затратами за практически такой же прирост доверия.

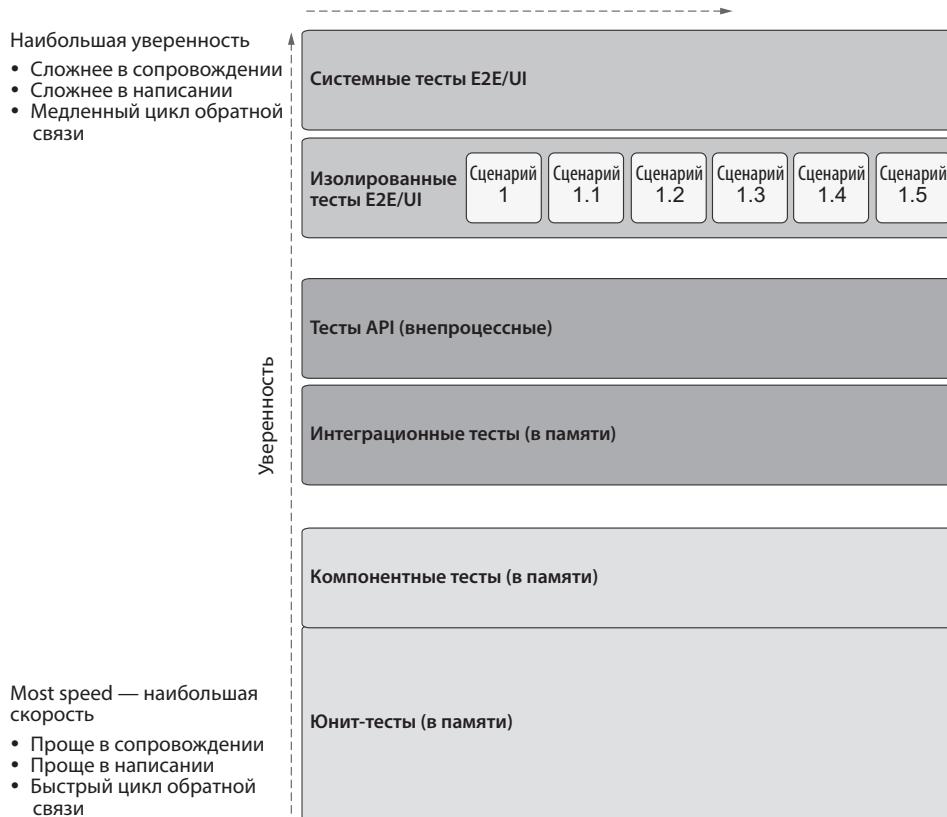


Рис. 10.2. Антипаттерн «только сквозные тесты»

Шаманы сборки

С тестами E2E вы не только получаете убывающую пользу, но и создаете новое узкое место в работе организации. Так как высокоуровневые тесты часто ненадежны, они могут не проходить по многим причинам, в том числе и не имеющим отношения к тесту. Тогда специальным людям в организации (обычно руководителям QA) приходится подолгу сидеть и анализировать каждый непропущенный тест, искать причину и определять, действительно ли это проблема или какой-то пустяк.

Я называю этих несчастных *шаманами сборки*. Когда сборка имеет красный статус, что происходит большую часть времени, именно «шаманы сборки» должны проанализировать данные и со знанием дела сказать после многочасовых разбирательств: «Да, кажется красным, но на самом деле зеленое».

Обычно организация загоняет «шаманов сборки» в угол и требует, чтобы они объявили сборку зеленой, потому что «нам очень нужно выпустить эту версию». Они решают судьбу выпуска, а это неблагодарная, напряженная, часто утомительная и однообразная работа. «Шаманы» обычно выгорают за год-два, а потом сбегают в другую организацию, где продолжают заниматься той же неблагодарной работой. Если вам часто приходится встречаться с «шаманами сборки», значит, в вашей компании используется антипаттерн с большим количеством высокоуровневых тестов E2E.

Автоматизация принятия решений

С этим хаосом можно разобраться: для этого нужно создавать и культивировать надежные, автоматизированные конвейеры тестирования, которые могут автоматически определять, имеет сборка зеленый статус или нет даже при наличии ненадежных тестов. Компания Netflix вела открытый блог про то, как создать собственный инструмент оценки статистической успешности сборки, чтобы ее можно было автоматически утвердить для полноценного развертывания (<http://mng.bz/BAA1>). Такой путь возможен, но для реализации подобного конвейера необходимы время и наличие соответствующей технической культуры. Я пишу о подобных конвейерах в своем блоге по адресу <https://pipelinedriven.org>.

Шаблон мышления «после нас хоть потоп»

Еще одна причина, по которой существование только тестов E2E вредит организации, — в том, что сопровождением и отслеживанием этих тестов занимаются люди из отдела контроля качества (QA). Это означает, что разработчики могут не интересоваться и даже не знать результатов этих сборок, и они не тратят свое время на исправление и обслуживание тестов. Они не отвечают за это.

Отношение к делу по принципу «после нас хоть потоп» способно породить значительные проблемы с передачей информации и снижением качества продукта, потому что одна часть организации не испытывает на себе последствия своих действий, а другая часть страдает от этих последствий, не имея возможности влиять на источник проблемы. Стоит ли удивляться, что во многих компаниях разработчики не ладят с персоналом QA? Система часто устроена так, чтобы они стали смертельными врагами, а не коллегами.

Когда встречается этот антипаттерн

Некоторые причины, по которым может возникать такой антипаттерн.

- *Разделение обязанностей* — во многих организациях существуют отделы разработки и QA с раздельными конвейерами (автоматизированными заданиями сборки и дашбордами). Когда отдел QA использует собственный конвейер, он с большей вероятностью будет писать тесты одного типа. Кроме того, отдел QA склонен писать только определенную разновидность тестов —

ту, к которой они привыкли и которую от них ожидают увидеть (иногда на основании политики компании).

- *Шаблон мышления «если работает, не трогай»* — группа может начать с тестов E2E и решить, что результаты ей нравятся. Она продолжает добавлять новые похожие тесты, потому что уже умеют это делать и тесты оказались полезными. Когда выполнение тестов начинает занимать слишком много времени, менять направление уже поздно (этот момент связан со следующим пунктом).
- *Иллюзия невозвратных издержек* — «У нас много тестов такого типа, и, если мы изменим их или заменим тестами более низкого уровня, это будет означать, что мы попусту потратили время и усилия на тесты, которые будут удалены». Это иллюзия, потому что сопровождение, отладка и анализ сбоев в этих тестах обойдутся очень дорого в пересчете на рабочее время. Если на то пошло, дешевле удалить эти тесты (с сохранением нескольких базовых сценариев) и сэкономить время.

Стоит ли полностью избегать тестов E2E?

Нет, избегать тестов E2E не надо. У них есть свои сильные стороны: в частности, они дают вам *уверенность* в том, что приложение работает. Это совершенно иной уровень доверия по сравнению с юнит-тестами, потому что проверяется интеграция всей системы, со всеми ее подсистемами и компонентами, с точки зрения пользователя. Если эти тесты проходят, вы испытываете огромное облегчение от того, что основные сценарии, с которыми должны столкнуться ваши пользователи, действительно работают.

Так что не избегайте тестов E2E. Вместо этого я настоятельно рекомендую свести их количество к *минимуму*. О том, что считать таким минимумом, мы поговорим в разделе 10.3.3.

10.2.2. Антипаттерн «только низкоуровневые тесты»

Другая крайность при построении стратегии тестирования — использование только низкоуровневых тестов. Юнит-тесты предоставляют быструю обратную связь, но они не обеспечивают полной уверенности в том, что приложение работает как одна интегрированная единица работы (рис. 10.3).

В этом антипаттерне автоматизированные тесты организации в основном или исключительно состоят из низкоуровневых тестов — таких, как компонентные или юнит-тесты. Могут присутствовать некие подобия интеграционных тестов, но тестов E2E нет вообще.

Самая большая проблема заключается в том, что такие тесты не дают достаточной уверенности в том, что ваше приложение работает. Это означает, что люди будут запускать тесты, а затем продолжать ручную отладку и тестирование,

пока не убеждатся, что продукт готов к релизу. Низкоуровневых тестов может быть достаточно только в одном случае: если вы выпускаете библиотеку кода, которая должна использоваться так, как в ваших юнит-тестах. Да, такие тесты выполняются быстро, но вы все равно будете тратить много времени на ручное тестирование и верификацию.

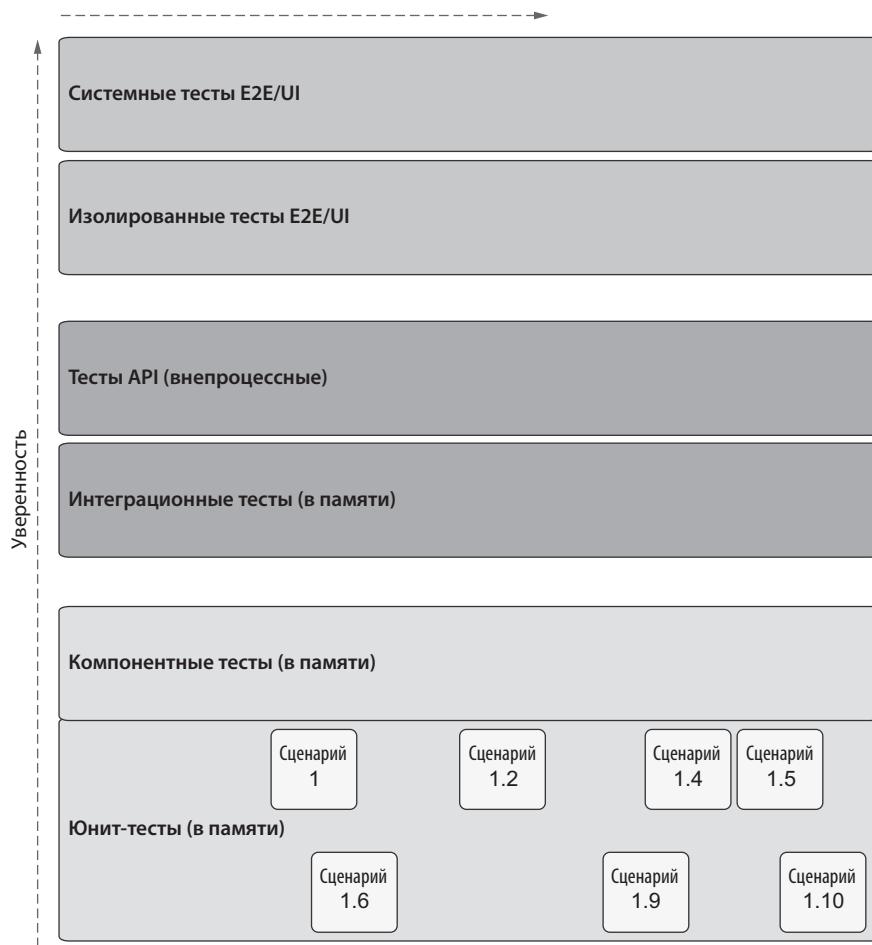


Рис. 10.3. Антипаттерн «только низкоуровневые тесты»

Этот антипаттерн часто возникает, если разработчики пишут только низкоуровневые тесты, не чувствуют себя уверенно в написании высокоуровневых либо если они ожидают, что такие тесты будут написаны в отделе QA.

Означает ли это, что юнит-тестов надо избегать? Очевидно, нет. Но я настоятельно рекомендую создавать *не только* юнит-, но и высокоуровневые тесты. Мы обсудим эти рекомендации в разделе 10.3.

10.2.3. Рассоединение низкоуровневых и высокоуровневых тестов

На первый взгляд эта идея выглядит рационально, но на самом деле это не так. Происходящее выглядит примерно так, как показано на рис. 10.4.

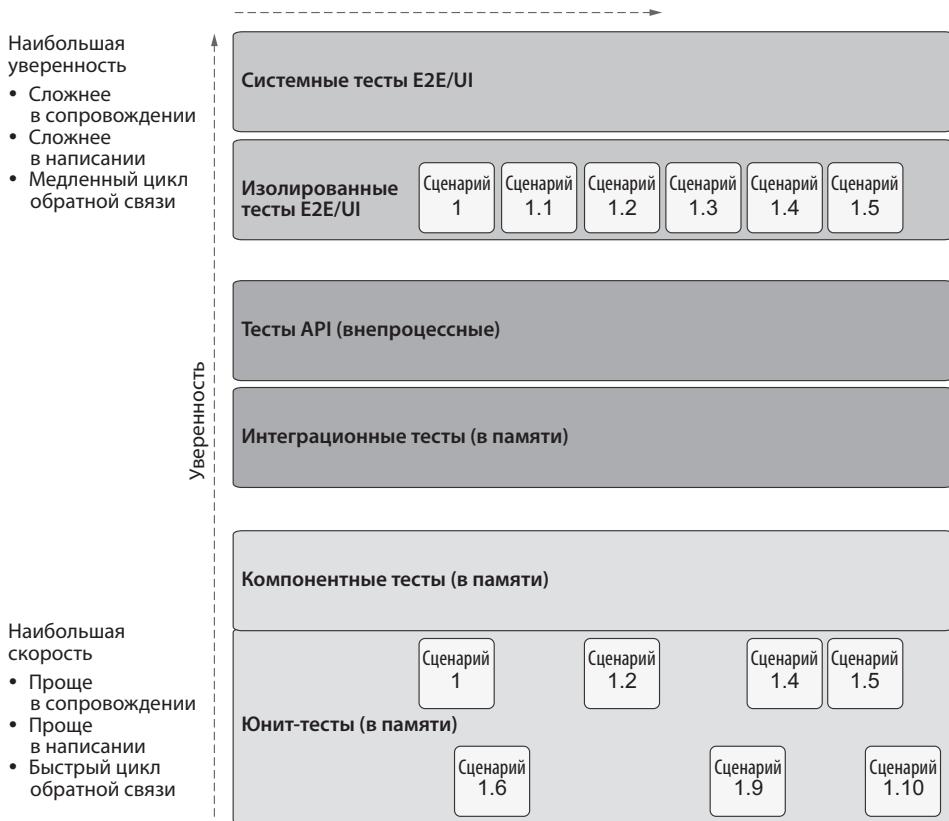


Рис. 10.4. Антипаттерн «рассоединение низкоуровневых и высокоуровневых тестов»

Да, желательно иметь как низкоуровневые тесты (для скорости), так и высокоуровневые (для уверенности). Но если вы заметили следующие признаки

в своей организации, то, скорее всего, вы столкнулись с упомянутым антипаттерном:

- Многие из тестов повторяются на нескольких уровнях.
- Низкоуровневые и высокоуровневые тесты пишутся разными людьми. Это означает, что они не обращают внимания на результаты тестов друг друга и, скорее всего, используют разные конвейеры для выполнения тестов разных типов. Когда один конвейер имеет красный статус, другая группа может даже не знать о том, что эти тесты не проходят, и не обращать на это внимания.
- Вы получаете худшие свойства обеих сторон: на верхнем уровне вы страдаете от долгого времени тестирования, сложностей в сопровождении, «шаманов сборки» и ненадежности; на нижнем уровне — от нехватки уверенности в достоверности результатов. А так как уровень коммуникаций недостаточен, вы не пользуетесь выигрышем от скорости низкоуровневых тестов, потому что они все равно повторяются на верхнем уровне. Также вы теряете уверенность в высокоуровневых тестах из-за ненадежности, обусловленной их большим количеством.

Этот антипаттерн часто встречается, когда тестами и разработкой занимаются разные подразделения с разными целями и метриками, а также разными заданиями и конвейерами, разрешениями и даже репозиториями кода. Чем больше компания, тем выше вероятность того, что это произойдет.

