



Принципы юнит- тестирования

Владимир Хориков

Unit Testing: Principles, Practices, and Patterns

VLADIMIR KHORIKOV



MANNING
SHELTER ISLAND

Владимир Хориков

Принципы юнит- тестирования



Санкт-Петербург · Москва · Екатеринбург · Воронеж
Нижний Новгород · Ростов-на-Дону · Самара · Минск

2021

УДК: 004.415.53
ББК: 32.973.2-018-07
Х79

Хориков Владимир

Х79 Принципы юнит-тестирования. — СПб.: Питер, 2021. — 320 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1683-6

Юнит-тестирование — это процесс проверки отдельных модулей программы на корректность работы. Правильный подход к тестированию позволит максимизировать качество и скорость разработки проекта. Некачественные тесты, наоборот, могут нанести вред: нарушить работоспособность кода, увеличить количество ошибок, растянуть сроки и затраты. Грамотное внедрение юнит-тестирования — хорошее решение для развития проекта.

Научитесь разрабатывать тесты профессионального уровня, без ошибок автоматизировать процессы тестирования, а также интегрировать тестирование в жизненный цикл приложения. Со временем вы овладеете особым чутьем, присущим специалистам по тестированию. Как ни удивительно, практика написания хороших тестов способствует созданию более качественного кода.

В этой книге: универсальные рекомендации по оценке тестов; тестирование для выявления и исключения антипаттернов; рефакторинг тестов вместе с рабочим кодом; использование интеграционных тестов для проверки всей системы.

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

УДК: 004.415.53
ББК: 32.973.2-018-07

Права 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.

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 978-1617296277 англ.
ISBN 978-5-4461-1683-6

© 2019 by Manning Publications USA. All rights reserved
© Перевод на русский язык ООО Издательство «Питер», 2021
© Издание на русском языке,
оформление ООО Издательство «Питер», 2021
© Серия «Для профессионалов», 2021

Оглавление

Предисловие к русскому изданию	14
Предисловие к оригинальному изданию	15
Благодарности	16
О книге	17
Для кого написана эта книга.....	17
Структура книги.....	18
О коде.....	18
Форум для обсуждения книги	19
Об авторе	19
Иллюстрация на обложке	19
От издательства.....	20
Часть I. Общая картина.....	21
Глава 1. Цель юнит-тестирования.....	22
1.1. Текущее состояние дел в юнит-тестировании.....	23
1.2. Цель юнит-тестирования.....	24
1.2.1. В чем разница между плохими и хорошими тестами?	26

1.3. Использование метрик покрытия для оценки качества тестов	28
1.3.1. Метрика покрытия code coverage	28
1.3.2. Branch coverage	30
1.3.3. Проблемы с метриками покрытия.....	31
1.3.4. Процент покрытия как цель	34
1.4. Какими должны быть успешные тесты?	35
1.4.1. Интеграция в цикл разработки.....	35
1.4.2. Проверка только самых важных частей кода	35
1.4.3. Максимальная защита от багов при минимальных затратах на сопровождение.....	36
1.5. Что вы узнаете из книги	37
Итоги.....	38
Глава 2. Что такое юнит-тест?	40
2.1. Определение юнит-теста	40
2.1.1. Вопрос изоляции: лондонская школа	41
2.1.2. Вопрос изоляции: классический подход.....	47
2.2. Классическая и лондонская школы юнит-тестирования.....	50
2.2.1. Работа с зависимостями в классической и лондонской школах.....	51
2.3. Сравнение классической и лондонской школ юнит-тестирования	54
2.3.1. Юнит-тестирование одного класса за раз.....	55
2.3.2. Юнит-тестирование большого графа взаимосвязанных классов.....	56
2.3.3. Выявление точного местонахождения ошибки	57
2.3.4. Другие различия между классической и лондонской школами	57
2.4. Интеграционные тесты в двух школах	58
2.4.1. Сквозные (end-to-end) тесты как подмножество интеграционных тестов	60
Итоги.....	62
Глава 3. Анатомия юнит-теста.....	64
3.1. Структура юнит-теста	65
3.1.1. Паттерн AAA.....	65
3.1.2. Избегайте множественных секций arrange, act и assert	66
3.1.3. Избегайте команд if в тестах.....	67
3.1.4. Насколько большой должна быть каждая секция?	68

3.1.5. Сколько проверок должна содержать секция проверки?.....	70
3.1.6. Нужна ли завершающая (teardown) фаза?.....	71
3.1.7. Выделение тестируемой системы	71
3.1.8. Удаление комментариев «arrange/act/assert» из тестов	72
3.2. Фреймворк тестирования xUnit	72
3.3. Переиспользование тестовых данных между тестами	74
3.3.1. Сильная связность (high coupling) между тестами как антипаттерн.....	75
3.3.2. Использование конструкторов в тестах ухудшает читаемость	76
3.3.3. Более эффективный способ переиспользования тестовых данных	76
3.4. Именование юнит-тестов	78
3.4.1. Рекомендации по именованию юнит-тестов	80
3.4.2. Пример: переименование теста в соответствии с рекомендациями	80
3.5. Параметризованные тесты.....	82
3.5.1. Генерирование данных для параметризованных тестов	85
3.6. Использование библиотек для дальнейшего улучшения читаемости тестов	86
Итоги.....	88
Часть II. Обеспечение эффективной работы ваших тестов.....	89
Глава 4. Четыре аспекта хороших юнит-тестов	90
4.1. Четыре аспекта хороших юнит-тестов.....	91
4.1.1. Первый аспект: защита от багов	91
4.1.2. Второй аспект: устойчивость к рефакторингу	92
4.1.3. Что приводит к ложным срабатываниям?.....	94
4.1.4. Тестирование конечного результата вместо деталей имплементации	98
4.2. Связь между первыми двумя атрибутами.....	99
4.2.1. Максимизация точности тестов	100
4.2.2. Важность ложных и ложноотрицательных срабатываний: динамика.....	101
4.3. Третий и четвертый аспекты: быстрая обратная связь и простота поддержки	103
4.4. В поисках идеального теста.....	103
4.4.1. Возможно ли создать идеальный тест?.....	104

4.4.2. Крайний случай № 1: сквозные (end-to-end) тесты	105
4.4.3. Крайний случай № 2: тривиальные тесты	106
4.4.4. Крайний случай № 3: хрупкие тесты.....	106
4.4.5. В поисках идеального теста: результаты	108
4.5. Известные концепции автоматизации тестирования	111
4.5.1. Пирамида тестирования.....	111
4.5.2. Выбор между тестированием по принципу «черного ящика» и «белого ящика».....	114
Итоги.....	115
Глава 5. Моки и хрупкость тестов	117
5.1. Отличия моков от стабов	118
5.1.1. Разновидности тестовых заглушек	118
5.1.2. Мок-инструмент и мок — тестовая заглушка.....	120
5.1.3. Не проверяйте взаимодействия со стабами	121
5.1.4. Использование моков вместе со стабами.....	122
5.1.5. Связь моков и стабов с командами и запросами	123
5.2. Наблюдаемое поведение и детали имплементации	124
5.2.1. Наблюдаемое поведение — не то же самое, что публичный API....	125
5.2.2. Утечка деталей имплементации: пример с операцией.....	126
5.2.3. Хорошо спроектированный API и инкапсуляция	129
5.2.4. Утечка деталей имплементации: пример с состоянием	130
5.3. Связь между моками и хрупкостью тестов	132
5.3.1. Определение гексагональной архитектуры	132
5.3.2. Внутрисистемные и межсистемные взаимодействия.....	136
5.3.3. Внутрисистемные и межсистемные взаимодействия: пример	138
5.4. Еще раз о различиях между классической и лондонской школами юнит-тестирования	141
5.4.1. Не все внепроцессные зависимости должны заменяться моками	142
5.4.2. Использование моков для проверки поведения.....	144
Итоги.....	144
Глава 6. Стили юнит-тестирования	147
6.1. Три стиля юнит-тестирования.....	148
6.1.1. Проверка выходных данных.....	148

6.1.2. Проверка состояния	149
6.1.3. Проверка взаимодействий	150
6.2. Сравнение трех стилей юнит-тестирования.....	151
6.2.1. Сравнение стилей по метрикам защиты от багов и быстроте обратной связи	152
6.2.2. Сравнение стилей по метрике устойчивости к рефакторингу	152
6.2.3. Сравнение стилей по метрике простоты поддержки	153
6.2.4. Сравнение стилей: результаты.....	156
6.3. Функциональная архитектура	157
6.3.1. Что такое функциональное программирование?	157
6.3.2. Что такое функциональная архитектура?	161
6.3.3. Сравнение функциональных и гексагональных архитектур	163
6.4. Переход на функциональную архитектуру и тестирование выходных данных	164
6.4.1. Система аудита.....	165
6.4.2. Использование моков для отделения тестов от файловой системы...	167
6.4.3. Рефакторинг для перехода на функциональную архитектуру.....	170
6.4.4. Потенциальные будущие изменения.....	176
6.5. Недостатки функциональной архитектуры.....	177
6.5.1. Применимость функциональной архитектуры	177
6.5.2. Недостатки по быстродействию	179
6.5.3. Увеличение размера кодовой базы	179
Итоги.....	180
Глава 7. Рефакторинг для получения эффективных юнит-тестов.....	182
7.1. Определение кода для рефакторинга.....	183
7.1.1. Четыре типа кода.....	183
7.1.2. Использование паттерна «Простой объект» для разделения переусложненного кода.....	187
7.2. Рефакторинг для получения эффективных юнит-тестов	190
7.2.1. Знакомство с системой управления клиентами	190
7.2.2. Версия 1: преобразование неявных зависимостей в явные	192
7.2.3. Версия 2: уровень сервисов приложения.....	193
7.2.4. Версия 3: вынесение сложности из сервисов приложения.....	195
7.2.5. Версия 4: новый класс Company	197

7.3. Анализ оптимального покрытия юнит-тестов	200
7.3.1. Тестирование слоя предметной области и вспомогательного кода	200
7.3.2. Тестирование кода из трех других четвертей	201
7.3.3. Нужно ли тестировать предусловия?	202
7.4. Условная логика в контроллерах	203
7.4.1. Паттерн «CanExecute/Execute»	205
7.4.2. Использование доменных событий для отслеживания изменений доменной модели	208
7.5. Заключение	212
Итоги.....	214
Часть III. Интеграционное тестирование	217
Глава 8. Для чего нужно интеграционное тестирование?.....	218
8.1. Что такое интеграционный тест?.....	219
8.1.1. Роль интеграционных тестов	219
8.1.2. Снова о пирамиде тестирования	220
8.1.3. Интеграционное тестирование и принцип Fail Fast	222
8.2. Какие из внепроцессных зависимостей должны проверяться напрямую ...	224
8.2.1. Два типа внепроцессных зависимостей.....	224
8.2.2. Работа с управляемыми и неуправляемыми зависимостями	225
8.2.3. Что делать, если вы не можете использовать реальную базу данных в интеграционных тестах?	226
8.3. Интеграционное тестирование: пример.....	227
8.3.1. Какие сценарии тестировать?	228
8.3.2. Классификация базы данных и шины сообщений	229
8.3.3. Как насчет сквозного тестирования?	229
8.3.4. Интеграционное тестирование: первая версия	231
8.4. Использование интерфейсов для абстрагирования зависимостей	232
8.4.1. Интерфейсы и слабая связность	232
8.4.2. Зачем использовать интерфейсы для внепроцессных зависимостей?	233
8.4.3. Использование интерфейсов для внутрипроцессных зависимостей.....	235
8.5. Основные приемы интеграционного тестирования.....	235

8.5.1. Явное определение границ модели предметной области	235
8.5.2. Сокращение количества слоев.....	236
8.5.3. Исключение циклических зависимостей.....	237
8.5.4. Использование нескольких секций действий в тестах	240
8.6. Тестирование функциональности логирования.....	241
8.6.1. Нужно ли тестировать функциональность логирования?.....	241
8.6.2. Как тестировать функциональность логирования?	243
8.6.3. Какой объем логирования можно считать достаточным?	248
8.6.4. Как передавать экземпляры логеров?	249
8.7. Заключение	250
Итоги.....	250
Глава 9. Рекомендации при работе с моками.....	254
9.1. Достижение максимальной эффективности моков	254
9.1.1. Проверка взаимодействий на границах системы	257
9.1.2. Замена моков шпионами	261
9.1.3. Как насчет IDomainLogger?	263
9.2. Практики мокирования	263
9.2.1. Моки только для интеграционных тестов	264
9.2.2. Несколько моков на тест.....	264
9.2.3. Проверка количества вызовов	265
9.2.4. Используйте моки только для принадлежащих вам типов	265
Итоги.....	267
Глава 10. Тестирование базы данных	268
10.1. Предусловия для тестирования базы данных.....	269
10.1.1. Хранение базы данных в системе контроля версий	269
10.1.2. Справочные данные являются частью схемы базы данных.....	270
10.1.3. Отдельный экземпляр для каждого разработчика	271
10.1.4. Развёртывание базы данных на основе состояния и на основе миграций	271
10.2. Управление транзакциями.....	274
10.2.1. Управление транзакциями в рабочем коде	274
10.2.2. Управление транзакциями в интеграционных тестах.....	282

10.3. Жизненный цикл тестовых данных	284
10.3.1. Параллельное или последовательное выполнение тестов?	284
10.3.2. Очистка данных между запусками тестов	285
10.3.3. Не используйте базы данных в памяти	287
10.4. Переиспользование кода в секциях тестов	287
10.4.1. Переиспользование кода в секциях подготовки	288
10.4.2. Переиспользование кода в секциях действий	290
10.4.3. Переиспользование кода в секциях проверки	291
10.4.4. Не создает ли тест слишком много транзакций?	292
10.5. Типичные вопросы при тестировании баз данных	293
10.5.1. Нужно ли тестировать операции чтения?	294
10.5.2. Нужно ли тестировать репозитории?	294
10.6. Заключение	296
Итоги	297
Часть IV. Антипаттерны юнит-тестирования.....	299
Глава 11. Антипаттерны юнит-тестирования	300
11.1. Юнит-тестирование приватных методов	300
11.1.1. Приватные методы и хрупкость тестов	301
11.1.2. Приватные методы и недостаточное покрытие	301
11.1.3. Когда тестирование приватных методов допустимо	302
11.2. Раскрытие приватного состояния	304
11.3. Утечка доменных знаний в тесты	306
11.4. Загрязнение кода	307
11.5. Мокирование конкретных классов	310
11.6. Работа со временем	313
11.6.1. Время как неявный контекст	313
11.6.2. Время как явная зависимость	314
11.7. Заключение	315
Итоги	315

Посвящаю моей жене Нине

Предисловие к русскому изданию

Я помню, как начинал работать программистом в 2004 году в небольшой московской компании. В те времена никто не только не писал юнит-тесты, многие даже не знали о такой практике. Сейчас юнит-тестирование — неотъемлемая часть любого сколько-нибудь крупного проекта. Причем неважно, работаете ли вы в России или трудитесь на аутсорсе, навык написания хороших, легких в сопровождении тестов необходим всем.

Несмотря на такую востребованность, найти информацию о том, как именно писать такие юнит-тесты, непросто. Существует множество турниралов, где показывают, как дать на вход калькулятору два числа и проверить возвращаемое значение или как использовать мок-библиотеку, но это по сути все. Писать эффективные тесты часто приходится, учась на своих ошибках.

Цель этой книги — не учить пользоваться фреймворками юнит-тестирования или настраивать TeamCity для регулярного прогона тестов, этого материала в интернете предостаточно. Ее цель — собрать воедино всю информацию о написании эффективных, простых в поддержке тестов. Большую часть этой информации можно узнать лишь опытным способом, набив по пути много шишек. Эта книга позволит вам пропустить стадию шишек и перейти сразу к плодам этих проб и ошибок, на что у меня ушло около 10 лет.

Предисловие к оригинальному изданию

Помню свой первый проект, в котором применил юнит-тестирование. Все прошло относительно хорошо, но после того как он был закончен, я взглянул на тесты и подумал, что многие из них были напрасной тратой времени. Большинство моих юнит-тестов тратило изрядную долю времени на настройку ожиданий и плетение сложной паутины зависимостей — и все это для проверки правильности всего трех строк кода в моем контроллере. Я не мог сформулировать, что именно не так с моими тестами, но было стойкое ощущение того, что это не нормально. К счастью, я не отказался от юнит-тестирования и продолжал применять его в последующих проектах. Тем не менее я чувствовал все большее и большее несогласие с общепринятыми (на тот момент) практиками юнит-тестирования. За эти годы я часто писал о юнит-тестировании. В этих статьях мне наконец удалось сформулировать, что пошло не так с моими первыми тестами, и обобщить эти знания для более широких областей юнит-тестирования. Эта книга — результат моих исследований, проб и ошибок, тщательно переработанных, уточненных и собранных в одном месте.

Я получил математическое образование и твердо считаю, что рекомендации в программировании, как и теоремы в математике, должны выводиться из фундаментальных принципов. Я постарался структурировать эту книгу аналогичным образом: не делать поспешных выводов и не выступать с необоснованными заявлениями, а начать с чистого листа: установить фундаментальные принципы и выводить все рекомендации по юнит-тестированию из этих принципов, «с нуля». Интересно, что после установления аксиоматики рекомендации часто выводятся сами собой, как простые следствия.

Юнит-тестирование постепенно становится обязательным требованием для программных проектов, и эта книга даст вам все необходимое для построения хороших и простых в сопровождении тестов.

Благодарности

На написание книги ушло много времени. Хотя я и был к этому готов, работы все равно оказалось намного больше, чем я мог себе представить.

Хочу поблагодарить многих людей: Сэма Зейдла (Sam Zaydel), Alessandro Кампейса (Alessandro Campeis), Фрэнсис Буран (Frances Buran), Тиффани Тейлор (Tiffany Taylor) и особенно Марину Майклз (Marina Michaels), чье бесценное мнение помогло мне поддерживать качество материала на высшем уровне и попутно улучшило мои писательские навыки. Также спасибо остальным сотрудникам Manning, работавшим над книгой в процессе выпуска и оставшимся незамеченными.

Также хочу поблагодарить научных редакторов, которые не пожалели времени на чтение моей рукописи в различных фазах работы и предоставили полезнейшую обратную связь: Аарона Бартона (Aaron Barton), Alessandro Кампейса (Alessandro Campeis), Конора Редмонда (Conor Redmond), Дрор Хелпер (Dror Helper), Грега Райта (Greg Wright), Хемант Конегу (Hemant Koneri), Джереми Ланге (Jeremy Lange), Хорхе Эзекиля Бо (Jorge Ezequiel Bo), Джорта Роденбурга (Jort Rodenburg), Марка Ненадова (Mark Nenadov), Марко Умека (Marko Umek), Маркуса Мецкера (Markus Matzker), Шрихари Шридхарана (Srihari Sridharan), Стивена Джона Уорнетта (Stephen John Warnett), Суманта Тамбе (Sumant Tambe), Тима ван Дьюрзена (Tim van Deurzen) и Владимира Купцова (Vladimir Kuptsov).

Но больше всего хочу поблагодарить свою жену Нину, которая поддерживала меня на протяжении всей работы над книгой.

О книге

В книге «Принципы юнит-тестирования» подробно рассматриваются рекомендации, паттерны и антипаттерны, встречающиеся в области юнит-тестирования. После чтения этой книги вы будете знать все необходимое для того, чтобы стать экспертом в области создания успешных проектов — проектов, которые легко расширять и сопровождать благодаря хорошим тестам.

Для кого написана эта книга

У большинства сетевых и печатных ресурсов имеется один недостаток: они подробно излагают основы юнит-тестирования, но практически не выходят за эти рамки. Такие ресурсы могут быть очень ценными, однако обучение на этом не заканчивается. Существует и следующий уровень: умение не просто писать тесты, но делать это так, чтобы ваши усилия приносили максимальную отдачу. К сожалению, многим людям приходится самим разбираться, как выйти на этот уровень, часто методом проб и ошибок. Эта книга поможет вам в этом. В ней приводится точное определение того, что собой представляет качественный тест. Это определение формирует единую систему отсчета, которая поможет вам взглянуть на многие из ваших тестов в новом свете и увидеть, какие из них работают на пользу проекта, а какие следует отрефакторить или вообще удалить.

Если у вас мало опыта в юнит-тестировании, из этой книги вы многое узнаете. Опытный программист, скорее всего, уже понимает некоторые идеи, изложенные здесь. Книга поможет ему осознать, почему приемы и практики, которыми он пользовался все это время, настолько полезны. И не стоит недооценивать этот навык: умение четко донести свои идеи коллегам чрезвычайно полезно.