10.3. РЕЦЕПТЫ ТЕСТИРОВАНИЯ КАК СТРАТЕГИЯ

Для достижения сбалансированности видов тестов, используемых организациями, я рекомендую использовать *рецепты тестирования*. Идея заключается в том, чтобы иметь неформальный план относительно того, как должна тестироваться та или иная функциональность. План должен включать не только основной сценарий (также называемый *счастливым путем*), но и все его важные вариации (также называемые *граничными случаями*), как показано на рис. 10.5. Подробное описание теста дает четкую картину того, какой уровень тестирования будет наиболее подходящим для каждого сценария.

10.3.1. Как написать рецепт тестирования

Лучше поручить создание рецепта тестирования как минимум двум людям, и желательно, чтобы у одного была точка зрения разработчика, а у другого — точка зрения тестировщика. Если в компании нет отдела тестирования, хватит двух разработчиков, один из которых может быть более опытным. Отображение каждого сценария на конкретный уровень в иерархии тестирования может быть чрезвычайно субъективной задачей, так что две пары глаз помогут контролировать неявные допущения друг друга.

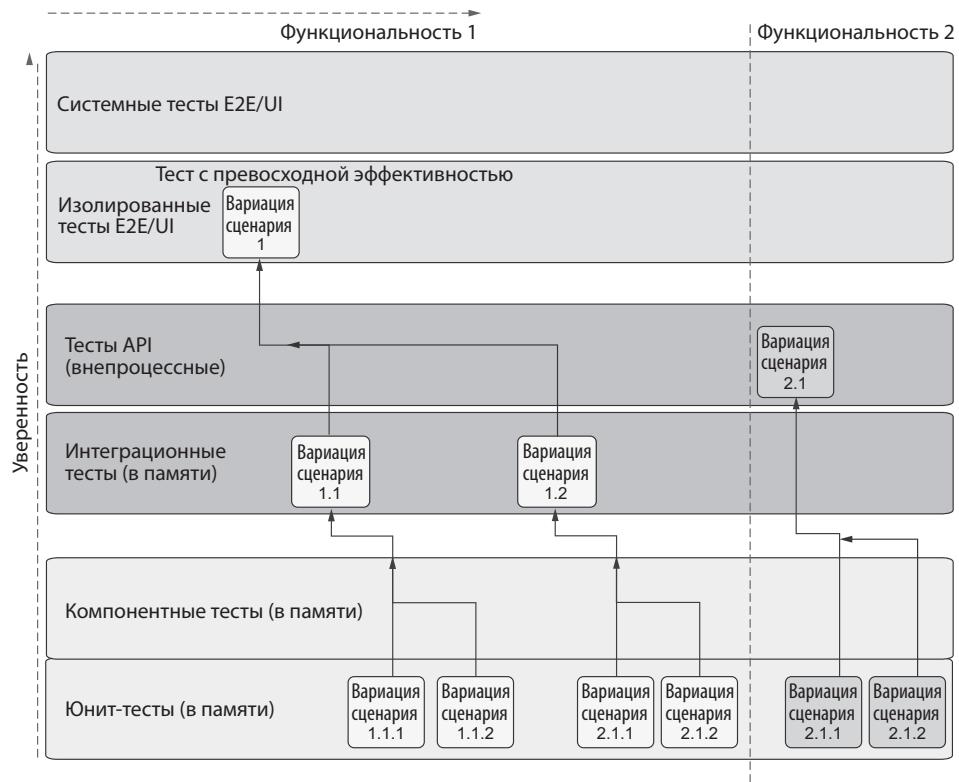


Рис. 10.5. Рецепт тестирования представляет собой план, описывающий, на каком уровне должна тестируться та или иная функциональность

Сами рецепты могут храниться в виде дополнительного текста в списке TODO или как часть описания задачи в таблице отслеживания. Отдельные инструменты для планирования тестов не нужны.

Рецепты тестирования лучше всего создавать непосредственно перед началом работы над функциональностью. Тогда рецепт становится частью определения ее «готовности», то есть работа не будет считаться завершенной, пока не будут проходить все тесты из рецепта.

Конечно, рецепт может изменяться с течением времени. Команда может добавлять или удалять из него отдельные сценарии. Рецепт — не жесткая схема, а непрерывный рабочий процесс, как и все остальное в области разработки.

Рецепт тестирования представляет собой список сценариев, который дает его создателям «неплохую уверенность» в том, что функциональность работает. На практике я предпочитаю выдерживать соотношение от 1:5 до 1:10 между разными уровнями тестов. На каждый высокоуровневый тест E2E у меня могут

быть 5 тестов более низкого уровня. Или можно смотреть снизу вверх. Допустим, у вас 100 юнит-тестов. Обычно на это количество вам потребуется не более 10 интеграционных тестов и 1 тест E2E.

Впрочем, не стоит относиться к рецептам тестирования как к некой формальности. Рецепт тестирования — не жесткое обязательство или список тестовых случаев для программы — планировщика тестов. Не используйте его как открытый отчет, пользовательскую историю или любую другую форму обещаний для стейкхолдеров проекта. По сути, рецепт представляет собой простой список из 5–20 строк текста с описанием несложных сценариев, которые должны тестироваться в автоматизированном режиме, с указанием уровней тестирования. Список может изменяться, пополняться новыми элементами или сокращаться. Считайте его своего рода комментарием. Я обычно размещаю его прямо в пользовательской истории или описании функциональности в Jira или другой подходящей программе.

Пример того, как это может выглядеть:

Рецепт тестирования функциональности пользовательских профилей

E2E – Вход, переход к экрану профиля, обновление адреса электронной почты, выход, вход с новым адресом, проверка обновления экрана профиля

API – Вызов UpdateProfile API с более сложными данными

Юнит-тест – Проверка логики обновления профиля с некорректным адресом

Юнит-тест – Логика обновления профиля с тем же адресом

Юнит-тест – СерIALIZАЦИЯ/ДЕСЕРИАЛИЗАЦИЯ профиля

10.3.2. Когда писать и использовать рецепты тестирования?

Непосредственно перед тем как переходить к программированию функциональности или пользовательской истории, пообщайтесь с другим разработчиком и попытайтесь проработать различные сценарии, которые необходимо протестировать. Обсудите, на каком уровне должен тестироваться сценарий. Обычно такая встреча занимает от 5 до 15 минут, и после нее начинается кодинг, включая написание тестов. (Если вы занимаетесь TDD, начните с тестов.)

В организациях с автоматизацией тестирования или ролями QA разработчик пишет низкоуровневые тесты, а персонал QA направляет усилия на создание высокоуровневых тестов в ходе работы над реализацией функциональности. Оба работают одновременно. Один не ждет, пока другой завершит свою работу, прежде чем писать свои тесты.

Если вы используете переключатели функциональности, также надо проверить, что при отключении функциональности ее тесты больше не выполняются.

10.3.3. Правила составления рецептов тестирования

При написании рецепта необходимо соблюдать несколько правил.

- *Скорость* — отдавайте предпочтение тестам на более низких уровнях (кроме случаев, когда получить уверенность в том, что функциональность работает, можно только при помощи высокоуровневого теста).
- *Уверенность* — рецепт готов, когда вы можете сказать себе: «Если все эти тесты пройдут, можно с уверенностью утверждать, что эта функциональность работает». Если вы не можете так сказать, напишите новые сценарии, с которыми это будет возможно.
- *Пересмотр* — не бойтесь добавлять или удалять тесты из списка в процессе кодинга. Главное — не забудьте сообщить об этом второму человеку, с которым вы работаете над рецептом.
- *Своевременность* — напишите рецепт непосредственно перед тем, как начинать кодинг, как только вы узнаете, кто будет этим заниматься.
- *Работа в паре* — не пишите его в одиночку, если это возможно. Люди мыслят по-разному, поэтому важно обсуждать сценарии и перенимать друг у друга идеи и образ мыслей.
- *Не повторяйтесь (из другой функциональности)* — если сценарий уже покрывается существующим тестом (возможно, тестом E2E для предыдущей функциональности), нет необходимости повторять сценарий на этом уровне.
- *Не повторяйтесь (с других уровней)* — постарайтесь не повторять один сценарий на разных уровнях. Если вы проверяете успешный вход на уровне E2E, то низкоуровневые тесты должны проверять только вариации этого сценария (вход с разными провайдерами, неуспешный результат входа и т. д.).
- *Больше, быстрее* — хорошее практическое правило рекомендует выдерживать соотношение не менее 1:5 между уровнями (на один тест E2E пишутся пять и более низкоуровневых тестов).
- *Прагматизм* — не считайте себя обязанным написать для заданной функциональности тесты на всех уровнях. Некоторые аспекты или пользовательские истории могут требовать только юнит-тестов, другие — только тестов API или E2E. Основная идея заключается в том, что при прохождении всех сценариев в рецепте вы должны почувствовать уверенность в работоспособности системы независимо от того, на каком уровне она тестируется. Если это не так, перемещайте сценарии на другие уровни, пока не достигнете большей уверенности без значительного ущерба для скорости и простоты в сопровождении.

Соблюдая эти правила, вы сможете пользоваться преимуществами быстрой обратной связи: большинство ваших тестов будут низкоуровневыми, но при этом без ущерба для достоверности, потому что самые важные сценарии все еще будут покрываться высокоуровневыми тестами. Стратегия рецептов тестирования

также позволяет избежать большей части повторений между тестами за счет размещения вариаций сценариев на уровнях ниже основного сценария. Наконец, если персонал QA тоже заинтересован в написании рецептов тестирования, вы формируете новый канал связи между людьми в вашей организации, что способствует достижению взаимопонимания в вашем проекте.

10.4. УПРАВЛЕНИЕ КОНВЕЙЕРОМ ПОСТАВКИ ПО

Как насчет тестов производительности? Тестов безопасности? Нагрузочных тестов? Как насчет множества других тестов, выполнение которых может занять целую вечность? Где и когда они должны запускаться? К какому уровню они относятся? Должны ли они быть частью автоматизированного конвейера?

Многие организации запускают эти тесты в составе интеграционного автоматизированного конвейера, выполняемого для каждого выпуска или запроса на включение изменений. Тем не менее это создает значительную задержку в обратной связи, и эта обратная связь часто оказывается «сбойной», хотя для таких типов тестов этот сбой не влияет на выпуск.

Эти виды тестов можно разделить на две основные группы.

- *Тесты, блокирующие поставку*, — эти тесты определяют решение по поводу изменения, планируемого к запуску и развертыванию. Юнит-тесты, E2E, системные тесты и тесты безопасности относятся к этой категории. Их обратная связь имеет бинарную природу: либо они проходят и сообщают о том, что изменение не создало никаких ошибок, либо они не проходят и указывают, что в код необходимо внести исправления перед выпуском.
- *Информационные тесты* — эти тесты создаются для получения информации и непрерывного мониторинга метрик KPI (Key Performance Indicator, ключевые показатели результата деятельности). К этой категории относятся анализ кода и сканирование сложности, тестирование производительности при высокой нагрузке и другие нефункциональные тесты, предоставляющие небинарную обратную связь. Если тесты не проходят, то в следующие спринты можно будет добавить новые рабочие задачи, но это не повлияет на выпуск продукта.

10.4.1. Конвейеры поставки и информационные конвейеры

Мы не хотим, чтобы информационные тесты отнимали ценнее время обратной связи у процесса поставки, поэтому конвейеры также делятся на два типа.

- *Конвейер поставки* — используется для блокирующих тестов. Когда конвейер имеет зеленый статус, мы можем быть уверены в том, что код можно

автоматически передавать для публикации. Тесты в этом конвейере должны предоставлять относительно быструю обратную связь.

- *Информационный конвейер* — используется для информационных тестов. Этот конвейер работает параллельно с конвейером поставки, но работает непрерывно и не учитывается как критерий выпуска. Так как ожидать от него обратную связь необязательно, тесты в этом конвейере могут занимать много времени. Если будут обнаружены ошибки, они станут новыми рабочими задачами в следующих спринтах команды, но выпуск продукта они не блокируют.

На рис. 10.6 представлены признаки этих двух видов конвейеров.

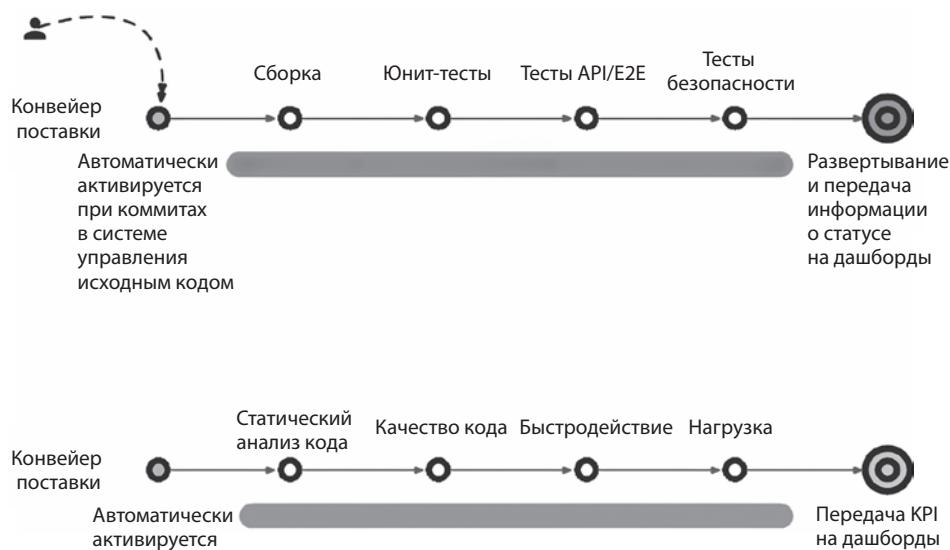


Рис. 10.6. Конвейеры поставки и информационные конвейеры

Задача конвейера поставки — предоставить проверку «да — нет», с которой при зеленом статусе наш код можно развертывать (может быть, даже в продакшн). Задача информационного конвейера — предоставить цели рефакторинга для команды (такие, как решение проблем с чрезмерной сложностью кода). Он также показывает, были ли усилия по рефакторингу эффективными по прошествии времени. Информационный конвейер ничего не развертывает, разве что для целей выполнения специализированных тестов или анализа кода и его различных метрик KPI. Результатом его работы являются числа на дашборде.

Скорость является важным фактором для повышения степени вовлеченности команд, и разбиение тестов на конвейеры поставки и получения информации — еще один полезный прием, который стоит держать в своем арсенале.

10.4.2. Параллелизация уровней тестирования

Так как быстрая обратная связь очень важна, существует распространенный паттерн, который можно (и нужно) применять во многих сценариях. Речь идет о параллельном выполнении разных уровней тестов для ускорения обратной связи от конвейера (рис. 10.7). Вы даже можете использовать параллельные среды, которые создаются динамически и уничтожаются в конце теста.