Структура книги

Одиннадцать глав этой книги разделены на четыре части. В части I изложены основы юнит-тестирования, а также напоминаются наиболее общие практики юнит-тестирования:

- Глава 1 показывает цели юнит-тестирования, в ней приводится краткий обзор того, как отличить хороший тест от плохого.
- В главе 2 анализируется определение юнит-тестирования и обсуждаются две основные школы в области юнит-тестирования.
- Глава 3 рассматривает некоторые базовые вопросы — такие как структура юнит-тестов, переиспользование тестовых данных и параметризация тестов.

В части II мы перейдем к сути дела — вы увидите, какими свойствами должен обладать хороший юнит-тест, а также узнаете, как провести рефакторинг тестов для повышения их качества:

- В главе 4 определяются четыре характеристики, по которым можно оценить качество теста, а также предоставляется общая система координат, которая используется на протяжении всей книги.
- В главе 5 объясняется, для чего нужны моки (*mocks*), и анализируется их связь с хрупкостью тестов.
- В главе 6 рассматриваются три стиля юнит-тестирования и то, какой из этих стилей производит тесты лучшего качества и почему.
- Глава 7 показывает, как провести рефакторинг раздутых, чрезмерно усложненных тестов и получить тесты, сочетающие в себе максимальную эффективность с минимальными затратами на сопровождение.

В части III изучаются вопросы интеграционного тестирования:

- В главе 8 рассматривается интеграционное тестирование в целом, его достоинства и недостатки.
- В главе 9 обсуждаются моки (*mocks*) и как работать с ними так, чтобы максимально повысить эффективность ваших тестов.
- В главе 10 рассматривается работа с реляционными базами данных в тестах.

В главе 11 части IV представлены стандартные антипаттерны юнит-тестирования.

О коде

Примеры кода написаны на C#, но те аспекты, которые они демонстрируют, применимы к любому объектно-ориентированному языку (например, Java или C++). Я старался не пользоваться специфическими языковыми возможностями C#.

и сделать код примеров по возможности простым, чтобы вы легко разобрались в нем. Весь код примеров можно скачать по адресу www.manning.com/books/unit-testing.

Форум для обсуждения книги

Приобретая книгу, вы получаете бесплатный доступ к закрытому веб-форуму Manning, на котором можно публиковать комментарии по поводу книги, задавать технические вопросы и получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, откройте страницу <https://livebook.manning.com/#!/book/unit-testing/discussion>. За информацией о форумах Manning и правилах поведения обращайтесь по адресу <https://livebook.manning.com/#!/discussion>.

В рамках своих обязательств перед читателями издательством Manning предоставляет ресурс для проведения осмысленного диалога между читателями и автором. Эти обязательства не подразумевают конкретный вклад со стороны автора, участие которого в работе форума остается добровольным (и неоплачиваемым). Задавайте автору интересные вопросы, чтобы он не терял интереса к происходящему! Форум и архивы предшествующих обсуждений доступны на веб-сайте издателя, пока книга находится в печати.

Другие сетевые ресурсы

- Мой блог находится по адресу EnterpriseCraftsmanship.com.
- У меня также имеется онлайн-курс по юнит-тестированию (в данный момент находится в разработке), на который можно записаться по адресу UnitTestingCourse.com.

Об авторе

Владимир Хориков — разработчик, Microsoft MVP и автор на платформе Pluralsight. Профессионально занимается разработкой программного обеспечения более 15 лет, а также обучением команд тонкостям юнит-тестирования. За последние годы Владимир опубликовал несколько популярных серий в блогах, а также онлайн-курс на тему юнит-тестирования. Главное достоинство его стиля обучения, которое часто отмечают студенты, — сильная теоретическая подготовка, которая затем используется на практике.

Иллюстрация на обложке

Иллюстрация, помещенная на обложку второго издания книги и озаглавленная «Esthinienne», была позаимствована из изданного в 1788 г. каталога национальных

костюмов Жака Грассе де Сен-Совера (1757–1810) «Costumes Civils Actuels de Tous les Peuples Connus». Каждая иллюстрация нарисована и раскрашена от руки. Иллюстрации из каталога Грассе де Сен-Совера напоминают о культурных различиях между городами и весями мира, имевших место почти две тысячи лет назад. Люди, проживавшие в изолированных друг от друга регионах, говорили на разных языках и диалектах. По одежде человека можно было определить, в каком городе, поселке или поселении он проживает.

С тех пор дресс-код сильно изменился, да и различия между разными регионами стали не столь выраженным. В наше время довольно трудно узнать жителей разных континентов, не говоря уже о жителях разных городов или регионов. Возможно, мы отказались от культурных различий в пользу более разнообразной личной жизни — и конечно, в пользу более разнообразной и стремительной технологической жизни.

Сейчас, когда все компьютерные книги похожи друг на друга, издательство Manning стремится к разнообразию и помещает на обложки книг иллюстрации, показывающие особенности жизни в разных странах мира два века назад.

От издательства

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

Мы будем рады узнать ваше мнение!

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

Часть I

Общая картина

Эта часть книги вводит читателя в курс текущего состояния дел в области юнит-тестирования. В главе 1 я расскажу о цели юнит-тестирования и покажу, как отличить хороший тест от плохого. Мы поговорим о метриках тестового покрытия и обсудим свойства хорошего юнит-теста.

В главе 2 будет приведено определение юнит-теста. Незначительное на первый взгляд расхождение в этом определении привело к формированию двух школ юнит-тестирования, которые будут описаны в этой главе. Глава 3 — это памятка по некоторым базовым темам, таким как структура юнит-тестов, переиспользование тестовых данных и параметризация тестов.

1

Цель юнит-тестирования

В этой главе:

- ✓ Состояние дел в юнит-тестировании.
- ✓ Цель юнит-тестирования.
- ✓ Последствия от написания плохих тестов.
- ✓ Использование метрик тестового покрытия для оценки качества тестов.
- ✓ Атрибуты успешных тестов.

Изучение юнит-тестирования не заканчивается на освоении его технических сторон: тестового фреймворка, библиотеки моков и т. д. Юнит-тестирование не сводится к простому написанию тестов. Важно стремиться к тому, чтобы свести к минимуму усилия, потраченные на написание тестов, и максимизировать преимущества, которые они приносят. Совместить эти две задачи не так просто.

Наблюдать за проектами, добившимися заветного баланса, одно удовольствие: они развиваются без лишних усилий, не требуют особого сопровождения и быстро адаптируются к постоянно изменяющимся потребностям заказчиков. С другой стороны, наблюдать за проектами, которые не справились с этой задачей, крайне мучительно. Несмотря на все усилия и впечатляющее количество юнит-тестов, такие проекты развиваются медленно, содержат множество багов и требуют больших затрат на сопровождение.

Существуют различные методы юнит-тестирования. Одни дают отличные результаты и помогают поддерживать качество кода на должном уровне. Другие с этим не справляются: полученные тесты не приносят особой пользы, часто ломаются и требуют значительных усилий при сопровождении.

Эта книга поможет вам отличать плохие методы юнит-тестирования от хороших. Вы узнаете, как анализировать эффективность ваших тестов и применить подходящие методы тестирования в вашей конкретной ситуации. Также вы научитесь обходить распространенные антипаттерны — паттерны, который на первый взгляд выглядят разумно, но приводят к проблемам в будущем.

Но начнем с азов. В этой главе приводится краткий обзор состояния дел в юнит-тестировании, описывается цель написания тестов, а также дается представление о том, что собой представляют успешные тесты.

1.1. Текущее состояние дел в юнит-тестировании

За два последних десятилетия программная индустрия начала постепенно практиковать юнит-тестирование. Во многих компаниях эти практики уже считаются обязательными — многие программисты пишут юнит-тесты и понимают их важность. Разногласий относительно того, нужно ли заниматься юнит-тестированием, уже нет.

При разработке корпоративных приложений практически каждый проект включает какое-то количество юнит-тестов. Соотношение между рабочим и тестовым кодом обычно лежит в диапазоне от 1:1 до 1:3 (на каждую строку рабочего кода приходится от одной до трех строк тестового кода). Иногда это соотношение достигает существенно большего значения — вплоть до 1:10.

Но как и все новые технологии, юнит-тестирование продолжает развиваться. С вопроса «нужно ли писать юнит тесты?» обсуждение перешло в другую плоскость: как писать хорошие юнит-тесты? Именно в этой области кроются основные разногласия.

Результаты этих разногласий проявляются в программных проектах. Многие проекты содержат автоматизированные тесты, но они зачастую не приносят результатов, на которые надеются разработчики: сопровождение проектов и разработка в них нового функционала все так же требуют значительных усилий, а в уже написанном функционале постоянно появляются новые ошибки. Юнит-тесты, которые вроде бы должны помогать, никак не способствуют решению этих проблем. Иногда они даже усугубляют ситуацию.

Это плачевная ситуация, и она часто возникает из-за того, что юнит-тесты неправляются со своей задачей. Различия между хорошими и плохими тестами не ограничиваются вкусами или личными предпочтениями. На практике эти различия влияют на весь проект — они могут либо помочь вам успешно завершить проект, либо привести к его провалу.

Важно понимать, какими качествами должен обладать хороший юнит-тест. И тем не менее информацию на эту тему найти довольно сложно. В интернете существуют

разрозненные статьи и выступления с конференций, но я еще не видел ни одного исчерпывающего материала по этой теме.

Ситуация с книгами ненамного лучше; многие из них сосредоточены на основах юнит-тестирования, но не выходят за эти рамки. Конечно, такие книги тоже полезны, особенно если вы только начинаете осваивать юнит-тестирование. Но обучение не заканчивается на основах. Важно не просто писать тесты, но делать это так, чтобы усилия приносили максимальную отдачу.

ЧТО ТАКОЕ «КОРПОРАТИВНОЕ ПРИЛОЖЕНИЕ»?

Корпоративным (enterprise) называется приложение, предназначенное для автоматизации внутренних процессов компании. Существует много разновидностей корпоративных приложений, но обычно оно обладает следующими характеристиками:

- ✓ высокая сложность бизнес-логики;
 - ✓ большой срок жизни проекта;
 - ✓ умеренные объемы данных;
 - ✓ низкие или средние требования к быстродействию.
-

Эта книга поможет вам в этом. В ней приводится точное, исчерпывающее определение качественного юнит-теста. Вы увидите, как это определение применяется к практическим примерам из реальной жизни.

Книга принесет наибольшую пользу, если вы занимаетесь разработкой корпоративных приложений, но основные идеи применимы в любом программном проекте.