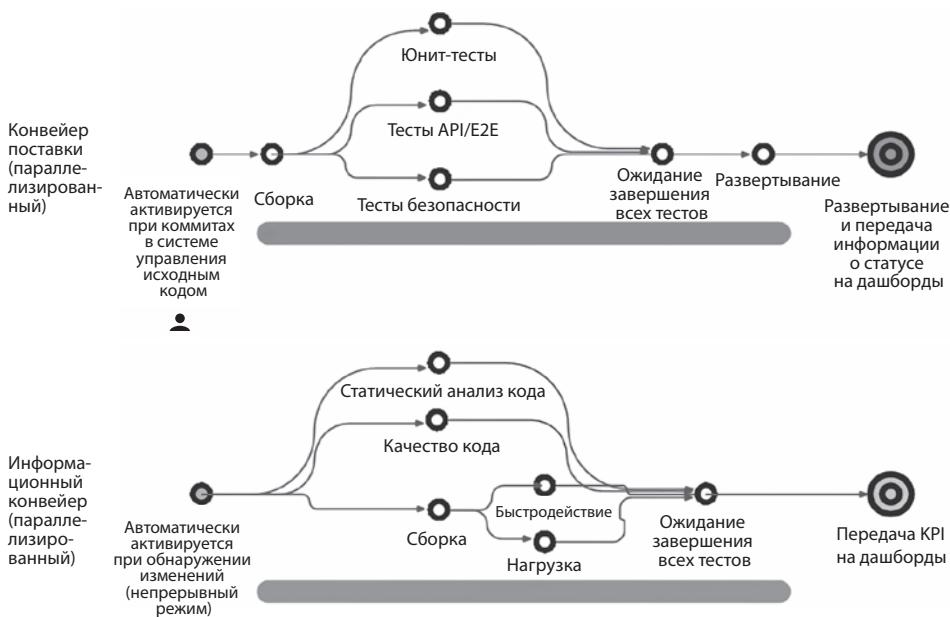


Рис. 10.7. Для ускорения получения результата конвейеры (и даже отдельные стадии конвейеров) могут выполняться параллельно

Такой подход сильно выигрывает от доступности динамических сред. Расходование денег на среды и автоматизацию параллельного тестирования почти всегда оказывается намного эффективнее затрат на дополнительный персонал, занимающийся ручным тестированием, или длительное ожидание обратной связи из-за того, что среда используется в данный момент.

Ручное тестирование не работает в долгосрочной перспективе, потому что объем ручной работы только растет со временем, а ее результаты становятся все более ненадежными и подверженными ошибкам. В то же время простое ожидание обратной связи от конвейера приводит к огромным потерям времени для всех. Время ожидания, умноженное на количество ожидающих людей и количество сборок за день, приводит к ежемесячным затратам, которые значительно превышают расходы на динамические среды и автоматизацию. Создайте файл

Excel и покажите своему руководству результаты вычислений по этим простым формулам.

Параллелизировать можно не только отдельные стадии конвейера; можно пойти еще дальше и организовать параллельное выполнение отдельных тестов. Например, если вы столкнулись с большим количеством тестов E2E, их тоже можно разбить на параллельные наборы тестов. Тем самым вы заметно ускорите цикл обратной связи.

Не используйте ночные сборки

Конвейер поставки лучше выполнять при каждом коммите кода, а не в конкретное время. Запуск тестов при каждом изменении в коде предоставляет более детализированную и быструю обратную связь, чем ночная сборка, в которой бессистемно накапливаются все изменения за предыдущий день. Но если по какой-то причине вам абсолютно необходимо выполнять свой конвейер в определенное время, по крайней мере, пусть он работает непрерывно, а не раз в день.

Если сборка в конвейере поставки занимает много времени, не ждите, пока он будет запущен по графику или по магическому включателю. Представьте, что вам как разработчику придется ждать до завтра, чтобы узнать, не сломали ли вы что-нибудь. При непрерывном выполнении тестов вам все равно придется ждать, но, по крайней мере, это будет всего пара часов вместо целого дня. Разве это не эффективнее?

Кроме того, не проводите сборку только по требованию. Цикл обратной связи ускорится, если вы будете запускать сборку автоматически сразу же после завершения предыдущей (конечно, при появлении изменений в коде по сравнению с предыдущей сборкой).

ИТОГИ

- Существует несколько уровней тестирования: юнит-тесты, компонентные и интеграционные тесты, выполняемые в памяти; и тесты API, изолированные сквозные (E2E) и системные E2E тесты с внепроцессным выполнением.
- Каждый тест может оцениваться по пяти критериям: сложность, ненадежность, уверенность при прохождении, простота в сопровождении и скорость выполнения.
- Компонентные и юнит-тесты хороши из-за простоты в сопровождении, скорости выполнения и отсутствия сложности и ненадежности, но оказываются худшими в отношении уверенности, которую они обеспечивают.

Интеграционные тесты и тесты API занимают промежуточное положение по балансу между уверенностью и другими метриками. Тесты E2E находятся на другом конце оси: они обеспечивают наибольшую уверенность, но приходится мириться с усложнением сопровождения, уменьшением скорости, сложностью и ненадежностью.

- Антипаттерн «*только сквозные тесты*» встречается тогда, когда сборка состоит исключительно из E2E-тестов. Прирост полезности от каждого дополнительного теста E2E невелик, тогда как затраты на сопровождение всех тестов одинаковы. Ваши усилия принесут наибольшую отдачу, если у вас будет небольшое количество тестов E2E, покрывающих самую важную функциональность.
- Антипаттерн «*только низкоуровневые тесты*» встречается тогда, когда сборка состоит исключительно из компонентных и юнит-тестов. Низкоуровневые тесты не могут обеспечить достаточной уверенности в том, что функциональность в целом работает, и они должны дополняться тестами более высокого уровня.
- *Рассоединение низкоуровневых и высокоуровневых тестов* тоже считается антипаттерном, потому что это явный признак того, что тесты были написаны двумя группами людей, которые не коммуницировали. Такие тесты часто дублируют друг друга; кроме того, они формируют высокие затраты на сопровождение.
- *Рецепт тестирования* представляет собой простой список от 5 до 20 строк текста, где говорится, какие простые сценарии должны тестироваться автоматически и на каком уровне. Рецепт тестирования должен давать уверенность в том, что при прохождении всех описанных в нем тестов функциональность будет работать так, как предполагалось.
- Разделите конвейер сборки на конвейеры поставки и получения информации. Первый конвейер должен использоваться для тестов, блокирующих поставку, и в случае непрохождения таких тестов выпуск тестируемого кода отменяется. Информационный конвейер используется для тестов, приносящих полезную информацию; он выполняется параллельно с конвейером поставки.
- Параллелизировать можно не только конвейеры, но и отдельные этапы внутри этих конвейеров и даже группы тестов внутри этапов.

11

Интеграция юнит-тестирования в организацию

В ЭТОЙ ГЛАВЕ

- ✓ Как стать инициатором перемен
- ✓ Реализация изменений сверху вниз или снизу вверх
- ✓ Ответы на непростые вопросы о юнит-тестировании

В качестве консультанта я помогал разным компаниям — как крупным, так и мелким — интегрировать непрерывные процессы поставки и различные инженерные практики (такие, как разработка через тестирование и юнит-тестирование) в их организационную культуру. Иногда мои усилия заканчивались неудачей, но у компаний, добивавшихся успеха, было кое-что общее. В организации любого типа изменение привычек имеет больше психологическую, нежели техническую природу. Люди не любят перемен; перемены обычно вызывают опасения, неопределенность и сомнения. Как будет показано в этой главе, для многих людей все это нелегко.

11.1. КАК СТАТЬ ИНИЦИАТОРОМ ИЗМЕНЕНИЙ

Если вы собираетесь стать инициатором изменений в своей организации, прежде всего необходимо принять эту роль. Люди будут рассматривать вас как ответственного за происходящее независимо от того, нравится вам это или нет, и отказываться бесполезно. Более того, если вы будете прятаться от ответственности, дело может принять неожиданный и неприятный оборот.

Когда вы начнете убеждать всех в необходимости изменений или воплощать их на практике, люди станут задавать непростые вопросы о том, что для них важно. Сколько времени мы потеряем на этом? Что это будет означать для меня как для инженера QA? Как узнать, что это работает? Будьте готовы ответить на эти вопросы. Многие ответы обсуждаются в разделе 11.5. Если вам удастся убедить других сотрудников в необходимости изменений до того, как вы начнете их внедрять, это очень сильно поможет вам, когда придется принимать непростые решения и отвечать на вопросы.

Наконец, кто-то должен стоять у руля и следить за тем, чтобы импульс не угас из-за нехватки инициативы. Этим человеком будете вы. Как будет показано в нескольких ближайших разделах, ситуацию можно поддерживать в движении.

11.1.1. Подготовьтесь к непростым вопросам

Изучите матчасть. Прочитайте вопросы и ответы в конце главы, ознакомьтесь с соответствующими источниками. Читайте форумы, списки, рассылки и блоги, консультируйтесь со своими коллегами. Если вы сможете ответить на собственные непростые вопросы, весьма вероятно, что и на чужие тоже сможете.

11.1.2. Убедите окружающих: сторонников и скептиков

Мало что приводит к такому сильному отчуждению, как решение идти против всех. Если вы единственный, кто думает, что вы занимаетесь полезным делом, у окружающих не будет особых стимулов помочь вам с реализацией того, за что вы агитируете. Подумайте, кто может поддержать ваши старания или, наоборот, противодействовать им? Назовем эти группы «сторонниками» и «скептиками».

Сторонники

Прежде чем убеждать окружающих в необходимости изменений, определите людей, которые, по вашему мнению, с наибольшей вероятностью помогут вам в вашей миссии. Они станут вашими *сторонниками*. Как правило, это ранние последователи или люди достаточно широких взглядов, желающие опробовать то, за что вы выступаете. Они могут быть уже наполовину убеждены, но им нужен

импульс для запуска изменений. А может быть, они уже пытались внедрить эти изменения, но у них не получилось.

Обратитесь к ним до того, как вы обратитесь к другим, и поинтересуйтесь их мнением относительно того, что собираетесь сделать. Они могут подсказать что-то такое, чего вы не учли, в том числе:

- команды, которые могут быть хорошими кандидатами для начала внедрения;
- места, в которых люди более открыты к таким изменениям;
- чего (и кого) стоит остерегаться в вашей миссии.

Обращаясь к ним, вы способствуете тому, чтобы они стали частью процесса. Люди, которые ощущают свою сопричастность, обычно стараются способствовать успеху дела. Сделайте их своими союзниками: спросите их, смогут ли они помочь вам и стать теми, к кому люди придут со своими вопросами. Подготовьте их к таким событиям.

Скептики

Затем определите *скептиков*. Это те люди в организации, которые с наибольшей вероятностью будут противодействовать вносимым изменениям. Например, руководитель может воспротивиться добавлению юнит-тестов, утверждая, что они заметно увеличивают время разработки и повышают объем кода, нуждающегося в сопровождении. Сделайте скептиков участниками процесса, а не его блокировщиками: предоставьте им активную роль (по крайней мере, тем, кто желает и способен на это).

Люди могут сопротивляться изменениям по разным причинам. Ответы на некоторые возможные возражения приведены в разделе 11.4. Кто-то беспокоится, что его рабочее место могут сократить; другие комфортно себя чувствуют и не хотят этого менять.

Обращение к потенциальным скептикам с подробным описанием того, что они могли бы сделать лучше, часто оказывается неконструктивным — я узнал об этом на собственном горьком опыте. Людям не нравится, когда им говорят, что они были не слишком эффективны.

Вместо этого предложите скептикам помочь вам в процессе — например, взять на себя ответственность за определение стандартов кодинга для юнит-тестов или проводить ежедневные код-ревью и ревью тестов с коллегами. Или сделайте их частью команды, которая выбирает учебные материалы либо внешних консультантов. Вы предоставите им новую обязанность, отчего они будут чувствовать, что от них что-то зависит и они играют важную роль в организации. Эти люди должны участвовать в изменениях, иначе они почти наверняка будут сопротивляться им.

11.1.3. Определите возможные отправные точки

Определите, где в организации можно начать реализовывать изменения. Самые успешные реализации идут по стабильному маршруту. Начните с пилотного проекта в небольшой команде и посмотрите, что получится. Если все идет нормально, распространяйте опыт на другие команды и другие проекты.

Несколько рекомендаций, которые немного упростят вашу задачу:

- выбирайте небольшие команды;
- создайте подкоманды;
- учитывайте, насколько для ваших целей годится проект, над которым они работают;
- используйте результаты код-ревью и ревью тестов как учебные инструменты.

Эти советы помогут вам преодолеть немалую часть пути во враждебной (в основном) среде.

Выбирайте небольшие команды

Определить возможные команды для начала работы, как правило, несложно. В общем случае вам нужна небольшая команда, работающая над проектом без особой шумихи и с низким риском. Если риск минимален, вам будет проще убедить людей опробовать предложенные изменения.

Одна из проблем заключается в том, что в команде должны быть участники, готовые изменить свой подход к работе и освоить новые навыки. Как ни странно, люди с меньшим опытом работы обычно более открыты к изменениям, а более опытные не склонны выходить из накатанной колеи. Если вы найдете команду с опытным лидером, открытым для изменений, в которой работает ряд менее опытных разработчиков, скорее всего, сопротивление будет минимальным. Обратитесь к ним и поинтересуйтесь их мнением о проведении пилотного проекта. Они скажут вам, есть ли смысл начинать изменения в их команде.

Создайте подкоманды

Другая потенциальная возможность для пилотного проекта — формирование подкоманды внутри существующей команды. Почти в каждой команде имеется компонент «черная дыра», который нуждается в сопровождении, и, хотя многое делается правильно, он также содержит много ошибок. Добавление функциональности в такой компонент — непростая задача, и подобные неприятности могут подтолкнуть людей к экспериментам с пилотным проектом.

Учитывайте пригодность проекта для ваших целей

Выбирая пилотный проект, не пытайтесь откусить больше, чем сможете прожевать. Реализация более сложных проектов требует большего опыта, так что вам стоит иметь как минимум два варианта проектов: сложный и простой, чтобы вы могли выбрать между ними.

Используйте результаты код-ревью и ревью тестов как учебные инструменты

Если вы являетесь техлидом небольшой команды (до восьми человек), одним из лучших способов обучения будет проведение код-ревью, также включающего ревью тестов. Идея в том, чтобы в ходе ревью вы учили людей, на что следует обращать внимание в коде и как следует подходить к написанию тестов или TDD. Несколько советов:

- проводите ревью лично, а не через программы дистанционного общения. Личное общение позволяет передавать намного больше информации на невербальном уровне, поэтому обучение происходит быстрее и эффективнее;
- первую пару недель рецензируйте каждую строку сохраненного кода. Это поможет избежать проблем типа «мы не думали, что этот код нуждается в ревью»;
- добавьте третьего участника в сеансы рецензирования — того, кто будет сидеть рядом и учиться тому, как проходит код-ревью. Это позволит ему позднее проводить ревью самостоятельно и учить других, чтобы вы не стали узким местом команды как единственный человек, способный делать это. Идея в том, чтобы развивать способность других проводить код-ревью и принимать на себя большую ответственности.

Если вы захотите больше узнать об этой методике, прочтите статью «What Should a Good Code Review Look and Feel Like?» в моем блоге для техлидов по адресу <https://5whys.com/blog/what-should-a-good-code-review-look-and-feel-like.html>.

11.2. ПУТИ К УСПЕХУ

Организация или команда может запустить процесс изменений двумя основными способами: снизу вверх или сверху вниз (а иногда в обоих направлениях сразу). Эти два направления очень сильно отличаются, как вы вскоре увидите, и любое из них может стать правильным подходом для вашей команды или компании. Единственно правильного способа не существует.

По ходу дела вы должны научиться убеждать руководство в том, что *ваши и их усилия* должны совпадать или что иногда будет разумно подключить кого-то извне для содействия. Видимый прогресс важен, как и определение четких целей,

которые могут измеряться объективно. Выявление и предотвращение препятствий также должны занимать одну из ведущих позиций вашего списка задач. Есть много областей, в которых вы сможете приложить свои усилия, и нужно выбрать из них правильные.

11.2.1. Партизанская реализация (снизу вверх)

Суть партизанской реализации заключается в том, что вы начинаете работу с командой, добиваетесь результатов и только потом убеждаете других людей, что ваши практики того стоят. Обычно направляющей силой для партизанской реализации становится команда, которой надоело делать все заранее предписаным образом. Они решают действовать по-своему; они самостоятельно учатся и воплощают изменения. Когда группа показывает хорошие результаты, другие люди в организации решают понемногу реализовать нечто похожее в своих командах.

В некоторых случаях партизанская реализация становится процессом, который сначала *принимается* разработчиками и только потом руководством. В других случаях это процесс, за который сначала *выступают* разработчики и только потом руководство. В чем различия? В том, что первое может быть воплощено незаметно — так, что высшие силы ничего не знают об этом. Второе делается во взаимодействии с руководством. Вы сами решаете, какой вариант вам лучше подойдет. Иногда что-то изменить можно только скрытно. Избегайте этого варианта, если можете, но если другого способа нет и вы уверены в необходимости изменений — переходите к действиям.

Не воспринимайте это как рекомендацию сделать нечто такое, что может отрицательно сказать на вашей карьере. Разработчики постоянно делают что-нибудь без разрешения: отладка кода, чтение электронной почты, написание комментариев в коде, построение блок-схем и т. д. Все эти задачи решаются разработчиками в ходе их повседневной работы. То же самое можно сказать и о юнит-тестировании. Многие разработчики уже пишут тесты того или иного вида (автоматизированные или нет). Идея в том, чтобы перенаправить время, проведенное за тестами, на нечто, способное принести выигрыши в долгосрочной перспективе.

11.2.2. Привлечение руководства (сверху вниз)

Внедрение изменений сверху вниз обычно начинается одним из двух способов. Руководитель или разработчик запускает процесс и начинает подталкивать всю организацию в этом направлении, шаг за шагом. А может, руководитель среднего звена видит презентацию, читает книгу (например, эту) или узнает от коллег о преимуществах конкретных изменений в подходе к работе. Такой руководитель

обычно запускает процесс, проводя презентацию для участников других команд, или даже внедряет изменения, пользуясь своими полномочиями.

11.2.3. Эксперименты как первый шаг

Это перспективный способ внедрения юнит-тестирования в большой организации (он также может использоваться для других видов преобразований или новых навыков). Объявите эксперимент, который должен продлиться два-три месяца. Эксперимент будет проходить только в рамках одной заранее выбранной команды, а предлагаемые изменения будут применяться только к одному или двум компонентам реального приложения. Убедитесь в том, что эксперимент не создает слишком больших рисков. В случае неудачи компания не должна разориться или потерять важного клиента. Кроме того, эксперимент не может быть бесполезным: он должен приносить реальную пользу, а не просто создавать «игровую площадку» для разработчиков. Это должно быть что-то такое, что вы в конечном счете внедрите в свою кодовую базу и будете использовать в рабочей версии, а не поделка по принципу «написали и забыли».

Слово «эксперимент» показывает, что изменения имеют временную природу, и если что-то не сработает, команда может вернуться к прежнему стилю работы. Кроме того, мероприятие ограничено по времени, чтобы все знали, когда эксперимент завершится.

Такой подход помогает людям спокойнее относиться к большим изменениям, потому что он сокращает риск для организации, численность задействованных участников (и следовательно, количество противников) и количество протестов, связанных с опасениями того, что ситуация изменяется «навсегда».

Еще одна подсказка: когда вы сталкиваетесь с несколькими вариантами проведения эксперимента или получаете возражения, которые должны направить работу в другую сторону, спросите коллег: «С чьей идеей мы хотим поэкспериментировать *в первую очередь*?»

Выполняйте свои обязательства

Будьте готовы к тому, что ваша идея может быть не включена в планы эксперимента. Когда дойдет до дела, вам придется вести работу в соответствии с указаниями руководства, нравится вам это или нет.

Впрочем, участие в экспериментах, предложенных другими людьми, не так плохо: эти эксперименты, как и ваши, ограничены по времени! В лучшем случае другой метод решит те проблемы, которые пытались решить вы; такой эксперимент стоит продолжить. Но если результат вам не понравился, просто вспомните, что это не навсегда, и переходите к следующему эксперименту.

Метрики и эксперименты

Обязательно зарегистрируйте значения базовых метрик до и после эксперимента. Эти метрики должны относиться к тому, что вы пытаетесь изменить: например, сокращение времени ожидания сборки, сокращение времени от начала разработки продукта до запуска в эксплуатацию или сокращение количества ошибок в рабочей версии.

За более подробной информацией о различных метриках, которые могут вам пригодиться, обращайтесь к моему докладу «*Lies, Damned Lies, and Metrics*», опубликованному в моем блоге по адресу <https://pipelinedriven.org/article/video-lies-damned-lies-and-metrics>.

11.2.4. Привлечение внешнего сторонника

Я настоятельно рекомендую привлечь внешнего специалиста, который поможет с изменениями. У внешнего консультанта, помогающего в решении проблем с юнит-тестированием и сопутствующих вопросов, есть преимущества перед сотрудником компании.

- *Свобода слова* — консультант может говорить то, что сотрудники компании не захотят услышать от того, кто здесь работает («С целостностью кода все плохо», «Ваши тесты нечитабельны» и т. д.).
- *Опыт* — у консультанта может быть больше опыта по преодолению сопротивления, он может дать хорошие ответы на непростые вопросы и знать, какие кнопки нужно нажать, чтобы запустить инициативу.
- *Специализация* — для консультанта это его работа. В отличие от других сотрудников компании, у которых есть более насущные дела, чем выступать за изменения (например, программировать), консультант занимается этим все свое время и специализируется на этой задаче.

Я часто видел, как изменения останавливаются на полпути, потому что их инициатор перегружен работой и у него нет времени, которое он мог бы посвятить процессу.

11.2.5. Демонстрация прогресса

Важно следить за тем, чтобы прогресс и состояние изменений были видны окружающим. Развешайте плакаты или доски на стенах в коридорах или кафетериях, где собираются люди. Представленные данные должны относиться к целям, которых вы стараетесь достичь. Вот примеры:

- Количество прошедших или непрощедших тестов в последней ночной сборке.
- Диаграмма с перечнем команд, уже применяющих автоматизированный процесс сборки.

- Разместите диаграмму сгорания Scrum для прогресса итерации или отчет о тестовом покрытии кода (как на рис. 11.1), если вы выбрали эти показатели в качестве целей. (За дополнительной информацией о Scrum обращайтесь по адресу www.controlchaos.com.)
- Разместите свои контактные данные и контактные данные вашей группы поддержки, чтобы кто-то из вас мог отвечать на возникающие вопросы.

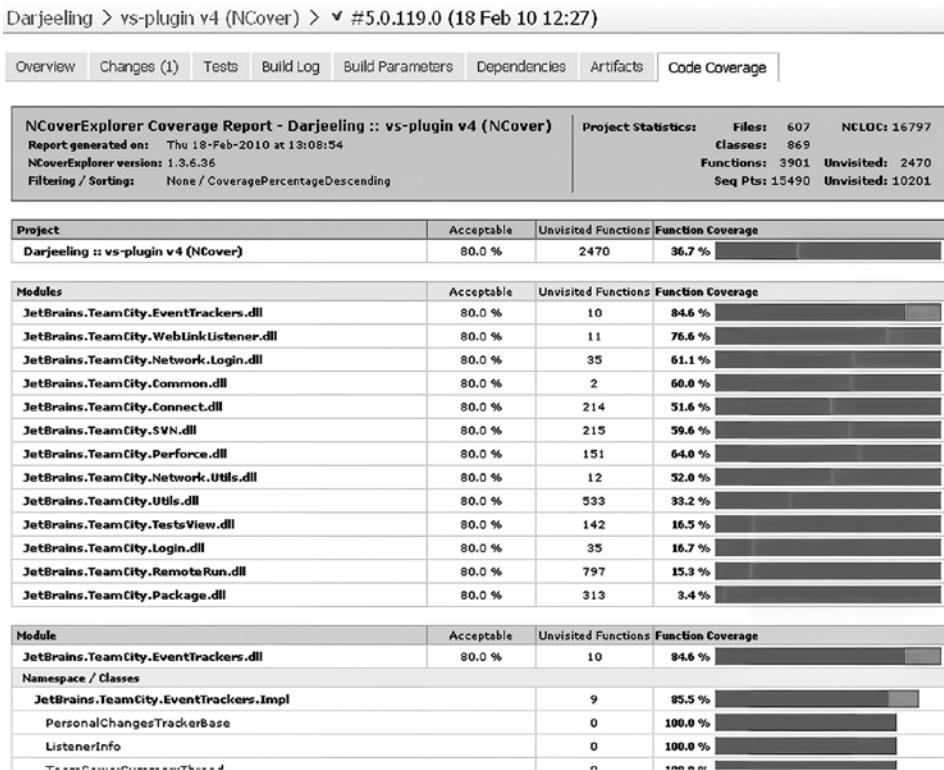


Рис. 11.1. Пример отчета с данными тестового покрытия кода в TeamCity с использованием NCover

- Создайте экран, на котором в постоянном режиме крупным шрифтом будет отображаться информация о статусе сборки, о том, что сейчас работает, а что нет. Разместите его на видном месте, например в коридоре, по которому часто ходят, или на стене помещения для коллективной работы.

Ваша цель — при помощи этих средств соединить две группы:

- *группу, участвующую в изменениях*, — у участников этой группы появится чувство гордости от достижения результата, когда данные (открытые для

всех) будут обновляться. У них возникнут дополнительные стимулы для завершения процесса, потому что он так заметен всем окружающим. Они также смогут сравнивать свои показатели с показателями других групп. Возможно, они будут прилагать больше стараний, зная, что другая группа быстрее реализовала некоторые практики;

- *сотрудников организации, не являющихся частью процесса*, — вы вызываете интерес и любопытство у этих людей, провоцируете разговоры и ажиотаж. Тем самым вы создаете течение, к которому они при желании смогут примкнуть.

11.2.6. Ориентация на конкретные цели, метрики и KPI

Если нет поставленной цели, то трудно объективно оценить успешность затеянного изменения и передать информацию другим. У вас получится неопределенное «нечто», которое может быть легко отменено при первых признаках проблем.

Запаздывающие индикаторы

На уровне организации юнит-тесты обычно являются частью большого набора целей, чаще всего связанных с непрерывной поставкой продукта. Если это можно сказать и о вашем случае, я настоятельно рекомендую использовать четыре основные метрики DevOps.

- *Частота развертывания* — как часто организация успешно выпускает рабочую версию.
- *Срок реализации изменений* — время от поступления запроса на добавление функциональности до выхода рабочего кода. Заметим, что во многих источниках этот показатель неверно определяется как время, необходимое коммиту для преобразования в рабочий код; это время с организационной точки зрения составляет лишь часть пути, проходимого функциональностью. Если измерять от времени коммита, результат получится ближе ко «времени цикла» функциональности. Срок реализации состоит из нескольких «времен цикла».
- *Пропущенные ошибки/частота сбоев* — количество сбоев, обнаруженных в рабочей версии за некоторую единицу измерения (обычно выпуск, развертывание или интервал времени). Также можно использовать процент развертываний, приводящих к сбоям в рабочем коде.
- *Время восстановления работоспособности* — сколько времени требуется организации для восстановления после сбоя в рабочем коде.

Эти четыре характеристики относятся к категории так называемых *запаздывающих индикаторов*, и их очень трудно фальсифицировать (в большинстве мест они достаточно легко измеряются). Они помогают убедиться в том, что мы не обманываем себя относительно результатов эксперимента.

Опережающие индикаторы

Часто нам хотелось бы ускорить получение обратной связи, гарантирующей, что мы движемся в правильном направлении. Здесь на помощь приходят *опережающие индикаторы* — показатели, которые можно контролировать в повседневной работе: покрытие кода, количество тестов, время сборки и т. д. Их проще фальсифицировать, но в сочетании с запаздывающими индикаторами они часто могут предоставить ранние признаки того, что мы движемся в правильном направлении.

На рис. 11.2 изображены пример структуры и некоторые идеи для запаздывающих и опережающих индикаторов, которые вы можете использовать в своей организации. Цветное изображение в высоком разрешении доступно по адресу <https://pipelinedriven.org/article/a-metrics-framework-for-continuous-delivery>.



Рис. 11.2. Пример структуры метрик, используемых для непрерывной поставки

Категории и группы индикаторов

Обычно я разбиваю опережающие индикаторы на две группы:

- *уровень команды* — метрики, которые могут контролироваться отдельной командой;
- *уровень технического руководства* — метрики, требующие сотрудничества между командами или обобщения метрик по нескольким командам.

Я также классифицирую их на основании того, для решения каких задач они будут использоваться.

- *Прогресс* — используются для решения задач видимости и принятия решений по плану.
- *Узкие места и обратная связь* — понятно из названия.
- *Качество* — допущенные ошибки в рабочем коде.
- *Навыки* — метрики для отслеживания постепенного удаления информационных барьеров между командами или внутри команд.
- *Обучение* — как если бы ваша организация занималась преподаванием.

Качественные метрики

Метрики в основном имеют количественную природу (то есть их можно измерить и представить результат в виде числа), но среди них встречаются некоторые качественные — когда вы спрашиваете у людей, что они чувствуют или думают о каком-то вопросе. Вот некоторые из таких метрик, которыми я пользуюсь.

- Насколько вы уверены, что тесты смогут найти и найдут ошибки в коде при их возникновении (от 1 до 5)? Вычислите среднее по ответам, полученным от участников команды или между командами.
- Делает ли код то, что ему положено делать (от 1 до 5)?

Все эти данные, которые можно получить в ходе опросов на ретроспективных встречах, занимают не более пяти минут.

Линии трендов — ваши друзья

Для всех индикаторов, опережающих и запаздывающих, необходимо отслеживать *тренды*, не ограничиваясь «снапшотами» числовых данных. По линиям трендов можно судить, улучшается или ухудшается ситуация.

Не попадайте в ловушку с созданием красивых дашбордов с большими цифрами. Числа, лишенные контекста, сами по себе не хороши и не плохи. Линии трендов показывают, улучшилась ли ваша ситуация за эту неделю по сравнению с предыдущей.

11.2.7. Понимание возможных препятствий

Препятствия будут всегда. Чаще всего они исходят от организационной структуры, некоторые имеют техническую природу. Справиться с техническими проблемами проще, потому что все сводится к поиску правильного решения. Решение организационных проблем требует внимательного и продуманного подхода, а также некоторых знаний по психологии.

Важно не поддаваться настроению временной неудачи, когда очередная итерация идет не так, тесты работают медленнее, чем предполагалось, и т. д. Иногда бывает тяжело двигаться вперед, и вам придется потрудиться хотя бы пару месяцев, чтобы освоиться с новым процессом и сгладить все зазубрины. Убедите руководство не прерывать эксперимент хотя бы три месяца, даже если он пойдет не по плану. Важно получить согласие заранее. Поверьте, вам не захочется бегать и пытаться кого-то убедить в первый же напряженный месяц.

Кроме того, усвойте короткое наблюдение, которым поделился Тим Отtingер (Tim Ottinger) в Twitter (@Tottinge): «Даже если ваши тесты не обнаружат все дефекты, они все равно упростят исправление дефектов, которые не были обнаружены. Это фундаментальная истин».

Итак, мы рассмотрели способы достижения успеха. Теперь рассмотрим некоторые обстоятельства, которые могут привести к неудаче.

11.3. ВОЗМОЖНЫЕ НЕУДАЧИ

В предисловии к этой книге я упоминал об одном проекте с моим участием, который завершился провалом — отчасти из-за того, что юнит-тестирование было реализовано неправильно. Это лишь одна из возможных причин неудачи. В этом разделе рассматриваются другие причины (вместе с той, которая стоила мне потери проекта) и некоторые меры, которые можно предпринять в таких ситуациях.

11.3.1. Недостаточный направляющий импульс

В тех местах, где я видел неудачные попытки внедрения изменений, самым мощным фактором была нехватка направляющего импульса. За последовательную работу по внедрению изменений приходится расплачиваться. Вам придется отвлечься от своей обычной работы, чтобы обучать других, помогать им и проявить свою политическую мудрость в борьбе за необходимые преобразования. Вы должны быть готовы пожертвовать своим временем для выполнения этих задач, иначе изменения не произойдут. Привлечение внешнего специалиста, как упоминалось в разделе 11.2.4, поможет вашей миссии по формированию последовательного направляющего импульса.