1.2. Цель юнит-тестирования

Прежде чем углубляться в тему юнит-тестирования, давайте рассмотрим, для чего вообще нужно юнит-тестирование и какой цели оно помогает добиться. Считается, что юнит-тестирование улучшает качество кода проекта. И это правда: необходимость написания юнит-тестов обычно приводит к улучшению качества кода. Но это не главная цель юнит-тестирования, а всего лишь приятный побочный эффект.

Какова же тогда цель юнит-тестирования? Его цель — обеспечение *стабильного* роста программного проекта. Ключевым словом здесь является «стабильный». В начале жизни проекта развивать его довольно просто. Намного сложнее поддерживать это развитие с прошествием времени.

На рис. 1.1 изображена динамика роста типичного проекта без тестов. Все начинается быстро, потому что ничего вас не тормозит. Еще не приняты неудачные архитектурные решения; еще нет существующего кода, который необходимо прорабатывать и поддерживать. Однако с течением времени вам приходится тратить все больше времени, чтобы написать тот же по объему функционал, что и в начале

проекта. Со временем скорость разработки существенно замедляется — иногда даже до состояния, в котором проект вообще перестает двигаться вперед.

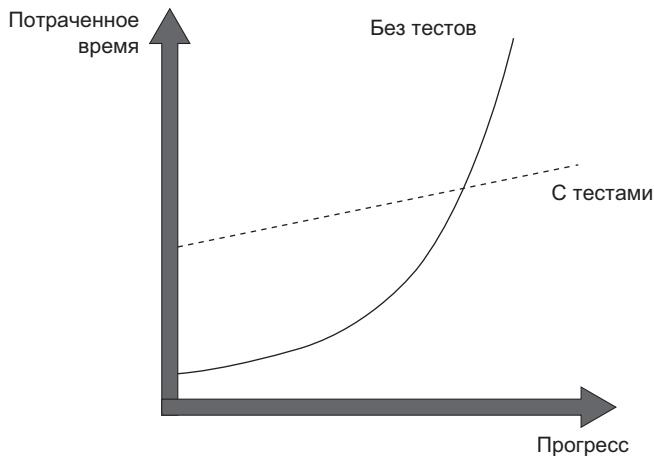


Рис. 1.1. Различия в динамике роста между проектами с тестами и без.
Проект без тестов быстро стартует, но и быстро замедляется до состояния,
в котором становится трудно двигаться вперед

Такое снижение скорости разработки называется *программная энтропия (software entropy)*. Энтропия (мера беспорядка в системе) — математическая и научная концепция, также применимая к программным системам. (Если вас интересует математическая и научная сторона энтропии, обращайтесь к описанию второго закона термодинамики.)

СВЯЗЬ МЕЖДУ ЮНИТ-ТЕСТИРОВАНИЕМ И СТРУКТУРОЙ КОДА

Сама возможность покрытия кода тестами — хороший критерий определения качества этого кода, но он работает только в одном направлении. Это хороший *негативный* признак — он выявляет низкокачественный код с относительно высокой точностью. Если вдруг обнаружится, что код трудно протестировать, это верный признак того, что код нуждается в улучшении. Плохое качество обычно проявляется в *сильной связности (tight coupling)* кода; это означает, что части кода недостаточно четко изолированы друг от друга, что в свою очередь создает сложности с их раздельным тестированием.

Но, к сожалению, возможность покрытия кода тестами является плохим *позитивным* признаком. Тот факт, что код проекта легко тестируется, еще не означает, что этот код написан хорошо. Качество кода может быть плохим даже в том случае, если он не страдает *сильной связностью*.

В программировании энтропия проявляется в форме ухудшения качества кода. Каждый раз, когда вы что-то изменяете в коде проекта, увеличивается степень беспорядка

в нем — его энтропия. Если не принять должных мер (например, постоянной чистки и рефакторинга), код постепенно усложняется и дезорганизуется. Исправление одной ошибки приводит к появлению новых ошибок, а изменение в одной части проекта нарушает работоспособность в нескольких других — возникает своего рода «эффект домино». Со временем код становится ненадежным. И что еще хуже, его становится все труднее вернуть в стабильное состояние.

Тесты помогают справиться с этой тенденцией. Они становятся своего рода «подушкой безопасности» — средством, которое обеспечивает защиту против большинства регрессий. Тесты помогают удостовериться в том, что существующая функциональность работает даже после разработки новой функциональности или рефакторинга кода.

ОПРЕДЕЛЕНИЕ

Термин «регрессия» означает, что некоторая функциональность перестает работать после определенного события (обычно внесения изменений в код). Термины «регрессия», «программная ошибка» и «баг» — синонимы.

Недостаток юнит-тестирования заключается в том, что тесты требуют начальных вложений, и иногда весьма значительных. Но в долгосрочной перспективе они окупаются, позволяя проекту расти на более поздних стадиях. Разработка большинства нетривиального программного обеспечения без помощи тестов практически невозможна.

1.2.1. В чем разница между плохими и хорошими тестами?

Хотя юнит-тесты помогают развитию проекта, просто писать тесты недостаточно. Плохо написанные тесты не меняют общей картины.

Как видно из рис. 1.2, плохие тесты на первых порах помогают замедлить ухудшение качества кода: уменьшение скорости разработки идет медленнее по сравнению с ситуацией, в которой тестов нет вообще. Однако это не меняет общей картины. Возможно, такому проекту понадобится больше времени для того, чтобы войти в фазу стагнации, но стагнация все равно неизбежна.

Не все тесты одинаково полезны. Некоторые из них вносят большой вклад в качество программного продукта. Другие только замедляют проект: дают много ложных срабатываний, не помогают выявлять баги, работают медленно и создают сложности с сопровождением. Многие компании пишут тесты без четкого понимания того, способствуют ли они развитию проекта.

Невозможно добиться цели юнит-тестирования, просто добавив в проект больше тестов. Необходимо учитывать как пользу этих тестов, так и затраты на их сопровождение. Составляющая затрат на сопровождение определяется количеством времени, ушедшего на:

- рефакторинг теста при рефакторинге нижележащего кода;
- выполнение теста при каждом изменении кода;
- отвлечение на ложные срабатывания теста;
- затраты на чтение теста при попытке понять, как работает нижележащий код.

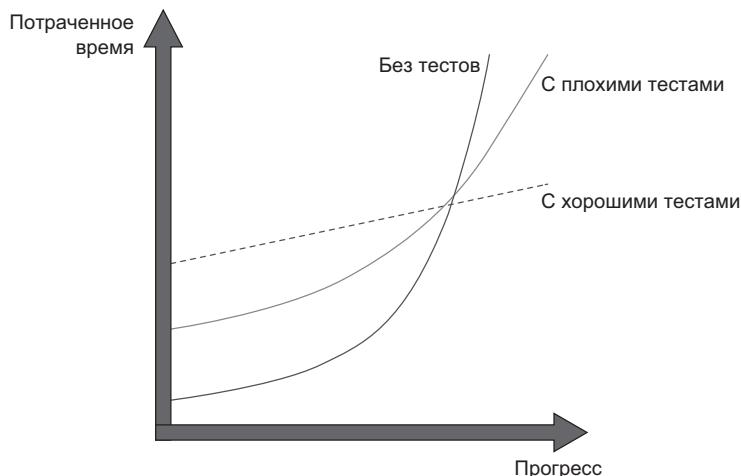


Рис. 1.2. Различия в динамике роста между проектами с плохими и хорошими тестами.
Проект с плохо написанными тестами в начальной стадии проявляет свойства проекта с хорошими тестами, но со временем все равно попадает в фазу стагнации

Легко создать тесты, общая польза которых близка к нулю или даже отрицательна из-за высоких затрат на сопровождение. Чтобы сделать возможным стабильный рост проекта, необходимо сосредоточиться исключительно на тестах с высоким качеством — только такие тесты стоят того, чтобы включать их в ваш проект.

Очень важно научиться отличать хорошие юнит-тесты от плохих. Эта тема рассматривается в главе 4.

ОСНОВНОЙ (РАБОЧИЙ) И ТЕСТОВЫЙ КОД

Люди часто думают, что основной (рабочий) код (*production code*) и тестовый код (*test code*) — не одно и то же. Предполагается, что тесты, в отличие от основного кода, не несут затрат на сопровождение. Вследствие этого люди часто полагают, что чем больше тестов, тем лучше. Тем не менее это не так. Код — обязательство, а не актив (*liability, not an asset*). Чем больше кода вы пишете, тем больше вы оставляете возможностей для появления потенциальных ошибок и тем выше будут затраты на сопровождение проекта. Лучше всего писать проекты, используя минимальное количество кода.

Тесты — это тоже код. Их следует рассматривать как часть кодовой базы, предназначенную для решения конкретной проблемы: обеспечения правильности приложения. Юнит-тесты, как и любой другой код, также подвержены ошибкам и требуют сопровождения.

1.3. Использование метрик покрытия для оценки качества тестов

В этом разделе речь пойдет о двух самых популярных метриках покрытия — code coverage и branch coverage: о том, как их вычислять, как они используются и какие проблемы с ними связаны. Я покажу, почему программистам не стоит ставить цель достичь какого-то конкретного процента тестового покрытия и почему тестовое покрытие само по себе не может служить критерием качества тестов.

ОПРЕДЕЛЕНИЕ

Метрика покрытия (coverage metric) показывает, какая доля исходного кода была выполнена хотя бы одним тестом — от 0 до 100 %.

Существуют различные типы метрик покрытия, которые используются для оценки качества тестов. Часто считается, что чем выше процент покрытия, тем лучше.

К сожалению, все не так просто. Хотя процент покрытия и предоставляет собой ценную обратную связь, он не может использоваться для оценки *качества* тестов. Ситуация здесь такая же, как с возможностью покрыть код проекта юнит-тестами: процент покрытия служит хорошим негативным признаком, но плохим позитивным.

Если покрытие слишком мало — допустим, всего 10 % — это хороший признак того, что тестов слишком мало. Однако обратное неверно: даже 100 %-ное покрытие еще не гарантирует хорошего качества тестов. Тесты, обеспечивающие высокое покрытие, тем не менее могут быть плохого качества.

Я уже упоминал, почему это так: нельзя просто добавить в проект случайные тесты и надеяться на то, что они помогут вам поддерживать качество этого проекта. Но давайте рассмотрим метрики тестового покрытия более подробно.