11.3.2. Недостаток политической поддержки

Если начальство открыто запрещает вам вносить изменения, сделать с этим почти ничего нельзя — разве что попытаться переубедить и заставить увидеть то, что видите вы. Но иногда недостаток поддержки выглядит не столь очевидно, и непросто даже понять, что вы сталкиваетесь с противодействием.

Например, вам могут сказать: «Конечно, действуй, реализуй эти тесты. Мы накинем на это еще 10 % от твоего времени». Любые значения ниже 30 % нереалистичны для первых усилий по юнит-тестированию. Это один из способов, которым руководитель может попытаться остановить новую тенденцию, — перекрыть ей кислород.

Необходимо понять, что вы сталкиваетесь с противодействием, но, когда вы знаете, на что обращать внимание, разобраться в ситуации становится проще. Если вы им скажете, что такие лимиты нереалистичны, вам ответят: «Тогда не делай».

11.3.3. Ситуативные реализации и первые впечатления

Если вы планируете реализовать юнит-тестирование, не умея писать хорошие юнит-тесты, окажите себе услугу: пригласите кого-нибудь, кто обладает опытом и следует хорошим практикам (вроде описанных в этой книге).

Я видел, как разработчики пускались во все тяжкие без правильного понимания того, что делать и с чего начинать. Мне не хотелось бы быть на их месте. Мало того, что вам понадобится очень много времени, чтобы научиться вносить изменения, подходящие для вашей ситуации; вам также станут меньше доверять из-за того, что вы начали с плохой реализации. Это может привести к закрытию пилотного проекта.

Если вы прочитали предисловие к книге, то уже знаете, что нечто подобное произошло и со мной. У вас есть всего пара месяцев на то, чтобы задать проекту нужный темп и убедить вышестоящих, что ваши эксперименты принесут практическую пользу. Эффективно используйте это время и устраните все возможные риски. Если вы не знаете, как писать хорошие тесты, прочитайте книгу или привлеките консультанта. Если вы не знаете, как улучшить тестируемость вашего кода, сделайте то же самое. Не тратьте время на то, чтобы заново изобретать методы тестирования.

11.3.4. Недостаток поддержки в команде

Если команда не поддерживает ваши усилия, добиться успеха будет практически невозможно, потому что вам будет слишком трудно совмещать свою дополнительную работу в новом проекте с повседневной. Вы должны стремиться к тому, чтобы команда стала частью нового процесса или, по крайней мере, не противодействовала ему.

Поговорите с участниками команды об изменениях. Попытки заручиться поддержкой одного за другим иногда неплохо работают, но откровенный разговор с группой о ваших усилиях — и ответы на их непростые вопросы — также может быть очень ценным. Что бы вы ни выбрали, не считайте поддержку команды

чем-то само собой разумеющимся. Следует реально понимать, на что вы идете; вам придется работать с этими людьми на ежедневной основе.

11.4. ФАКТОРЫ ВЛИЯНИЯ

Я посвятил влиянию на поведение целую главу в своей книге «Elastic Leadership» (Manning, 2016). Если эта тема покажется вам интересной, рекомендую ознакомиться с этой книгой или дополнительно почитать о ней на сайте 5whys.com.

Одна из тем, которую я считаю еще более интересной, чем юнит-тесты, — люди и почему они ведут себя именно так, а не иначе. Бывает очень обидно, когда вы пытаетесь убедить кого-то начать что-то делать (например, TDD), но, несмотря на все ваши старания, они этого делать не желают. Возможно, вы уже привели им разумные доводы, но видите, что они никак не реагируют.

В книге «Influencer: The Power to Change Anything»¹ (McGraw-Hill, 2007), написанной Керри Паттерсоном (Kerry Patterson), Джозефом Гренни (Joseph Grenny), Дэвидом Максфилдом (David Maxfield), Роном Макмилланом (Ron McMillan) и Элом Свитцлером (Al Switzler), вы найдете следующую мантру (слегка переформулированную):

Мир идеально выстроен для каждого поведения, которое мы наблюдаем. Но существуют другие факторы, кроме желания человека что-то делать или его способности это делать, которые влияют на поведение. Тем не менее мы редко задумываемся об этих факторах.

В книге представлены шесть факторов влияния.

- *Личная состоятельность* — обладает ли человек всеми навыками или знаниями, чтобы сделать то, что от него требуется?
- *Личная мотивация* — получает ли человек удовлетворение от правильного поведения или неудовольствие от неправильного? Обладает ли он достаточным самоконтролем, чтобы придерживаться правильного поведения, когда это труднее всего сделать?
- *Социальная состоятельность* — предоставляете ли вы или другие поддержку, информацию и ресурсы, необходимые этому человеку, особенно в критический момент?
- *Социальная мотивация* — окружающие активно поощряют правильное поведение и осуждают неправильное? Насколько эффективно вы и другие участники моделируете правильное поведение?

¹ Паттерсон К., Гренни Д., Максфилд Д., Макмиллан Р., Свитцлер Э. Агент влияния. Как изменить все, что угодно.

- *Структурная состоятельность* — присутствуют ли в окружении аспекты (машина сборки, бюджет и т. д.), которые делают это поведение удобным, простым и безопасным? Достаточно ли ориентиров и напоминаний, помогающих выдержать нужный курс?
- *Структурная мотивация* — существуют ли четкие, осмысленные стимулы (денежные выплаты, привилегии и т. д.) в тех случаях, когда вы или другие ведете себя правильно или неправильно? Способствуют ли краткосрочные вознаграждения желаемым долгосрочным результатам и вариантам поведения, которые вы хотите подкрепить или избежать?

Считайте это своего рода коротким чек-листом, который поможет вам понять, почему что-то идет вразрез с ожиданиями. Затем учтите еще один важный факт: в вашей ситуации могут действовать сразу несколько факторов. Чтобы поведение изменилось, следует изменить все действующие факторы. Если изменить только один, то поведение останется прежним.

В таблице 11.1 приведен пример воображаемого чек-листа для разработчика, не применяющего TDD. (Учтите, что для разных работников списки будут выглядеть по-разному.)

Таблица 11.1. Чек-лист факторов влияния

| Фактор влияния | Вопросы, которые следует задать | Примеры ответов |
|----------------------------|---|--|
| Личная состоятельность | Обладает ли человек всеми навыками или знаниями, чтобы сделать то, что от него требуется? | Да. Он прошел трехдневный учебный курс TDD с Роем Ошеровым |
| Личная мотивация | Получает ли человек удовлетворение от правильного поведения или неудовольствие от неправильного? Обладает ли он достаточным самоконтролем, чтобы придерживаться правильного поведения, когда это труднее всего сделать? | Я с ним переговорил — и да, ему нравится практика TDD |
| Социальная состоятельность | Предоставляете ли вы или другие поддержку, информацию и ресурсы, необходимые этому человеку, особенно в критический момент? | Да |
| Социальная мотивация | Окружающие активно поощряют правильное поведение и осуждают неправильное? Насколько эффективно вы и другие участники моделируете правильное поведение? | Насколько это возможно |

Таблица (окончание)

| Фактор влияния | Вопросы, которые следует задать | Примеры ответов |
|-----------------------------|---|---|
| Структурная состоятельность | Присутствуют ли в окружении аспекты (машина сборки, бюджет и т. д.), которые делают это поведение удобным, простым и безопасным? Достаточно ли ориентиров и напоминаний, помогающих выдержать нужный курс? | У него нет бюджета на машину сборки* |
| Структурная мотивация | Существуют ли четкие, осмысленные стимулы (денежные выплаты, привилегии и т. д.) в тех случаях, когда вы или другие ведете себя правильно или неправильно? Способствуют ли краткосрочные вознаграждения желаемым долгосрочным результатам и вариантам поведения, которые вы хотите подкрепить или избежать? | Когда он пытается выделить время на юнит-тестирование, начальство говорит, что он попусту тратит время. При досрочном выпуске продукта скверного качества он получает премию* |

Я пометил звездочками пару пунктов в правом столбце, требующих дополнительной работы. Здесь я выявил две проблемы, которые необходимо решить. Если решить только проблему с бюджетом на машину сборки, это не изменит его поведение. Ему нужно заполучить машину сборки и отговорить руководство от поощрения досрочного выпуска некачественной версии.

Эта тема более подробно рассматривается в моей книге «*NOTEs to a Software Team Leader*» (Team Agile Publishing, 2014), посвященной управлению технической группой. Книгу можно найти на сайте 5whys.com.

11.5. НЕПРОСТЫЕ ВОПРОСЫ И ОТВЕТЫ

В этом разделе представлен ряд вопросов, с которыми я сталкивался в разных организациях. Обычно они возникают из-за предположений о том, что реализация юнит-тестирования может навредить кому-то лично — руководителю, отвечающему за соблюдение сроков, или работнику QA, который опасается, что перестанет быть нужным компании. Как только вы поймете, почему возникла проблема, важно разобраться с ней непосредственно или опосредованно. В противном случае всегда будет существовать подспудное сопротивление.

11.5.1. Насколько юнит-тестирование удлиняет текущий процесс?

Тимлиды, руководители проектов и клиенты обычно чаще всех спрашивают, сколько времени добавит юнит-тестирование к процессу. Эти люди больше других беспокоятся о затратах времени.

Начнем с фактов. Исследования показали, что повышение общего качества кода в проекте может увеличить производительность и сократить сроки поставки продукта. Как это согласуется с тем фактом, что создание тестов требует дополнительного времени? Положительный эффект достигается прежде всего за счет того, что сопровождать код и исправлять ошибки станет проще.

ПРИМЕЧАНИЕ Если вам захочется ознакомиться с исследованиями в области качества кода и производительности, обращайтесь к книгам «Programming Productivity» (McGraw-Hill College, 1986) и «Software Assessments, Benchmarks, and Best Practices» (Addison-Wesley Professional, 2000); автор обеих книг — Кейперс Джонс (Capers Jones).

Спрашивая вас о времени, тимлиды на самом деле могут подразумевать: «Что мне сказать руководителю проекта, если мы не уложимся в дедлайн?» При этом они могут считать процесс полезным, но искать аргументы, чтобы возразить вам в предстоящем его обсуждении. А может, они задают вопрос в отношении не всего продукта, а конкретной функциональности. Хотя, конечно, руководитель проекта или клиент, задающий вопрос о времени, обычно имеет в виду окончательный релиз продукта.

Так как разных людей интересуют разные вопросы, ваши ответы могут варьироваться. Например, юнит-тестирование может удвоить время реализации некоторой функциональности, но при этом общее время выпуска продукта сократится. Чтобы понять это, рассмотрим реальную ситуацию, в которой я сам побывал.

История о двух командах

Одна крупная компания, которую я консультировал, решила внедрить в свой процесс практику юнит-тестирования. Начать предполагалось с пилотного проекта. В нем участвовала группа разработчиков, добавлявших новую функциональность в большое уже существующее приложение. Основным источником дохода компании было это приложение для выставления счетов и настройка его частей для разных клиентов. На компанию трудились тысячи разработчиков по всему миру.

Для проверки успеха пилотного проекта использовались следующие метрики:

- время, потраченное командой на каждую стадию разработки;
- общее время выпуска проекта для клиента;
- количество ошибок, обнаруженных клиентом после выпуска.

Та же статистика была собрана для аналогичной функциональности, создаваемой другой командой для другого клиента. Обе задачи были практически одинакового размера, а группы имели приблизительно одинаковый

уровень квалификации и опыта. В обоих случаях целью была настройка продукта — с юнит-тестами и без. В таблице 11.2 сравниваются временные характеристики.

В целом время выпуска версии с тестами было меньше, чем у версии без тестов. При этом руководители команды с юнит-тестами изначально не верили в успех пилотного проекта, потому что они рассматривали как критерий успеха только статистику реализации (кодинга), то есть первую строку в таблице 11.2, а не последнюю. Программирование функциональности заняло вдвое больше времени (потому что юнит-тесты требуют, чтобы вы писали больше кода). Несмотря на это, дополнительные затраты времени были с избытком компенсированы, когда команда QA обнаружила меньше ошибок, требующих исправления.

Таблица 11.2. Прогресс команд и результаты, достигнутые с юнит-тестами и без них

| Стадия | Без тестов | С тестами |
|---|--|--|
| Реализация (написание кода) | 7 дней | 14 дней |
| Интеграция | 7 дней | 2 дня |
| Тестирование и исправление ошибок | Тестирование, 3 дня Исправление, 3 дня Тестирование, 3 дня Исправление, 2 дня Тестирование, 1 день Итого: 12 дней | Тестирование, 3 дня Исправление, 1 день Тестирование, 1 день Исправление, 1 день Тестирование, 1 день Итого: 7 дней |
| Общее время выпуска | 26 дней | 23 дня |
| Количество обнаруженных ошибок в продакшен-версии | 71 | 11 |

Вот почему так важно подчеркнуть, что, хотя юнит-тестирование увеличивает время реализации функциональности, общее время выпуска продукта сокращается из-за повышения качества и простоты сопровождения.

11.5.2. Не будет ли юнит-тестирование угрожать моей работе в отделе QA?

Юнит-тестирование не лишает работы специалистов по QA. QA-инженеры будут получать приложения с полными наборами юнит-тестов; это означает, что они смогут убедиться в прохождении всех юнит-тестов перед началом собственного процесса тестирования. Наличие готовых юнит-тестов в действительности

делает их работу более интересной. Вместо того чтобы проводить UI-отладку (где каждый второй щелчок по кнопке приводит к какому-нибудь исключению), они смогут сосредоточиться на поиске ошибок в логике реальных сценариев приложения. Юнит-тесты формируют первую линию защиты от ошибок, а работа QA — вторую линию: уровень приемлемости системы для пользователя. Когда инженер QA может сосредоточиться на более серьезных проблемах, это способствует созданию более качественных приложений.

В некоторых компаниях инженеры QA пишут код; они могут помочь с написанием юнит-тестов для приложений. Они работают в кооперации с разработчиками приложений, а не вместо них. Как разработчики, так и инженеры QA могут создавать юнит-тесты.

11.5.3. Есть ли доказательства, что юнит-тестирование действительно помогает?

Я не знаю конкретных исследований относительно того, помогает ли юнит-тестирование писать более качественный код, на которые я бы мог сослаться. Многие исследования посвящены Agile-методам; юнит-тестирование является лишь одним из них. Некоторые эмпирические доказательства можно найти в интернете, у компаний и коллег, которые добились замечательных результатов и никогда не захотят возвращаться к кодовой базе без тестов. Некоторые исследования по теме TDD можно найти на сайте QA Lead: <http://mng.bz/dddo>.

11.5.4. Почему отдел QA все еще находит ошибки?

Возможно, в вашей компании уже нет отдела QA, но эта практика все еще достаточно широко распространена. Как бы то ни было, ошибки все равно будут обнаруживаться. Используйте тесты на разных уровнях, как описано в главе 10, чтобы достичь уверенности в работоспособности вашего приложения на разных уровнях. Юнит-тесты предоставляют быструю обратную связь и делают сопровождение проще, но наивысшая степень уверенности может быть обретена только на уровнях интеграционного тестирования.

11.5.5. У нас большой объем кода без тестов: с чего начинать?

Исследования, проведенные в 1970-х и 1980-х годах, показали, что обычно 80 % ошибок обнаруживается в 20 % кода. Фокус в том, чтобы найти код с наибольшим количеством проблем. Как правило, любая команда может сказать, какие компоненты наиболее проблематичны. Начните с них. Вы всегда можете добавить метрики, оценивающие количество ошибок на класс.

Источники соотношения 80/20

Примеры исследований, демонстрирующих, что 80 % ошибок содержится в 20 % кода: Albert Endres, «An analysis of errors and their causes in system programs», IEEE Transactions on Software Engineering 2 (июнь 1975), 140–149; Lee L. Gremillion, «Determinants of program repair maintenance requirements», Communications of the ACM 27, no. 8 (август 1984), 826–832; Barry W. Boehm, «Industrial software metrics top 10 list», IEEE Software 4, no. 9 (сентябрь 1987), 84–85 (перепечатано в бюллетене IEEE и доступно в интернете по адресу <http://mng.bz/rjjJ>); Shull и др., «What we have learned about fighting defects», Proceedings of the 8th International Symposium on Software Metrics (2002), 249–258.

Тестирование унаследованного кода требует несколько иного подхода, чем написание нового кода с тестами. Подробнее об этом в главе 12.

11.5.6. А если мы разрабатываем комбинацию программного и аппаратного обеспечения?

Юнит-тесты можно использовать даже в этом случае. Изучите уровни тестов, рассмотренные в предыдущей главе, и убедитесь в том, что они покрывают как программную, так и аппаратную часть. Тестирование оборудования обычно требует использования имитаторов и эмуляторов на разных уровнях, однако на практике часто создаются наборы тестов как для низкоуровневого встроенного, так и для высокоуровневого кода.

11.5.7. Как узнать, что в наших тестах нет ошибок?

Необходимо убедиться в том, что тесты проходят там, где должны проходить, и проваливаются там, где должны проваливаться. TDD помогает убедиться в том, что вы не забыли проверить эти ситуации. За кратким вводным описанием TDD обращайтесь к главе 1.

11.5.8. Зачем мне нужны тесты, если отладчик показывает, что мой код работает?

Отладчики мало чем помогут с многопоточным кодом. Кроме того, вы можете убедиться в том, что ваш собственный код работает нормально, но как насчет кода, написанного другими? Как вы узнаете, что он работает? А как они будут знать, что ваш код работает и они ничего не сломали при внесении изменений? Помните, что написание кода — всего лишь первый шаг его жизненного пути.

Большую часть своего срока жизни код будет находиться в режиме сопровождения. Необходимо позаботиться о том, чтобы код сообщал о своих сбоях при помощи юнит-тестов.

В исследовании Кертиса (Curtis), Краснера (Krasner) и Иско (Iscoe) («A field study of the software design process for large systems», Communications of the ACM 31, no. 11 (ноябрь 1988), 1268–1287) показано, что большая часть дефектов обусловлена не самим кодом, а недопониманием между людьми, изменениями требований и недостаточным знанием предметной области. Даже если вы — лучший программист в мире, всегда есть шанс, что кто-то предложит вам закодировать что-то неправильное и вы это сделаете. А когда код нужно будет изменить, вы будете рады, что у вас есть тесты для всего остального. Они помогут убедиться в том, что вы ничего не сломали.

11.5.9. Как насчет TDD?

TDD — вопрос стиля. Лично я вижу большую практическую ценность TDD, и многие люди находят эту методологию эффективной и полезной, но другие считают, что для них достаточно хорошо писать тесты после кода. Выбирайте сами.

ИТОГИ

- Реализация юнит-тестирования в организации — задача, с которой многим читателям книги придется время от времени иметь дело.
- Не портите отношения с людьми, которые могут вам помочь. Научитесь распознавать сторонников и скептиков внутри организации. Сделайте обе группы частью процесса изменений.
- Определите возможные отправные точки. Начните с небольшой команды или маленького проекта, чтобы добиться быстрого успеха и свести к минимуму риски.
- Наглядно демонстрируйте окружающим прогресс в работе. Ориентируйтесь на конкретные цели, метрики и KPI.
- Обратите внимание на потенциальные причины неудач, такие как отсутствие направляющей силы, недостаточная поддержка руководства или команды.
- Приготовьтесь дать обстоятельные ответы на вопросы, которые вам с большой вероятностью зададут.

12

Работа с унаследованным кодом

В ЭТОЙ ГЛАВЕ

- ✓ Типичные проблемы с унаследованным кодом
- ✓ С чего начать написание тестов для такого кода

Однажды я консультировал крупную компанию-разработчика, выпускавшую ПО для выставления счетов. У них было свыше 10 000 программистов, а в продуктах, подпродуктах и связанных проектах использовались .NET, Java и C++. Программный код существовал в той или иной форме более пяти лет, и в обязанности многих инженеров входило сопровождение и расширение существующей функциональности.

Моей работой было обучение методам TDD сотрудников нескольких подразделений (для всех языков). Примерно для 90 % разработчиков, с которыми я работал, эти методы так и не стали рабочей реальностью по ряду причин, и часть этих причин была связана с унаследованным кодом.

- Писать тесты для существующего кода было трудно.
- Провести рефакторинг существующего кода было практически невозможно (или на это не было времени).
- Некоторые люди не желали изменять свои решения.

- Несовершенство инструментов (или их отсутствие) препятствовало работе.
- Было трудно определить, с чего начать.

Каждый, кто когда-либо пытался добавить тесты в уже существующую систему, знает, что в большинстве случаев написать тесты практически невозможно. Обычно эти системы создавались без включения специальных мест (*швов*) в программном обеспечении, которые допускают расширение или замену существующих компонентов.

При работе с унаследованным кодом приходится решать две проблемы.

- У нас много работы, с чего начать добавление тестов? На чем сосредоточить усилия?
- Как провести безопасный рефакторинг кода, если у нас изначально нет тестов?

В этой главе рассматриваются эти непростые вопросы, связанные с унаследованными кодовыми базами. Я перечислю некоторые приемы, источники информации и инструменты, которые могут вам помочь.

12.1. С ЧЕГО НАЧИНАТЬ ДОБАВЛЕНИЕ ТЕСТОВ?

Допустим, у вас имеется существующий код внутри компонентов и вам нужно создать приоритизированный список тех компонентов, тестирование которых принесет наибольшую пользу. Рассмотрим несколько факторов, которые необходимо учитывать.

- *Логическая сложность* — этим термином обозначается объем логики в компоненте (вложенные `if`, варианты `switch`, рекурсия и т. д.). Такая сложность также называется *цикломатической*, и вы можете использовать разные инструменты для ее автоматической проверки.
- *Уровень зависимостей* — относится к количеству зависимостей в компоненте. Сколько зависимостей вам придется разорвать, чтобы протестировать этот класс? Взаимодействует ли он, например, с внешним компонентом электронной почты или, возможно, где-то вызывает статический метод `log`?
- *Приоритет* — общий приоритет компонента в проекте.

Каждому компоненту можно назначить оценку для этих факторов: от 1 (низкий приоритет) до 10 (высокий приоритет). В таблице 12.1 перечислены классы с оценками этих факторов. Назовем ее *таблицей тестопригодности*.

По данным из таблицы 12.1 можно построить диаграмму вроде изображенной на рис. 12.1, связывающую компоненты с их ценностью для проекта и количеством зависимостей. Вы можете смело игнорировать позиции, находящиеся ниже выбранного вами порога логики (обычно я выбираю порог 2 или 3), так

что на Person и ConfigManager можно не обращать внимания. Остаются только верхние два компонента на рис. 12.1.

Таблица 12.1. Простая таблица тестопригодности

| Компонент | Логическая сложность | Уровень зависимостей | Приоритет | Примечания |
|---------------|----------------------|----------------------|-----------|---|
| Utils | 6 | 1 | 5 | Вспомогательный класс содержит мало зависимостей, но много логики. Его проще тестиировать, и это принесет ощутимую пользу |
| Person | 2 | 1 | 1 | Класс для хранения данных с минимумом логики и без зависимостей. Его тестирование не принесет сколько-нибудь заметной пользы |
| TextParser | 8 | 4 | 6 | Класс содержит множество зависимостей с большим объемом логики. Вдобавок он является частью высокоприоритетной задачи в проекте. Его тестирование будет очень полезным, но сложным и затратным по времени |
| ConfigManager | 1 | 6 | 1 | Класс хранит конфигурационные данные и читает файлы с диска. Он содержит мало логики, но много зависимостей. Его тестирование принесет мало пользы, но будет сложным и потребует много времени |

Чтобы решить, с каких компонентов начать тестирование, на эту диаграмму можно взглянуть с двух сторон (рис. 12.2):

- выбрать вариант, более сложный по логике, но простой для тестирования (вверху слева);
- выбрать вариант, более сложный по логике и более сложный для тестирования (вверху справа).

Вопрос в том, какой путь вы выберете. С чего начать — с простого или сложного?

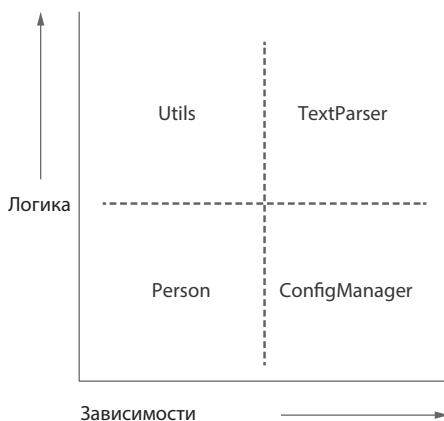


Рис. 12.1. Свойства компонентов для тестопригодности

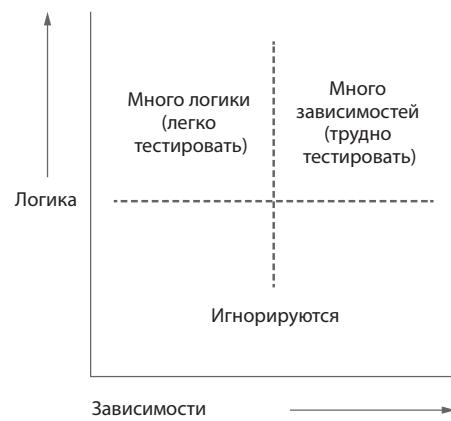


Рис. 12.2. Простые, сложные и несущественные компоненты в соответствии с объемом логики и количеством зависимостей

12.2. ПРИНЯТИЕ РЕШЕНИЯ ПО СТРАТЕГИИ ВЫБОРА

Как объяснялось в предыдущем разделе, можно начать с компонентов, которые просты, или с тех, которые сложны в тестировании (из-за большого количества зависимостей). Каждая стратегия создает те или иные проблемы.

12.2.1. Плюсы и минусы стратегии «начать с простого»

Если начать с компонентов, имеющих меньше зависимостей, то на первой стадии писать исходные тесты будет быстрее и проще. Но, как показано на рис. 12.3, здесь скрывается ловушка.

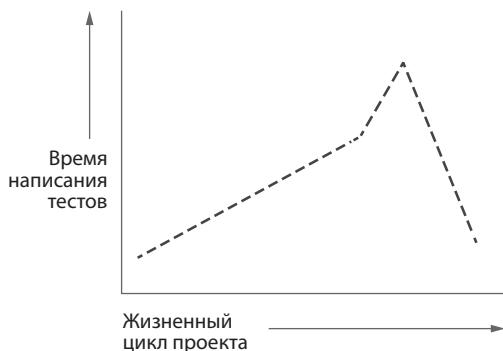


Рис. 12.3. Если начать с более простых компонентов, в будущем время тестирования будет возрастать, пока вы не справитесь с самыми сложными компонентами

На рис. 12.3 показано, сколько времени занимает организация тестирования компонентов на протяжении жизненного цикла проекта. На начальной стадии тесты пишутся легко, но с течением времени у вас остаются компоненты, тестировать которые будет все труднее и труднее, причем самые сложные ожидают вас в конце цикла, когда все только и думают, как бы поскорее закончить проект.

Если ваша команда еще не имеет достаточного опыта в юнит-тестировании, лучше начать с простого. С течением времени команда освоит приемы, необходимые для работы с более сложными компонентами и зависимостями. Возможно, таким командам на начальной стадии лучше избегать любых компонентов, у которых количество зависимостей превышает некоторый порог (разумное значение — 4).

12.2.2. Плюсы и минусы стратегии «начать со сложного»

Казалось бы, начиная с более сложных компонентов, вы несете потери на начальной стадии, но у такого подхода есть свои преимущества при условии, что у вашей команды имеется опыт применения методов юнит-тестирования. На рис. 12.4 представлено среднее время написания теста для отдельного компонента на протяжении жизненного цикла проекта, если вы начнете с тестирования компонентов с большим количеством зависимостей.

При такой стратегии у вас может уйти день и более на то, чтобы добиться прохождения даже простейших тестов для более сложных компонентов. Но обратите внимание на быстрое сокращение времени, необходимого для написания тестов, относительно медленного сокращения на рис. 12.3. Каждый раз, когда вы переходите к тестированию очередного компонента и проводите рефакторинг, чтобы улучшить его тестируемость, вы одновременно решаете проблемы, как связанные с тестируемостью используемых им зависимостей, так и касающиеся других компонентов. Так как этот компонент содержит большое количество зависимостей, его рефакторинг может улучшить ситуацию для других частей системы. Этим и объясняется быстрое сокращение времени написания тестов.

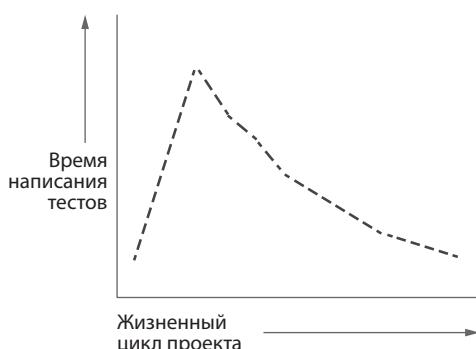


Рис. 12.4. Если применить стратегию «начать со сложного», время написания тестов для компонентов на начальной стадии будет высоким, но затем оно сокращается благодаря рефакторингу многочисленных зависимостей

Стратегия «начать со сложного» возможна только в том случае, если у вашей команды имеется опыт юнит-тестирования, потому что эта стратегия сложнее реализуется. Если у команды есть необходимый опыт, используйте критерий приоритетности компонентов для принятия решения о том, стоит ли начинать с простых или сложных частей. Возможно, вы выберете некоторое их сочетание, но важно заранее знать, сколько усилий это потребует и к каким возможным последствиям может привести.

12.3. НАПИСАНИЕ ИНТЕГРАЦИОННЫХ ТЕСТОВ ПЕРЕД РЕФАКТОРИНГОМ

Если вы планируете провести рефакторинг кода для улучшения testируемости (чтобы вы потом могли писать юнит-тесты), то желательно написать тесты в интеграционном стиле для вашей рабочей системы, чтобы убедиться, что вы ничего не сломали в фазе рефакторинга.

Однажды я консультировал компанию, работавшую над большим унаследованным проектом. Мы взаимодействовали с разработчиком, которому была поручена реализация менеджера конфигурации XML. Проект не имел тестов, и вряд ли можно было говорить о его testируемости. Вдобавок он был написан на C++, так что у нас не было инструмента, позволяющего легко изолировать компоненты от зависимостей без рефакторинга кода.

Разработчику нужно было добавить очередной атрибут в файл XML, чтобы потом иметь возможность прочитать и изменить его через существующий компонент конфигурации. В итоге мы написали пару интеграционных тестов, которые использовали реальную систему для сохранения и загрузки конфигурационных данных и проверяли значения, прочитанные и записанные в файл компонентом конфигурации. Эти тесты установили «исходное» рабочее поведение менеджера конфигурации как основу для дальнейшей работы.

Затем мы написали интеграционный тест: он показывал, что после того, как компонент прочитал этот файл, в памяти не было атрибута с именем, которое мы собирались добавить. Это доказывало отсутствие функциональности, и теперь у нас был тест, который будет проходить после того, как мы добавим новый атрибут в файл XML и правильно запишем в него данные из компонента.

После написания кода, который сохранял и загружал дополнительный атрибут, мы провели три интеграционных теста (два теста для исходной базовой реализации и один новый, который пытался прочитать новый атрибут). Все три теста проходили, и мы знали, что добавление новой функциональности не привело к нарушению существующей.

Как видите, процесс относительно прост.