1.3.1. Метрика покрытия code coverage

Первая и наиболее часто используемая метрика покрытия — code coverage, также известная как test coverage (рис. 1.3). Эта метрика равна отношению количества строк кода, выполняемых по крайней мере одним тестом, к общему количеству строк в основном коде проекта.

$$\text{Code coverage (test coverage)} = \frac{\text{Количество выполненных строк кода}}{\text{Общее количество строк кода}}$$

Рис. 1.3. Code coverage вычисляется как отношение количества строк кода, выполняемых тестами, к общему количеству строк в основном коде проекта

Пример поможет вам лучше понять, как вычисляется эта метрика. В листинге 1.1 показан метод `IsStringLong` и тест, который покрывает его код. Метод определяет, является ли строка, переданная во входном параметре, длинной (в данном случае «длинной» считается любая строка, длина которой превышает 5 символов). Тест выполняет метод со строкой "abc" и проверяет, является ли эта строка длинной.

Листинг 1.1. Пример метода с частичным покрытием

```
public static bool IsStringLong(string input)
{
    if (input.Length > 5)
        return true;
    return false;
}

public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

The diagram illustrates the code coverage for the `IsStringLong` method. It shows two main sections: the method body and the test code. In the method body, the `if` block is labeled 'Покрыто тестом' (Covered by test) because it contains the `return true;` statement, which is executed by the test. The `return false;` statement is labeled 'Не покрыто тестом' (Not covered by test). In the test code, the entire block is labeled 'Покрыто тестом' (Covered by test) because it calls the method and asserts its result.

Покрытие в этом примере вычисляется легко. Общее количество строк в методе равно 5 (фигурные скобки тоже считаются). Количество строк, выполняемых в teste, равно 4 — тест проходит все строки кода, кроме команды `return true;`. Таким образом, покрытие равно $4/5 = 0,8 = 80\%$.

Что будет, если отрефакторить этот метод и убрать избыточную команду `if`?

```
public static bool IsStringLong(string input)
{
    return input.Length > 5;
}

public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

Изменился ли процент покрытия? Да, изменился. Так как тест теперь выполняет все три строки кода (команда `return` и две фигурные скобки), покрытие кода увеличилось до 100 %.

Но улучшилось ли качество тестов с таким рефакторингом? Конечно же, нет. Я просто переставил код внутри метода. Тест по-прежнему проверяет то же количество ветвлений в коде.

Этот простой пример показывает, как легко подтасовать процент покрытия. Чем компактнее ваш код, тем лучше становится этот процент, потому что в нем учитывается

только количество строк. В то же время попытки втиснуть больше кода в меньший объем не изменяют общую эффективность тестов.

1.3.2. Branch coverage

Другая метрика покрытия называется *branch coverage* (*покрытием ветвей*). Branch coverage показывает более точные результаты, чем code coverage. Вместо того чтобы использовать количество строк кода, эта метрика ориентируется на управляющие структуры — такие как команды `if` и `switch`. Она показывает, какое количество таких управляющих структур обходится по крайней мере одним тестом в проекте (рис. 1.4).

$$\text{Branch coverage} = \frac{\text{Количество покрытых ветвей}}{\text{Общее количество ветвей}}$$

Рис. 1.4. Branch coverage вычисляется как отношение количества ветвей кода, выполненных хотя бы одним тестом, к общему количеству ветвей в коде

Чтобы вычислить метрику branch coverage, необходимо подсчитать все возможные ветви (*branches*) в коде и посмотреть, сколько из них выполняются тестами. Вернемся к предыдущему примеру:

```
public static bool IsStringLong(string input)
{
    return input.Length > 5;
}

public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

Метод `IsStringLong` содержит две ветви: одна для ситуации, в которой длина строкового аргумента превышает пять символов, и другая для строк, длина которых менее или равна 5 символам. Тест покрывает только одну из этих ветвей, поэтому метрика покрытия составляет $1/2 = 0,5 = 50\%$. При этом неважно, какое представление будет выбрано для тестируемого кода — будете ли вы использовать команду `if`, как прежде, или выберете более короткую запись. Метрика branch coverage принимает во внимание только количество ветвей; она не учитывает, сколько строк кода понадобилось для реализации этих ветвей.

Рис. 1.5 показывает, как можно визуализировать эту метрику. Все возможные ветви в тестируемом коде представляются в виде графа, и вы проверяете, сколько из них были проидены тестами. В `IsStringLong` таких путей два, а тест отрабатывает только один из них.

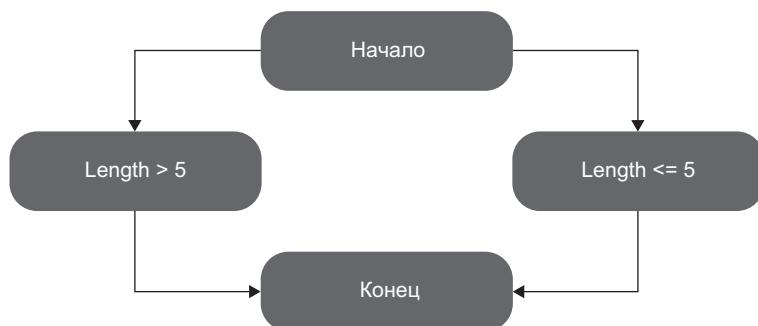


Рис. 1.5. Метод `IsStringLong` представлен в виде графа возможных путей выполнения кода. Тест покрывает только один из двух путей, обеспечивая таким образом 50%-ное покрытие

1.3.3. Проблемы с метриками покрытия

Хотя метрика `branch coverage` дает результаты лучше, чем метрика `code coverage`, вы все равно не сможете положиться на эту метрику для определения качества тестов по двум причинам:

- Невозможно гарантировать, что тест проверяет все компоненты результата работы тестируемой системы.
- Ни одна метрика покрытия не может учитывать ветвления кода во внешних библиотеках.

Рассмотрим каждую из этих причин подробнее.

Невозможно гарантировать, что тест проверяет все компоненты результата работы тестируемой системы

Чтобы код не просто отработал, а был протестирован, ваши юнит-тесты должны содержать подходящие проверки. Иначе говоря, необходимо проверить результат работы тестируемой системы. Более того, этот результат может состоять из нескольких компонентов, и чтобы метрики покрытия имели смысл, необходимо проверить все эти компоненты.

В листинге 1.2 приведена другая версия метода `IsStringLong`, которая записывает последний результат в свойство `WasLastStringLong`.

Листинг 1.2. Версия `IsStringLong` с сохранением последнего результата

```

public static bool WasLastStringLong { get; private set; }

public static bool IsStringLong(string input)
{
    bool result = input.Length > 5;           | Первый результат
    WasLastStringLong = result;               ←
    return result;                          ← Второй результат
}
  
```

```
public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);           ← Тест проверяет только второй результат
}
```

Теперь метод `IsStringLong` имеет два результата: явный, закодированный возвращаемым значением, и неявный, которым является новое значение свойства. И хотя второй, косвенный результат не проверяется, метрики покрытия демонстрируют те же результаты: 100 % для code coverage, 50 % для branch coverage. Как видите, метрики покрытия не гарантируют, что код реально тестируется — только то, что он выполнялся в какой-то момент.

Крайним выражением этой ситуации с частично тестируемыми результатами является тестирование без проверок (*assertion-free testing*): когда вы пишете тесты, которые вообще не содержат никаких проверочных команд. В листинге 1.3 приведен пример тестирования без проверок.

Листинг 1.3. Тест без проверок всегда проходит

```
public void Test()
{
    bool result1 = IsStringLong("abc");   ← Возвращает true
    bool result2 = IsStringLong("abcdef"); ← Возвращает false
}
```

В этом teste обе метрики — как code coverage, так и branch coverage — достигают 100 %. В то же время этот тест совершенно бесполезен, поскольку он ничего не проверяет.

Но допустим, вы тщательно проверяете каждый результат тестируемого кода. Создаст ли это (в сочетании с использованием branch coverage вместо code coverage) надежный механизм, который может использоваться для определения качества тестов? К сожалению, нет.

Ни одна метрика покрытия не может учитывать ветвления кода во внешних библиотеках, что является второй проблемой в использовании этих метрик. Возьмем следующий пример:

```
public static int Parse(string input)
{
    return int.Parse(input);
}

public void Test()
{
    int result = Parse("5");
    Assert.Equal(5, result);
}
```

Метрика branch coverage показывает 100 %, и при этом тест проверяет все составляющие результата метода. Такая составляющая здесь всего одна — возвращаемое

значение. В то же время такой тест совершенно не является исчерпывающим: он не учитывает ветвления, через которые может проходить метод `.NET Framework int.Parse`. В то же время даже такой простой метод может содержать большое количество ветвлений, как видно из рис. 1.6.

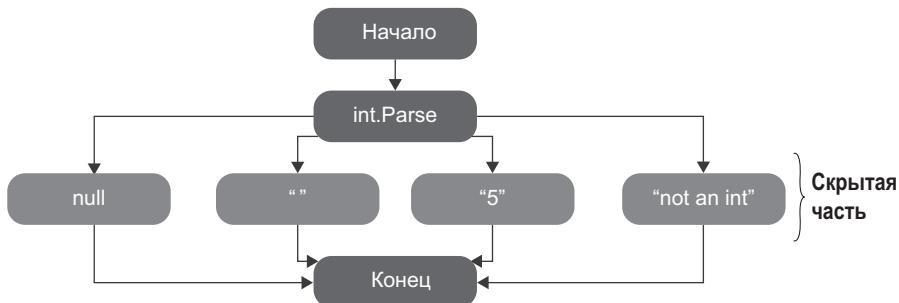


Рис. 1.6. Скрытые пути во внешних библиотеках. Метрики покрытия не видят, сколько существует таких путей и сколько из них будет отработано вашими тестами

ИСТОРИЯ С ПОЛЕЙ

Тестирование без проверок может показаться довольно глупой идеей, но такое случается на практике.

Много лет назад я работал в компании, в которой руководство установило требование стопроцентного покрытия для каждого проекта. За этой инициативой стояли благие намерения. История происходила в те дни, когда юнит-тестирование еще не получило такого широкого распространения, как сейчас. Лишь немногие программисты в организации писали тесты, еще меньше делали это на постоянной основе.

Группа разработчиков отправилась на конференцию, на которой многие доклады были посвящены юнит-тестированию. После возвращения они решили применить свои новые знания на практике. Руководство их поддержало, и так начался переход на усовершенствованные методы программирования. Проводились внутренние презентации, устанавливались новые утилиты. Но что самое важное, было объявлено новое правило: все группы разработчиков должны были сосредоточиться на написании тестов до тех пор, пока не будет достигнуто 100%-ное покрытие. После достижения этой цели любой новый код, который снижал эту метрику, должен был отвергаться системой сборки.

Как нетрудно догадаться, ничего хорошего из этого не вышло. Разработчики стали искать пути обойти эту систему. Многие пришли к интересному наблюдению: если обернуть все тесты в блоки `try/catch` и не включать в них проверки (`assertions`), то такие тесты всегда проходят успешно. Люди стали добавлять такие тесты ради достижения 100 % покрытия.

Не стоит и говорить, что эти тесты не приносили никакой пользы проекту. Более того, они вредили, так как отнимали время у более продуктивной деятельности и повышали затраты на сопровождение проекта.

Со временем требования были снижены до 90 %, затем до 80 %, а через какое-то время были полностью сняты (и к лучшему!).

Встроенный тип `integer` содержит много ветвлений, скрытых от инструментов, проверяющих покрытие. Эти ветвления могут приводить к разным результатам в зависимости от типа входящего значения. Ниже перечислены лишь некоторые из этих значений, и ни одно из них не сможет быть преобразовано в целое число:

- Null;
- пустая строка;
- строка, не представляющая целое число;
- слишком большая строка.

Вы можете столкнуться с многочисленными пограничными случаями, и нет никакой возможности проверить, что все они учитываются вашими тестами.

Это не означает, что метрики покрытия должны учитывать пути выполнения во внешних библиотеках, а указывает лишь на то, что на эти метрики нельзя полагаться для определения того, хороши или плохи ваши юнит-тесты. Метрики покрытия не могут сказать, насколько исчерпывающими являются ваши тесты; не могут они сказать и то, достаточно ли вы создали тестов.

1.3.4. Процент покрытия как цель

Надеюсь, я убедил вас, что метрики покрытия не могут стать основой для определения качества тестов в вашем проекте. Также не стоит ставить себе цель достичь какого-то конкретного процента покрытия, будь то 100 %, 90 % или даже «скромные» 70 %. Лучше всего рассматривать метрику покрытия как индикатор, а не как самоцель.

Представьте себе пациента в больнице. Высокая температура может быть симптомом лихорадки и является полезным наблюдением. Однако больница не должна ставить себе цель снижение температуры пациента любой ценой. В противном случае она может прийти к быстрому и «эффективному» решению, установив кондиционер рядом с пациентом и сбивая температуру за счет потока холодного воздуха, направленного на пациента. Конечно, никакого смысла в таком «решении» нет.

СОВЕТ

Полезно иметь высокое покрытие в наиболее важных частях системы. Плохо превращать такое высокое покрытие в требование.

Аналогичным образом стремление к конкретному проценту покрытия создает неверный стимул, противоречащий цели юнит-тестирования. Вместо того чтобы сосредоточиться на тестировании действительно важных вещей, люди начнут искать способы для достижения этой искусственной цели. Юнит-тестирование и без того достаточно сложно. Установление произвольной цели в виде конкретного процента покрытия только отвлекает разработчиков и мешает им обдумывать то, что и зачем они тестируют.

Еще раз: метрики покрытия — хороший негативный признак, но плохой позитивный. Низкие показатели покрытия — допустим, ниже 60 % — являются верным признаком проблем с тестами. Они означают, что в проекте присутствует большой объем непротестированного кода. Однако высокое покрытие не означает, что проблем с тестами нет. Таким образом, метрики покрытия — только первый шаг на пути к определению качества тестов.

1.4. Какими должны быть успешные тесты?

Большая часть этой главы прошла за обсуждением неправильных способов оценки качества тестов: использования метрик покрытия. Какой способ будет правильным? Как оценить качество ваших тестов? Единственный надежный способ — оценка каждого теста по отдельности. Конечно, вам не нужно оценивать их все сразу; это может быть слишком серьезным мероприятием, требующим значительных вложений. Оценку можно проводить постепенно. Суть в том, что автоматизированного способа проверки качества тестов не существует. Вам придется руководствоваться субъективной оценкой.

Давайте рассмотрим более широкую картину того, что обеспечивает успешность тестов в целом. (Вопрос о том, как отличить хорошие тесты от плохих, будет подробно рассмотрен в главе 4.)

Успешный набор тестов обладает следующими свойствами:

- он интегрирован в цикл разработки;
- он проверяет только самые важные части вашего кода;
- он дает максимальную защиту от багов с минимальными затратами на сопровождение.

1.4.1. Интеграция в цикл разработки

Создавать автоматизированные тесты есть смысл только в одном случае: если они постоянно используются. Все тесты должны быть интегрированы в цикл разработки. В идеале они должны выполняться при каждом изменении кода, даже самом незначительном.

1.4.2. Проверка только самых важных частей кода

Как уже говорилось выше, не все тесты одинаково полезны; по той же причине не все части кода проекта заслуживают одинакового внимания в отношении юнит-тестирования. Эффективность тестов зависит не только от того, как структурированы сами тесты, но и от кода, который они проверяют.

Важно направить ваши усилия по юнит-тестированию на самые критические части системы, уделяя остальным частям лишь поверхностное внимание. В большинстве

случаев самой важной является часть, содержащая бизнес-логику, — *модель предметной области (доменная модель)*¹. Тестирование бизнес-логики обеспечивает тестам наилучшую эффективность.

Все остальные части можно разделить на три категории:

- инфраструктурный код;
- внешние сервисы и зависимости — например, базы данных и сторонние системы;
- код, связывающий все компоненты воедино.

Некоторые из этих компонентов могут потребовать тщательного юнит-тестирования. Например, инфраструктурный код может содержать сложные и важные алгоритмы, для которых также следует определить множество тестов. Но в общем случае большая часть вашего внимания должна быть направлена на модель предметной области.

Некоторые тесты — например, интеграционные — могут выходить за пределы модели предметной области и проверять, как работает система в целом, включая некритические части вашей кодовой базы. И это нормально. Но в первую очередь сосредоточиться нужно именно на модели предметной области.

Чтобы выполнить эту рекомендацию, следует отделить модель предметной области от несущественных частей кодовой базы. Эта тема более подробно рассматривается в части II.

1.4.3. Максимальная защита от багов при минимальных затратах на сопровождение

Самая трудная часть юнит-тестирования — достижение максимальной защиты от багов при минимуме затрат на сопровождение. Это главная тема книги.

Недостаточно внедрить тесты в систему сборки, и недостаточно поддерживать высокое тестовое покрытие модели предметной области. Также важно включать в проект только тесты, обладающие наибольшей эффективностью. В первом приближении эффективность тестов можно рассматривать как разницу между защитой от багов и затратами на сопровождение (эта тема более подробно рассматривается в главе 4).

Саму задачу написания эффективных тестов можно разделить на две подзадачи:

- умение распознать эффективный тест (и по аналогии — тест с низкой эффективностью);
- умение написать эффективный тест.

¹ См. *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Эрик Эванс (Eric Evans), Addison-Wesley, 2003. (На русском языке: Эванс, Эрик. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. : Пер. с англ. — М.: ООО «И.Д. Вильямс», 2011. — 448 с. — Примеч. ред.)

Может показаться, что это одно и то же, однако эти задачи различаются по своей природе. Чтобы распознать эффективный тест, необходимо иметь правильную систему координат. С другой стороны, для написания эффективного теста также необходимо владеть методами проектирования кода. Юнит-тесты и код, для которого они пишутся, тесно связаны друг с другом, и написание эффективных тестов невозможно без вложения значительных усилий в код, который они покрывают.

Это примерно то же самое, что умение отличить хорошую песню от плохой и умение написать хорошую песню. Чтобы стать композитором, необходимо потратить несравненно больше усилий, чем для того чтобы научиться отличать хорошую музыку от плохой. Тот же принцип применим и к юнит-тестированию. Чтобы написать новый тест, требуется больше усилий, чем для анализа существующего теста, прежде всего потому что тесты не пишутся в вакууме: необходимо принимать во внимание существующий код. Таким образом, хотя основное внимание в книге уделяется юнит-тестам, в ней вы также найдете обсуждение архитектуры кода.

1.5. Что вы узнаете из книги

В этой книге описана система координат, которой вы сможете воспользоваться для анализа любого теста в вашем проекте. Эта система координат фундаментальна: освоив ее, вы сможете взглянуть на многие из своих тестов в новом свете и понять, какие из них приносят пользу, а какие следует переработать или вообще удалить.

После подготовки фундамента (глава 4) книга анализирует существующие методы и практики юнит-тестирования (главы 4–6 и часть главы 7). При этом неважно, знакомы вы с этими методами и практиками или нет. Если знакомы, книга поможет вам взглянуть на них по-новому. Скорее всего, вы уже владеете этими практиками на интуитивном уровне. Книга поможет вам осознать, *почему* методы и приемы, которыми вы пользовались все это время, настолько эффективны.

Не стоит недооценивать этот навык. Способность четко донести ваши идеи до коллег бесцenna. Разработчику — даже очень хорошему — редко достаются лавры за принятые решения, если он не может объяснить, почему именно было принято это решение. Книга поможет вам преобразовать свои знания из области подсознательного в нечто такое, что можно обсудить с другими людьми.

Если у вас нет особого опыта применения методов и передовых практик юнит-тестирования, вы узнаете много нового. Кроме системы координат, которая может использоваться для анализа любого теста в проекте, эта книга научит вас:

- проводить рефакторинг тестов вместе с основным кодом приложения;
- применять разные стили юнит-тестирования;

- писать интеграционные тесты для проверки поведения системы в целом;
- выявлять антипаттерны в юнит-тестах и избегать их.

Кроме юнит-тестов в книге рассматривается вся тема автоматизации тестирования, так что вы также узнаете об интеграционных и end-to-end тестах.

Я использую C# и .NET в своих примерах, но для чтения книги не нужно быть профессионалом в C#. Все обсуждаемые концепции не привязаны к конкретному языку и могут применяться в любом другом объектно-ориентированном языке (например, Java или C++).

Итоги

- Код проекта становится хуже по мере роста проекта. Каждый раз, когда вы что-то изменяете в коде, возрастает ее энтропия, или степень беспорядка в ней. Если не принять должных мер (таких как постоянная чистка и рефакторинг), система непрерывно усложняется и дезорганизуется. Тесты помогают справиться с этой тенденцией. Они становятся своего рода «подушкой безопасности» — средством, которое обеспечивает защиту от багов.
- Писать юнит-тесты важно. Не менее важно писать хорошие юнит-тесты. Проекты с плохими тестами и проекты без тестов вообще приходят к одинаковому результату: либо стагнация, либо множество багов с каждым новым релизом.
- Целью юнит-тестирования является обеспечение стабильного роста проекта. Хорошие юнит-тесты помогают предотвратить стагнацию и сохранить темп разработки со временем. С такими тестами вы будете уверены в том, что изменения не приведут к багам. В свою очередь, это упростит рефакторинг кода или добавление новой функциональности.
- Не все тесты одинаково полезны. С каждым тестом связаны плюсы и минусы, которые необходимо тщательно оценивать. Включайте в проект только наиболее эффективные тесты и избавляйтесь от всех остальных. И код приложения, и код тестов — обязательство, а не актив (*liabilities, not assets*).
- Возможность покрытия кода юнит-тестами — хороший критерий оценки качества этого кода, но он работает только в одном направлении. Это хороший *негативный* признак (если юнит-тестирование кода невозможно, значит, это код плохого качества), но плохой позитивный признак (возможность юнит-тестирования кода не гарантирует качество этого кода).
- Аналогичным образом метрики покрытия служат хорошим негативным, но плохим позитивным признаком. Низкий процент покрытия — хороший признак проблем с тестами, но высокий процент покрытия еще не означает высокого качества тестов.

- Branch coverage предоставляет более качественную информацию о полноте тестов, чем code coverage, но по нему все равно нельзя судить о том, достаточно хороши ваши тесты или нет. Ни одна из метрик покрытия не учитывает наличия проверок (assertions) и ветвей выполнения в сторонних библиотеках, используемых в вашем проекте.
- Установление конкретного процента покрытия как цели создает неправильный стимул. Обеспечивать высокий процент покрытия для основных частей вашей системы хорошо, но не следует превращать этот высокий процент в требование.
- Успешные тесты обладают следующими свойствами:
 - интегрирован в цикл разработки;
 - проверяет только самые важные части вашего кода;
 - дает максимальную защиту от багов с минимальными затратами на сопровождение.
- Чтобы добиться цели юнит-тестирования (то есть обеспечить стабильный рост проекта), необходимо:
 - научиться отличать хорошие тесты от плохих;
 - научиться рефакторить тесты для повышения их качества.

Что такое юнит-тест?

В этой главе:

- ✓ Что такое юнит-тест.
- ✓ Различия между совместными (*shared*), приватными (*private*) и нестабильными (*volatile*) зависимостями.
- ✓ Две школы юнит-тестирования: классическая и лондонская.
- ✓ Различия между юнит-, интеграционными и сквозными (*end-to-end*) тестами.

Как упоминалось в главе 1, в определении юнит-теста кроется на удивление много нюансов. Эти нюансы важнее, чем можно подумать, — до такой степени, что различия в их интерпретации привели к появлению двух разных подходов к юнит-тестированию.

Эти подходы известны под названиями *классической* и *лондонской* школ юнит-тестирования. Классическая школа называется «классической», потому что изначально все именно так подходили к юнит-тестированию. Лондонская школа происходит из сообщества программистов в Лондоне. Обсуждение различий между классической и лондонской школой закладывает фундамент для главы 5, в которой рассматривается тема моков (*mocks*) и хрупкости тестов.

2.1. Определение юнит-теста

Существует много определений юнит-теста. Все эти определения сводятся к трем важным атрибутам, перечисленным ниже. Юнит-тестом называется автоматизированный тест, который:

- проверяет правильность работы небольшого фрагмента кода (также называемого юнитом);
- делает это быстро
- и поддерживая изоляцию от другого кода.

По поводу первых двух атрибутов особых споров нет. Существуют разногласия относительно того, что именно можно считать быстрым юнит-тестом, так как это довольно субъективная метрика. Но в целом это не так важно — если вас устраивает скорость работы ваших тестов, это означает, что они достаточно быстры.

КЛАССИЧЕСКАЯ И ЛОНДОНСКАЯ ШКОЛЫ ЮНИТ-ТЕСТИРОВАНИЯ

Классический подход также иногда называется «дetroitским». Пожалуй, наиболее канонической книгой по классической школе следует считать книгу Кента Бека (Kent Beck) «Test-Driven Development: By Example» (Addison-Wesley Professional, 2002).

Самыми заметными сторонниками лондонского стиля являются Стив Фримен (Steve Freeman) и Нат Прайс (Nat Pryce). Я рекомендую их книгу «Growing Object-Oriented Software, Guided by Tests» (Addison-Wesley Professional, 2009) как хороший источник информации по данной теме.

Большие расхождения во мнениях проявляются по поводу третьего атрибута. Вопрос изоляции — это корень различий между классической и лондонской школой юнит-тестирования. Как будет показано в следующем разделе, все остальные различия двух школ происходят из несогласия относительно того, что же именно означает «изоляция».

2.1.1. Вопрос изоляции: лондонская школа

Что же означает «изоляция кода» в юнит-тестировании? Лондонская школа описывает это как изоляцию тестируемого кода от его зависимостей. Это означает, что если класс имеет зависимость от другого класса или нескольких классов, все такие зависимости должны быть заменены на тестовые заглушки (test doubles). Это позволит вам сосредоточиться исключительно на тестируемом классе, изолировав его поведение от внешнего влияния.

ОПРЕДЕЛЕНИЕ

Тестовая заглушка (test double) — объект, который выглядит и ведет себя как его рабочий аналог, но в действительности представляет собой упрощенную версию, более удобную для тестирования. Термин ввел Джерард Месарос (Gerard Meszaros) в своей книге «xUnit Test Patterns: Refactoring Test Code» (Addison-Wesley, 2007).

На рис. 2.1 показано, как обычно достигается изоляция. Юнит-тест, который в противном случае проверял бы тестируемую систему со всеми зависимостями, теперь может делать это отдельно от этих зависимостей.

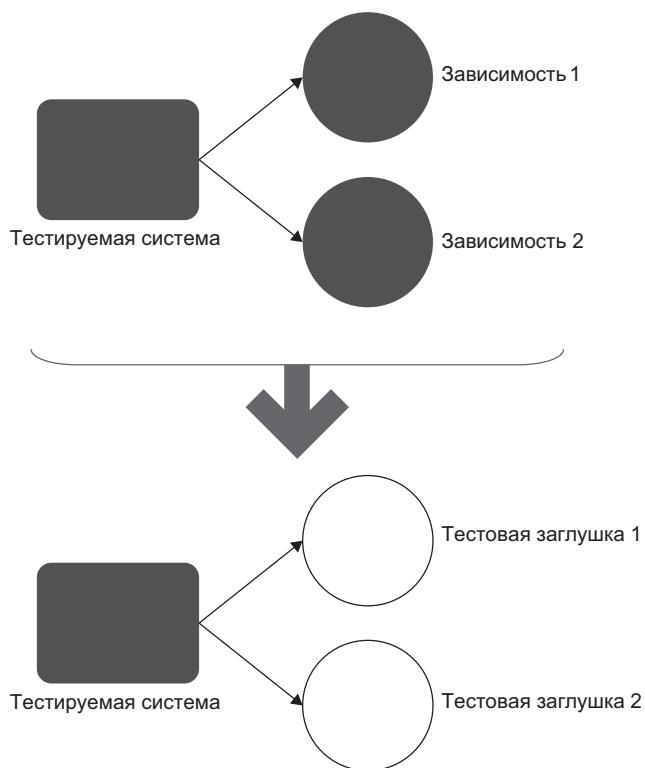


Рис. 2.1. Замена зависимостей в тестируемой системе на заглушки позволяет сосредоточиться исключительно на тестируемой системе

Одно из преимуществ такого подхода заключается в том, что в случае падения теста вы точно знаете, какая часть кода содержит ошибку: это сама тестируемая система. Других подозреваемых быть не может, потому что все соседние классы были заменены заглушками.

Другим преимуществом является возможность разбиения *графа объектов* — сети взаимодействующих классов, решают одни задачу. Такая сеть может быть достаточно сложной: каждый класс может иметь несколько зависимостей, у каждой из которых могут быть собственные зависимости, и т. д. Классы даже могут создавать циклические зависимости, в которых цепочка зависимостей в конечном итоге замыкается на начальный класс.

Без тестовых заглушек протестировать код с множеством зависимостей достаточно сложно. Единственный вариант здесь — это воссоздание полного графа объектов в тесте, что может оказаться неподъемной задачей, если количество задействованных классов слишком велико.

Тестовые заглушки позволяют выйти из положения. Вы можете заменить непосредственные зависимости класса. Как следствие, вам не придется заниматься зависимостями этих зависимостей и остальными классами в графе. Разбивая таким образом граф объектов, вы можете значительно сократить количество кода в юнит-тесте.

И не стоит забывать еще одно небольшое, но полезное преимущество такого подхода к изоляции юнит-тестов: он позволяет установить простые правила тестирования рабочего кода. А именно, юнит-тестируовать только один класс за раз. На рис. 2.2 показано, как обычно выглядит такой подход.

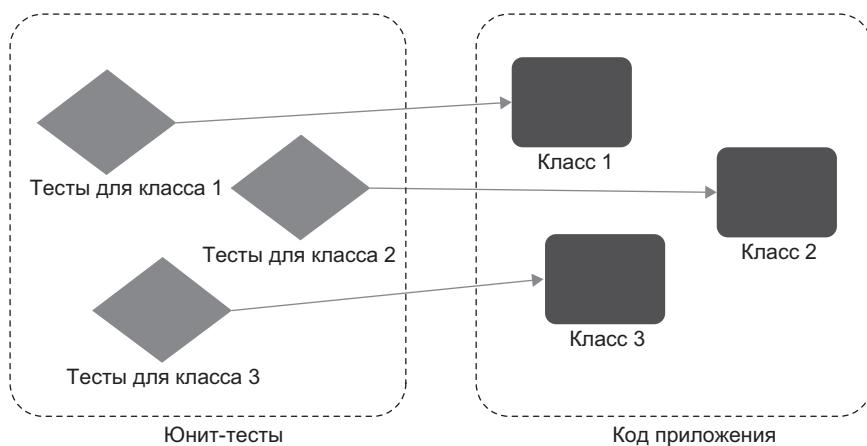


Рис. 2.2. Изоляция тестируемого класса от его зависимостей помогает установить простую структуру тестов: одному классу в тестах соответствует один класс в коде приложения

Рассмотрим несколько примеров. Так как классический стиль лучше знаком большинству людей, я сначала приведу примеры тестов, написанных в этом стиле, а потом перепишу их с использованием подхода лондонской школы.

Допустим, у вас есть интернет-магазин с одним сценарием использования: покупка товара. Если количества товара на складе достаточно, покупка считается успешной, а запас товара сокращается на величину заказа. Если товара недостаточно, то покупка отклоняется, и состояние склада не меняется.

В листинге 2.1 приведены два теста, которые проверяют, что покупка завершается успешно только при достаточном количестве товара на складе. Тесты написаны в классическом стиле и используют стандартную последовательность из трех фаз: подготовка, действие и проверка (Arrange/Act/Assert — сокращенно AAA; эта последовательность более подробно рассматривается в главе 3).

Как видно из листинга, в фазе подготовки (arrange) тесты подготавливают тестируемую систему и ее зависимости. Вызов `customer.Purchase()` относится к фазе

действия (act), в которой выполняется проверяемое поведение. Команды `Assert` в фазе проверки (assert) проверяют, привело ли действие к ожидаемому результату.

Листинг 2.1. Тесты, написанные в классическом стиле юнит-тестирования