- Добавьте один или несколько интеграционных тестов (без моков и стабов) в систему, чтобы доказать, что исходная система работает так, как нужно.
- Проведите рефакторинг или добавьте непроходящий тест для функциональности, которую вы собираетесь добавить в систему.
- Проведите рефакторинг и изменяйте систему небольшими блоками. Выполняйте интеграционные тесты как можно чаще, чтобы увидеть, не сломали ли вы что-нибудь.

Иногда кажется, что интеграционные тесты проще написать, чем юнит-тесты, потому что вам не нужно понимать внутреннюю структуру кода или знать, в каком месте следует внедрять различные зависимости. Но выполнение этих тестов в вашей локальной системе может оказаться трудозатратным или слишком долгим, потому что вам придется следить за тем, чтобы каждая мелочь, необходимая системе, была на своем месте.

Фокус в том, чтобы работать над теми частями системы, которые вам нужно исправить или функциональность которых расширить. Не отвлекайтесь на другие части системы. При таком подходе она будет расти в правильных местах, но при этом останутся мосты, которые вы сможете перейти, когда доберетесь до них.

Добавляя новые и новые тесты, вы сможете провести рефакторинг системы и добавить в нее юнит-тесты, чтобы добиться простоты в сопровождении и тестируемости. Это требует времени (иногда целые месяцы), но дело того стоит.

В главе 7 книги Владимира Хорикова «Unit Testing Principles, Practices, and Patterns»¹ (Manning, 2020) рассматривается подробный пример такого рефакторинга. Прочитайте эту книгу, чтобы узнать больше по данной теме.

12.3.1. Прочтайте книгу Майкла Физерса об унаследованном коде

Книга Майкла Физерса «Working Effectively with Legacy Code»² (Pearson, 2004) – еще один ценный источник информации о проблемах, с которыми можно столкнуться при работе с унаследованным кодом. В книге подробно описаны многие методы рефакторинга и ловушки, о которых на этих страницах не говорится ни слова. Она на вес золота. Прочтите ее.

¹ Хориков В. Принципы юнит-тестирования. СПб.: Питер.

² Физерс М. Эффективная работа с унаследованным кодом.

12.3.2. Используйте CodeScene для анализа своего рабочего кода

Программа, которая называется CodeScene, позволяет обнаружить многие «технические долги» и скрытые проблемы (и не только) в унаследованном коде. Это коммерческий продукт, и, хотя лично я им не пользовался, слышал отличные отзывы. За дополнительной информацией обращайтесь на сайт <https://codescene.com/>.

ИТОГИ

- Прежде чем переходить к написанию тестов для унаследованного кода, важно классифицировать разные компоненты по количеству зависимостей, объему логики и общей приоритетности каждого компонента для проекта. Под логической (цикломатической) сложностью компонента понимается объем логики в компоненте (вложенные `if`, варианты `switch`, рекурсия и т. д.).
- Располагая этой информацией, вы можете выбрать компоненты для работы в зависимости от того, насколько легко или трудно будет организовать их тестирование.
- Если опыт юнит-тестирования в вашей команде недостаточен (или его вообще нет), желательно начать с простых компонентов, чтобы уверенность команды росла с добавлением все большего количества тестов в систему.
- Если у вас опытная команда, стратегия начала тестирования со сложных компонентов может помочь быстрее пройти через остальные части системы.
- Перед крупномасштабным рефакторингом напишите интеграционные тесты, которые переживут этот рефакторинг без значительных изменений. После завершения рефакторинга замените большинство этих интеграционных тестов меньшими, более простыми в сопровождении юнит-тестами.

Приложения

Манки-патчинг

функций и модулей

В главе 3 был представлен ряд методов создания стабов, которые я назвал «общепринятыми» в том смысле, что они считаются безопасными как для простоты в сопровождении, так и для читабельности кода и тестов, которые пишутся на их основе. В этом приложении я опишу некоторые менее общепринятые и менее безопасные способы, позволяющие создавать в тестах фейки для целых модулей.

П.1. ОБЯЗАТЕЛЬНОЕ ПРЕДУПРЕЖДЕНИЕ

Что касается глобального исправления ошибок и создания стабов для функций и модулей, у меня есть как хорошие, так и плохие новости. Да, вы это сможете сделать: я покажу несколько способов. Стоит ли так делать? Не уверен. По моему опыту затраты на сопровождение тестов с методами, которые будут описаны, выше, чем при сопровождении хорошо параметризованного кода или кода с правильно встроенными швами.

Тем не менее в некоторых особых случаях у вас может возникнуть необходимость в этих методах. Например, это может произойти при моделировании зависимостей в коде, который вам не принадлежит и который вы не можете изменить, а также при использовании непосредственно исполняемых функций или модулей. Еще один возможный вариант — предоставление модулем функций без объектов, что сильно ограничивает возможности создания фейков.

Старайтесь избегать приемов, описанных в этом приложении, насколько это возможно. Если вы сможете найти способ записать свои тесты или провести рефакторинг кода, чтобы эти методы вам не понадобились, поступайте именно так. Если все остальное не проходит, приемы из этого приложения можно считать неизбежным злом. Если вы вынуждены использовать их, постарайтесь свести к минимуму область их применения. Ваши тесты пострадают, станут более хрупкими и будут хуже читаться.

А теперь к делу.

П.2. МАНКИ-ПАТЧИНГ ФУНКЦИЙ И ВОЗМОЖНЫЕ ПРОБЛЕМЫ

Под *манки-патчингом* (monkey-patching) понимается акт изменения поведения экземпляра программы во время ее выполнения. Впервые я столкнулся с этим термином в ходе работы на Ruby, где этот прием применяется очень часто. В JavaScript также легко «исправить» функцию во время выполнения.

В главе 3 мы рассмотрели проблему управления временем в тестах и коде. С помощью манки-патчинга можно взять любую функцию, глобальную или локальную, и заменить ее (для конкретной области видимости JavaScript) другой реализацией. Если вы захотите повлиять на получение времени, можно изменить глобальную функцию `Date.now`, чтобы в дальнейшем любой код — как рабочий, так и код тестов — был с этими изменениями.

В листинге П. 1 приведен тест, который делает это с исходным рабочим кодом, использующим `Date.now` напрямую. Он заменяет глобальную функцию `Date.now` фейком, чтобы иметь возможность контролировать время в ходе выполнения теста.

Листинг П.1. Проблемы с заменой глобальной функции `Date.now()`

```
describe('v1 findRecentlyRebooted', () => {
  test('given 1 of 2 machines under threshold, it is found', () => {
    const originalNow = Date.now;
    const fromDate = new Date(2000,0,3); | Замена Date.now
    Date.now = () => fromDate.getTime(); | произвольной датой
                                              ← Сохранение исходной реализации Date.now

    const rebootTwoDaysEarly = new Date(2000,0,1);
    const machines = [
      { lastBootTime: rebootTwoDaysEarly, name: 'ignored' },
      { lastBootTime: fromDate, name: 'found' }];
    const result = findRecentlyRebooted(machines, 1, fromDate);

    expect(result.length).toBe(1);
    expect(result[0].name).toContain('found');

    Date.now = originalNow; ← Восстановление исходной
  }); | реализации Date.now
});
```

В этом листинге глобальная реализация `Date.now` заменяется произвольно выбранной датой. Так как функция является глобальной, замена может повлиять на другие тесты, поэтому мы «прибираем за собой» в конце теста, восстанавливая исходную реализацию `Date.now` на прежнем месте.

Такой тест создает несколько серьезных проблем. Прежде всего, если проверки не проходят, они выдают исключения; это означает, что в случае сбоя

восстановление исходной версии `Date.now` никогда не произойдет. А значит, измененное глобальное время может негативно влиять на другие тесты.

Кроме того, сохранять функцию времени, а потом возвращать ее на место неудобно. Тест становится длиннее, его труднее прочитать, а вдобавок и труднее написать. Наконец, разработчик легко может забыть о необходимости восстановить глобальное состояние.

Наконец, такое решение идет в ущерб параллелизму. Jest неплохо справляется с этой проблемой, так как для каждого файла теста создается отдельный набор зависимостей, но в других фреймворках с параллельным выполнением тестов может возникнуть ситуация гонки. Сразу несколько тестов могут изменять глобальное время или ожидать, что оно имеет определенное значение. При параллельном выполнении эти тесты начнут конфликтовать, создавать ситуации гонки в глобальном состоянии и влиять друг на друга. В нашем случае это не обязательно, но, если вы захотите устраниТЬ неопределенность, Jest позволяет включить в командную строку дополнительный параметр `--runInBand` для запрета параллелизма.

Некоторых из этих проблем можно избежать, воспользовавшись вспомогательными функциями `beforeEach()` и `afterEach()`.

Листинг П.2. Использование функций `beforeEach()` и `afterEach()`

```
describe('v2 findRecentlyRebooted', () => {
  let originalNow;
  beforeEach(() => originalNow = Date.now());
  afterEach(() => Date.now = originalNow);
  test('given 1 of 2 machines under threshold, it is found', () => {
    const fromDate = new Date(2000,0,3);
    Date.now = () => fromDate.getTime();

    const rebootTwoDaysEarly = new Date(2000,0,1);
    const machines = [
      { lastBootTime: rebootTwoDaysEarly, name: 'ignored' },
      { lastBootTime: fromDate, name: 'found' }];
    const result = findRecentlyRebooted(machines, 1, fromDate);
    expect(result.length).toBe(1);
    expect(result[0].name).toContain('found');
  });
});
```

Листинг П. 2 решает часть проблем, но не все. В нем хорошо то, что вам уже не нужно помнить о необходимости сохранять и восстанавливать `Date.now`, потому что вызовы `beforeEach()` и `afterEach()` сделают это за вас. Кроме того, тесты теперь лучше читаются.

Однако при этом остается потенциальная проблема с параллельным выполнением тестов. У Jest хватает сообразительности, чтобы запускать параллельные тесты только на уровне файла, то есть тесты из какого-то конкретного файла будут выполняться линейно, но такое поведение не гарантировано для тестов из разных файлов. Любой из параллельных тестов может иметь собственные функции `beforeEach()` и `afterEach()`, которые сбрасывают глобальное состояние и могут случайно повлиять на наши тесты.

Я не сторонник замещения глобальных объектов (паттерн «синглтон» в большинстве типизированных языков), если только без этого можно обойтись. За этот подход всегда приходится расплачиваться: больше кода, более сложное сопровождение, большая хрупкость тестов, неявное влияние на другие тесты и постоянное беспокойство о восстановлении измененного, — и это еще не все причины. В большинстве случаев код становится лучше, когда я намеренно включаю швы в структуру тестируемого кода вместо того, чтобы делать это неявно, как мы только что сделали.

Если учесть, что все больше фреймворков начинают копировать функциональность Jest и запускать тесты параллельно, замещение фейками глобальных объектов становится все более и более рискованным.

П.2.1. Манки-патчинг функции по схеме Jest

Для полноты картины стоит упомянуть о том, что в Jest также поддерживается идея манки-патчинга с использованием пары функций, работающих совместно: `spyOn` и `mockImplementation`.

Начнем с `spyOn`:

```
Date.now = jest.spyOn(Date, 'now')
```

`spyOn` получает в параметрах область видимости и функцию, которую нужно отслеживать. Обратите внимание: в параметре должна передаваться строка, что может создать проблемы при рефакторинге, — об этом легко забыть, если вы переименуете функцию.

П.2.2. Шпионы Jest

Слово «шпион» (spy) имеет чуть больше оттенков серого, чем термины, встречавшиеся ранее в книге, поэтому я стараюсь использовать его пореже (или вообще не использовать), если только это возможно. К сожалению, это слово является важной частью Jest API, поэтому стоит убедиться в том, что мы правильно понимаем его.

В книге Джерарда Месароша «XUnit Test Patterns»¹ (Addison-Wesley Professional, 2007) при рассмотрении шпионов говорится: «Используйте

¹ Месарош Д. Шаблоны тестирования xUnit: рефакторинг кода тестов.

тестовый дублер для сохранения непрямых выходных вызовов, обращенных к тестируемой системой (SUT) к другому компоненту для последующей валидации тестом». Единственное отличие шпиона от фейка или тестового дублера заключается в том, что шпион вызывает реальную реализацию используемой функции и отслеживает входные и выходные данные этой функции для последующей проверки в teste. Фейки и тестовые дублеры не используют реальную реализацию функции.

Мое уточненное определение *шпиона* довольно близко к этому. Шпион — это упаковка *единицы работы* в невидимую отслеживающую прослойку в *точках входа и выхода* без изменения нижележащей функциональности с целью отслеживания ее входных и выходных данных в ходе тестирования.

П.2.3. `spyOn` с `mockImplementation()`

Это поведение «отслеживания без изменения функциональности», присущее шпионам, также объясняет, почему одного использования `spyOn` недостаточно для замещения `Date.now`. Оно предназначено для отслеживания, а не для создания фейков.

Чтобы создать полноценный *фейк* для функции `Date.now` и превратить его в *стаб*, мы воспользуемся функцией `mockImplementation` (с именем, вводящим в заблуждение) для замены функциональности единицы работы:

```
jest.spyOn(Date, 'now').mockImplementation(() => /*вернуть стаб для времени */);
```

Слишком много «`мок`»

Если бы мне предложили выбрать новое имя для функции `mockImplementation`, я бы назвал ее `fakeImplementation`, потому что она может легко использоваться для создания как стабов, возвращающих данные, так и для моков, проверяющих данные, передаваемые им в параметрах. Слово «*мок*» (*mock*) слишком часто используется в нашей отрасли для обозначения всего, что не существует в реальности, тогда как понимание различий в терминах помогло бы в создании менее хрупких тестов. «*Mock*» в имени подразумевает, что речь идет о чем-то, что позднее будет использовано для проверки, — по крайней мере в моем представлении и с учетом того, как я различаю понятия моков и стабов в этой книге.

Во фреймворке Jest слово «*мок*» употребляется слишком часто, особенно если сравнить его API с изолирующим фреймворком вроде Sinon.js, в котором используются менее неожиданные имена, а «*мок*» не используется там, где оно является лишним.

А теперь посмотрим, как выглядит комбинация `spyOn` и `mockImplementation` в нашем коде.

Листинг П.3. Использование `jest.spyOn()` для манки-патчинга `Date.now()`

```
describe('v4 findRecentlyRebooted with jest spyOn', () => {
  afterEach(() => jest.restoreAllMocks());

  test('given 1 of 2 machines under threshold, it is found', () => {
    const fromDate = new Date(2000,0,3);
    Date.now = jest.spyOn(Date, 'now')
      .mockImplementation(() => fromDate.getTime());

    const rebootTwoDaysEarly = new Date(2000,0,1);
    const machines = [
      { lastBootTime: rebootTwoDaysEarly, name: 'ignored' },
      { lastBootTime: fromDate, name: 'found' }];
  });
}
```

Последняя деталь пазла скрывается в коде внутри `afterEach()`. Здесь вызывается другая функция с именем `jest.restoreAllMocks`, которая используется в Jest для возврата любого глобального состояния, для которого было включено отслеживание, к исходной реализации без каких-либо дополнительных следящих прослойек.

Обратите внимание: хотя мы используем шпион, мы не проверяем, что функция действительно была вызвана. Это означало бы, что мы используем его как мок, чего в нашем примере не происходит. У нас шпион используется как стаб. В Jest вам придется пройти через «шпиона» для создания стаба.

Все достоинства и недостатки, перечисленные выше, присутствуют и здесь. Я предпоготаю использовать параметры там, где это имеет смысл, вместо глобальных функций или переменных.

П.3. ИГНОРИРОВАНИЕ ЦЕЛОГО МОДУЛЯ В JEST

Из всех методов, упоминавшихся в приложении, этот наиболее безопасный, потому что он не обращается к внутренней механике тестируемой единицы работы. Он просто что-то игнорирует.

Если какой-то модуль вам не нужен во время выполнения тестов и вы просто хотите устраниить его из сценария без получения каких-либо фейковых данных, простой вызов `jest.mock('путь к модулю')` в начале файла теста прекрасно сработает без лишних хлопот.

Следующий раздел пригодится в том случае, если вы хотите моделировать в каждом тесте специальные данные от фейкового модуля, в результате чего приходится прикладывать больше усилий.

П.4. МОДЕЛИРОВАНИЕ ПОВЕДЕНИЯ МОДУЛЯ В КАЖДОМ ТЕСТЕ

Создание фейка для модуля, по сути, означает создание фейка для глобального объекта, который загружается каждый раз, когда `import` или `require` впервые используется тестируемым кодом. В зависимости от того, какой фреймворк тестирования вы применяете, модуль может кэшироваться во внутренней реализации или через стандартный механизм Node.js `require.cache`. Так как это происходит всего один раз, когда наш тест импортирует тестируемую систему, возникают небольшие проблемы при попытках моделирования другого поведения или данных для разных тестов в одном файле.

Чтобы смоделировать нестандартное поведение для нашего фейкового модуля, необходимо позаботиться о том, чтобы в тестах выполнялись следующие операции: удаление необходимого модуля из памяти, его замена, повторная загрузка и использование тестируемым кодом нового модуля вместо исходного повторным подключением тестируемого кода. Не так уж мало. Я обозначаю этот паттерн сокращением *CFRA* (Clear-Fake-Require-Act).

1. *Очистка* (Clear) — перед каждым тестом удалить все кэшированные или необходимые модули в памяти программы для запуска тестов.
2. В фазе подготовки теста:
 - a) *создание фейка* (Fake) — создание фейка для модуля, который будет за- требован действием `require`, вызываемым кодом тестов;
 - b) *запрос* (Require) — загрузка тестируемого кода при помощи `require` непо- средственно перед его вызовом.
3. *Действие* (Act) — активация точки входа.

Если вы забудете какие-либо из этих фаз или выполните их в неправильном порядке или в неправильной точке жизненного цикла теста, у вас появится много вопросов, когда при выполнении тестов моделирование будет работать неправильно. Но еще хуже, если *иногда* оно будет работать правильно.

Рассмотрим реальный пример, начиная со следующего кода.

Листинг П.4. Тестируемый код с зависимостью

```
const { getAllMachines } = require('./my-data-module'); ←
const daysFrom = (from, to) => {
  const ms = from.getTime() - new Date(to).getTime();
  const diff = (ms / 1000) / 60 / 60 / 24; // secs * min * hrs
  console.log(diff);
  return diff;
};
```

Зависимость,
для которой
создается фейк

```
const findRecentlyRebooted = (maxDays, fromDate) => {
  const machines = getAllMachines();
  return machines.filter(machine => {
    const daysDiff = daysFrom(fromDate, machine.lastBootTime);
    console.log(` ${daysDiff} vs ${maxDays}`);
    return daysDiff < maxDays;
  });
};
```

Первая строка содержит зависимость, которую в тесте необходимо устраниć. В данном случае это функция `getAllMachines`, деструктуризованная из модуля `my-data-module`. Так как мы используем функцию, отделенную от родительского модуля, нельзя просто создать фейки для функций родительского модуля и ожидать, что тесты благополучно пройдут. Приходится получать *деструктуризованную* функцию, чтобы получить фейковую функцию во время процесса деструктуризации, и здесь-то происходит самое интересное.

П.4.1. Создание стаба для модуля базовым вызовом require.cache

Прежде чем использовать Jest и другие фреймворки и создавать фейки для цепных модулей, давайте посмотрим, как добиться желаемого эффекта, и исследуем возможности разных фреймворков.

Паттерн CFRA можно применить без каких-либо фреймворков, используя `require.cache` напрямую.

Листинг П.5. Создание стабов с require.cache

```
const assert = require('assert');
const { check } = require('./custom-test-framework');

const dataModulePath = require.resolve('../my-data-module');

const fakeDataFromModule = fakeData => {
  delete require.cache[dataModulePath];           ← Очистка
  require.cache[dataModulePath] = {                ← Создание фейка
    id: dataModulePath,
    filename: dataModulePath,
    loaded: true,
    exports: {
      getAllMachines: () => fakeData
    }
  };
  require(dataModulePath);
};
```

```

const requireAndCall_findRecentlyRebooted = (maxDays, fromDate) => {
  const { findRecentlyRebooted } = require('../machine-scanner4'); ← Запрос
  return findRecentlyRebooted(maxDays, fromDate); ← Действие
};

check('given 1 of 2 machines under the threshold, it is found', () => {
  const rebootTwoDaysEarly = new Date(2000,0,1);
  const fromDate = new Date(2000,0,3);

  fakeDataFromModule([
    { lastBootTime: rebootTwoDaysEarly, name: 'ignored' },
    { lastBootTime: fromDate, name: 'found' }
  ]);

  const result = requireAndCall_findRecentlyRebooted(1, fromDate);
  assert(result.length === 1);
  assert(result[0].name.includes('found'));
});

```

К сожалению, с Jest этот код работать не будет, потому что Jest игнорирует `require.cache` и реализует собственный внутренний алгоритм кэширования. Чтобы выполнить этот тест, запустите его напрямую в командной строке Node.js. Вы увидите, что я реализовал собственную маленькую функцию `check()`, чтобы не использовать Jest API. С таким фреймворком, как Jasmine, тест будет работать нормально.

Помните эту строку в тестируемом коде?

```
const { getAllMachines } = require('./my-data-module');
```

Наши тесты должны *выполнять* эту деструктуризацию каждый раз, когда вы захотите вернуть фейковое значение. Это означает, что `require` или `import` для тестируемой единицы работы необходимо выполнять не в начале файла, а где-то в середине выполнения теста. Следующая часть листинга П. 5 показывает, как это происходит:

```

const requireAndCall_findRecentlyRebooted = (maxDays, fromDate) => {
  const { findRecentlyRebooted } = require('../machine-scanner4');
  return findRecentlyRebooted(maxDays, fromDate);
};

```

Именно из-за этого паттерна деструктуризации кода модули не являются простыми объектами со свойствами, к которым можно применить обычные методы манки-патчинга. Приходится пускаться на дополнительные ухищрения.

Установим соответствие четырех шагов CFRA с кодом из листинга П. 5.

- *Очистка (C)* — часть функции `fakeDataFromModule`, вызываемой во время теста.

- *Создание фейка (F)* — мы приказываем элементу словаря `require.cache` вернуть нестандартный объект, который вроде бы представляет модуль, но при этом имеет нестандартную реализацию, возвращающую `fakeData`.
- *Запрос (R)* — тестируемый код запрашивается как часть функции `requireAndCall_findRecentlyRebooted()`, вызываемой в ходе теста.
- *Действие (A)* — это часть той же функции `requireAndCall_findRecentlyRebooted()`, вызываемой тестом.

Обратите внимание: `beforeEach()` для этого теста не используется. Все делается прямо из теста, потому что каждый тест создает фейки для своих данных из модуля.

П.4.2. Создание стабов для нестандартных данных из модулей в Jest затруднено

Мы рассмотрели «базовый» способ создания стабов для нестандартных данных из модулей. Впрочем, при использовании Jest это обычно делается не так. Jest содержит ряд функций с очень похожими (и неудачно выбранными) именами для очистки и замены модулей, включая `mock`, `doMock`, `genMockFromModule`, `resetAllMocks`, `clearAllMocks`, `restoreAllMocks`, `resetModules` и другие. Вот это да!

Код, который я порекомендую здесь, кажется самым чистым и простым из всего Jest API в отношении читабельности и простоты в сопровождении. Другие вариации рассматриваются в репозитории GitHub по адресу <https://github.com/royoshero/ajout3-samples> и в каталоге `other-variations` по адресу <http://mng.bz/Jddo>.

При создании фейков модулей с Jest применяется стандартный паттерн.

1. Используйте `require` для модуля, который в своих тестах вы хотели бы заменить фейком.
2. Создайте стаб для модуля над тестами вызовом `jest.mock(имя_модуля)`.
3. В каждом teste прикажите Jest переопределить поведение одной из функций этого модуля при помощи `[имя_модуля].функция.mockImplementation()` или `mockImplementationOnce()`.

Результат может выглядеть так:

Листинг П.6. Создание стаба для модуля с Jest

```
const dataModule = require('../my-data-module');
const { findRecentlyRebooted } = require('../machine-scanner4');

const fakeDataFromModule = (fakeData) =>
  dataModule.getAllMachines.mockImplementation(() => fakeData);

jest.mock('../my-data-module');
```

```

describe('findRecentlyRebooted', () => {
  beforeEach(jest.resetAllMocks); //<- самый чистый способ

  test('given no machines, returns empty results', () => {
    fakeDataFromModule([]);
    const someDate = new Date(2000,0,1);

    const result = findRecentlyRebooted(0, someDate);

    expect(result.length).toBe(0);
  });

  test('given 1 of 2 machines under threshold, it is found', () => {
    const fromDate = new Date(2000,0,3);
    const rebootTwoDaysEarly = new Date(2000,0,1);
    fakeDataFromModule([
      { lastBootTime: rebootTwoDaysEarly, name: 'ignored' },
      { lastBootTime: fromDate, name: 'found' }
    ]);
    const result = findRecentlyRebooted(1, fromDate);
    expect(result.length).toBe(1);
    expect(result[0].name).toContain('found');
  });
});

```

А вот как можно было бы подойти к реализации каждой части CFRA с Jest:

| | |
|----------------|--|
| Очистка | <code>jest.resetAllMocks</code> |
| Создание фейка | <code>jest.mock() + [fake].mockImplementation()</code> |
| Запрос | Как обычно в начале файла |
| Действие | Как обычно |

Методы `jest.mock` и `jest.resetAllMocks` предназначены для создания фейка для модуля и замены фейковой реализации пустой. Обратите внимание: после `resetAllMocks` модуль все еще остается фейковым. Только его поведение сбрасывается к фейковой реализации по умолчанию. Если при вызове не указать, что должно возвращаться, вы будете получать странные ошибки.

С методом `FromModule` реализация по умолчанию заменяется функцией, которая возвращает фиксированные значения в каждом teste.

Для создания мока можно было бы воспользоваться `mockImplementationOnce()` вместо метода `fakeDataFromModule()`, но, на мой взгляд, при этом создаются слишком хрупкие тесты. Со стабами нас обычно не должно интересовать, сколько раз возвращаются фейковые значения. Если же количество вызовов для нас существенно, их следовало бы использовать как моки (данная тема рассматривается в главе 4).

П.4.3. Избегайте ручных моков Jest

В Jest предусмотрена концепция *ручных моков*, но лучше не использовать их, если без этого можно обойтись. Этот метод требует включения в ваши тесты особого каталога `__mocks__` с фиксированным кодом фейковых модулей, имена которых строятся по специальной схеме на основании имени модуля.

Такая схема будет работать, но, когда вы захотите управлять фейковыми данными, затраты на сопровождение будут слишком большими. Читабельность также сильно страдает, так как она требует слишком много лишней прокрутки и переключения между разными файлами для понимания теста. За дополнительной информацией о ручных моках обращайтесь к документации Jest: <https://jestjs.io/docs/en/manual-mocks.html>.

П.4.4. Создание стаба для модуля с Sinon.js

Для сравнения (и чтобы вы увидели, как паттерн CFRA воспроизводится с другими фреймворками) приведу реализацию того же теста с Sinon.js — фреймворком, предназначенным для создания стабов.

Листинг П.7. Создание стаба для модуля с Sinon.js

```
const sinon = require('sinon');
let dataModule;

const fakeDataFromModule = fakeData => {
  sinon.stub(dataModule, 'getAllMachines')
    .returns(fakeData);
};

const resetAndRequireModules = () => {
  jest.resetModules();
  dataModule = require('../my-data-module');
};

const requireAndCall_findRecentlyRebooted = (maxDays, someDate) => {
  const { findRecentlyRebooted } = require('../machine-scanner4');
  return findRecentlyRebooted(maxDays, someDate);
};

describe('4 sinon sandbox findRecentlyRebooted', () => {
  beforeEach(resetAndRequireModules);

  test('given no machines, returns empty results', () => {
    const someDate = new Date('01 01 2000');
    fakeDataFromModule([]);
  });
});
```

```
const result = requireAndCall_findRecentlyRebooted(2, someDate);
expect(result.length).toBe(0);
});
```

Ниже приведена карта паттерна CFRA с Sinon:

| | |
|-----------------------------|--|
| Очистка | Перед каждым тестом: jest.resetModules + повторное require для фейкового модуля |
| Создание фейка | Перед каждым тестом: sinon.stub(модуль, 'функция') .returns(fakeData) |
| Запрос (тестируемый модуль) | Перед активацией точки входа |
| Действие | После повторного запроса для тестируемого модуля |

П.4.5. Создание стаба для модуля с testdouble

Testdouble — еще один изолирующий фреймворк, который легко может использоваться для создания стабов. Благодаря рефакторингу, уже проведенному в предыдущих тестах, изменения в коде будут минимальными.

Листинг П.8. Создание стаба для модуля с testdouble

```
let td;

const resetAndRequireModules = () => {
  jest.resetModules();
  td = require('testdouble');
  require('testdouble-jest')(td, jest);
};

const fakeDataFromModule = fakeData => {
  td.replace('../my-data-module', {
    getAllMachines: () => fakeData
  });
};

const requireAndCall_findRecentlyRebooted = (maxDays, fromDate) => {
  const { findRecentlyRebooted } = require('../machine-scanner4');
  return findRecentlyRebooted(maxDays, fromDate);
};
```

318 Приложения. Манки-патчинг функций и модулей

Карта важнейших частей CFRA для testdouble:

| | |
|-----------------------------|--|
| Очистка | Перед каждым тестом: <code>jest.resetModules + require('testdouble'); require('testdouble-jest') (td, jest);</code> |
| Создание фейка | Перед каждым тестом: <code>Td.replace(модуль, объект-фейк)</code> |
| Запрос (тестируемый модуль) | Перед активацией точки входа |
| Действие | После повторного запроса для тестируемого модуля |

Реализация теста получается точно такой же, как в примере для Sinon. Мы также используем вариант `testdouble-jest`, так как он связан со средствами замены модулей Jest. Если бы мы использовали другой фреймворк тестирования, это было бы необязательно.

Эти методы *работают*, но я рекомендую держаться от них подальше, если только у вас нет другого выхода. Однако какой-либо другой выход почти всегда существует, многие из них были представлены в главе 3.

Рой Ошеров, Владимир Хориков

**Искусство юнит-тестирования
с примерами на JavaScript. 3-е межд. изд.**

Перевел с английского Е. Матвеев

| | |
|-------------------------|-----------------------------------|
| Руководитель дивизиона | <i>Ю. Сергиенко</i> |
| Руководитель проекта | <i>Ю. Сергиенко</i> |
| Ведущий редактор | <i>Е. Строганова</i> |
| Научный редактор | <i>Д. Колфилд</i> |
| Литературный редактор | <i>Ю. Широнина</i> |
| Художественный редактор | <i>В. Мостипан</i> |
| Корректоры | <i>М. Лауконен, Т. Никифорова</i> |
| Верстка | <i>Л. Соловьева</i> |

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 03.2025.

Наименование: книжная продукция.

Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 24.01.25. Формат 70×100/16. Бумага офсетная.
Усл. п. л. 25,800. Тираж 1000. Заказ 0000.

КРОК

СОЗДАЕМ НАСТОЯЩЕЕ,
ИНТЕГРИРУЕМ БУДУЩЕЕ



croc.ru

КРОК – технологический партнер с комплексной экспертизой в области построения и развития инфраструктуры, внедрения информационных систем, разработки программных решений и сервисной поддержки.

Центры компетенций КРОК фокусируются на ключевых отраслевых кластерах – промышленность, финансовый сектор, розничные продажи, муниципальное управление, спорт и культура.

Ежегодно сотни проектов КРОК становятся системообразующими для экономики и социально-культурной сферы.