```
[Fact]
public void Purchase_succeeds_when_enough_inventory()
{
    // Arrange
    var store = new Store();
    store.AddInventory(Product.Shampoo, 10);
    var customer = new Customer();

    // Act
    bool success = customer.Purchase(store, Product.Shampoo, 5);

    // Assert
    Assert.True(success);
    Assert.Equal(5, store.GetInventory(Product.Shampoo)); ← Уменьшает количество товара на складе на 5
}

[Fact]
public void Purchase_fails_when_not_enough_inventory()
{
    // Arrange
    var store = new Store();
    store.AddInventory(Product.Shampoo, 10);
    var customer = new Customer();

    // Act
    bool success = customer.Purchase(store, Product.Shampoo, 15);

    // Assert
    Assert.False(success);
    Assert.Equal(10, store.GetInventory(Product.Shampoo)); ← Количество товара на складе остается неизменным
}

public enum Product
{
    Shampoo,
    Book
}
```

В фазе подготовки тесты собирают воедино объекты двух видов: саму тестируемую систему (SUT, system under test) и одного коллаборатора (collaborator). В данном случае `Customer` — это SUT, а `Store` — коллаборатор. Коллаборатор необходим по двум причинам:

- чтобы не было ошибок компиляции, методу `customer.Purchase()` необходим параметр типа `Store`;

- для фазы проверки, так как одним из результатов `customer.Purchase()` является потенциальное уменьшение количества товара на складе.

`Product.Shampoo`, а также числа `5` и `15` — константы.

Этот код является примером классического стиля юнит-тестирования: тест не заменяет коллаборатор (класс `Store`) на мок, а использует его рабочую версию. Одно из следствий этого стиля заключается в том, что тест теперь фактически проверяет оба класса (`Customer` и `Store`), а не только SUT (`Customer`). Такие тесты упадут при любой ошибке в `Store`, затрагивающей `Customer`, даже если сам `Customer` при этом работает правильно. Тесты не изолируют эти два класса друг от друга.

ОПРЕДЕЛЕНИЕ

Тестируемый метод (MUT, method under test) — метод SUT, вызываемый тестом. Термины MUT и SUT часто используются как синонимы, но под MUT подразумевают метод, тогда как под SUT — весь класс.

Теперь изменим пример, переписав его в лондонском стиле. Для этого нужно заменить экземпляры коллаборатора `Store` на тестовые заглушки, а именно моки (mocks).

Для моков я использую фреймворк Moq (<https://github.com/moq/moq4>), но в C# существуют и альтернативные фреймворки, например NSubstitute (<https://github.com/nsubstitute/NSubstitute>). Аналогичные фреймворки существуют во всех объектно-ориентированных языках. Например, в Java можно использовать Mockito, JMock или EasyMock.

ОПРЕДЕЛЕНИЕ

Мок (mock) — особая разновидность тестовой заглушки, которая позволяет проанализировать взаимодействия между тестируемой системой и ее коллабораторами.

Мы вернемся к теме моков (mocks), стабов (stubs) и различий между ними в других главах. Пока просто помните, что моки являются подмножеством тестовых заглушек (test doubles). Термины «тестовая заглушка (test double)» и «мок (mock)» часто используются как синонимы, но на самом деле это не так (подробнее об этом в главе 5):

- *тестовая заглушка (test double)* — общий термин, описывающий любые разновидности фиктивных зависимостей, используемых в тестах;
- *мок (mock)* — всего лишь одна из разновидностей таких зависимостей.

В листинге 2.2 показано, как выглядят тесты после изоляции `Customer` от коллаборатора `Store`.

Обратите внимание, насколько эти тесты отличаются от написанных в классическом стиле. В фазе подготовки (`arrange`) тесты уже не создают полнофункциональный

экземпляр `Store`; вместо этого для него создается замена при помощи класса `Mock<T>` из библиотеки Moq.

Листинг 2.2. Тесты, написанные в лондонском стиле

```
[Fact]
public void Purchase_succeeds_when_enough_inventory()
{
    // Arrange
    var storeMock = new Mock<IStore>();
    storeMock
        .Setup(x => x.HasEnoughInventory(Product.Shampoo, 5))
        .Returns(true);
    var customer = new Customer();

    // Act
    bool success = customer.Purchase(
        storeMock.Object, Product.Shampoo, 5);

    // Assert
    Assert.True(success);
    storeMock.Verify(
        x => x.RemoveInventory(Product.Shampoo, 5),
        Times.Once);
}

[Fact]
public void Purchase_fails_when_not_enough_inventory()
{
    // Arrange
    var storeMock = new Mock<IStore>();
    storeMock
        .Setup(x => x.HasEnoughInventory(Product.Shampoo, 5))
        .Returns(false);
    var customer = new Customer();

    // Act
    bool success = customer.Purchase(
        storeMock.Object, Product.Shampoo, 5);

    // Assert
    Assert.False(success);
    storeMock.Verify(
        x => x.RemoveInventory(Product.Shampoo, 5),
        Times.Never);
}
```

Также вместо того, чтобы изменять состояние `Store` добавлением в него товара для дальнейшей покупки, мы напрямую сообщаем моку, как следует реагировать на вызовы `HasEnoughInventory()`. Мок реагирует на этот запрос так, как требуется тесту, независимо от фактического состояния `Store`. Более того, тесты вообще не используют `Store` — мы добавили интерфейс `IStore` и используем этот интерфейс вместо класса `Store`.

Интерфейсы необходимы для изоляции тестируемой системы от колабораторов. (Также можно заменять на мок сам класс, но это является антипаттерном; данная тема рассматривается в главе 11.) В главе 8 работа с интерфейсами описывается более подробно.

Фаза проверки тоже изменилась, и именно здесь кроется ключевое различие. Мы проверяем результат работы метода `customer.Purchase`, как и прежде, но взаимодействие между `Customer` и `Store` теперь проверяется по-другому. Ранее для этого использовалось состояние магазина. Теперь же мы анализируем взаимодействия между `Customer` и `Store`: тесты проверяют, какой метод и с какими параметрами `Customer` вызвал у `Store`. Для этого в мок передается вызываемый метод (`x.RemoveInventory`), а также сколько раз этот метод должен был вызываться в течение теста. Если покупка завершается успехом, метод должен вызываться один раз (`Times.Once`). В случае же неудачи метод не должен вызываться вообще (`Times.Never`).

2.1.2. Вопрос изоляции: классический подход

Итак, лондонский стиль рассматривает вопрос изоляции как отделение тестируемого кода от его колабораторов с помощью моков. Интересно, что этот подход также влияет на представление о том, что именно должен собой представлять тестируемый фрагмент кода (юнит). Ниже снова перечисляются атрибуты юнит-теста:

- он проверяет правильность работы небольшого фрагмента кода (также называемого юнитом);
- делает это быстро
- поддерживая изоляцию от другого кода.

Кроме третьего атрибута, оставляющего место для разных интерпретаций, первый атрибут также интерпретируется неоднозначно. Насколько небольшим должен быть небольшой фрагмент кода (юнит)? Как было показано в предыдущем разделе, если принять лондонский подход и изолировать каждый класс, то естественным следствием этого подхода будет то, что юнит должен быть одним классом или методом внутри класса. Юнит не может быть больше по определению. Время от времени вы можете тестировать пару классов одновременно, но в большинстве случаев тестируется будет только один класс.

Как упоминалось ранее, изоляцию кода также можно интерпретировать иначе — классическим способом. В классическом подходе изолируются друг от друга не фрагменты рабочего кода, а сами тесты. Такая изоляция позволяет вам запускать эти тесты параллельно, последовательно и в любом порядке, не влияя на результат работы этих тестов.

Классический подход к изоляции не запрещает вам тестировать несколько классов одновременно, при условии что все они находятся в памяти и не обращаются

к совместному состоянию (*shared state*), через которое тесты могут влиять на результат выполнения друг друга. Типичными примерами такого совместного состояния служат внепроцессные (*out-of-process*) зависимости — база данных, файловая система и т. д.

Например, один тест может создать запись в базе данных, а другой удалит эту запись до того, как первый тест будет полностью выполнен. Если эти два теста запустить параллельно, первый завершится неудачей — и не потому, что код приложения работает неправильно, а из-за влияния со стороны второго теста.

СОВМЕСТНЫЕ, ПРИВАТНЫЕ И ВНЕПРОЦЕССНЫЕ ЗАВИСИМОСТИ

Совместной (*shared*) зависимостью называется зависимость, к которой имеют доступ более одного теста и которая предоставляет им возможность влиять на результаты друг друга. Типичный пример совместной зависимости — статическое изменяемое поле. Изменение в таком поле отражается на всех юнит-тестах, выполняемых в одном процессе. База данных — другой типичный пример совместной зависимости.

Приватной (*private*) зависимостью называется зависимость, которая не является совместной.

Внепроцессной (*out-of-process*) зависимостью называется зависимость, работающая вне процесса приложения; это посредник (*proxy*) к данным, которых еще нет в памяти. В подавляющем большинстве случаев внепроцессная (*out-of-process*) зависимость соответствует совместной (*shared*) зависимости, но не всегда. Например, база данных — это одновременно и внепроцессная, и совместная зависимость. Но если эта база данных будет запускаться в контейнере Docker перед каждым тестом, она перейдет в категорию внепроцессной, но не совместной, так как у каждого теста будет свой экземпляр этой базы данных. Аналогичным образом база данных, доступная только для чтения, также является внепроцессной, но не совместной, несмотря на то что она используется несколькими тестами. Тесты не могут изменить состояние такой базы данных, а следовательно, не могут влиять на результаты друг друга.

Такой подход к вопросу изоляции приводит к намного меньшему использованию моков и других тестовых заглушек по сравнению с лондонским подходом: как правило, только для совместных (*shared*) зависимостей, чтобы избежать влияния тестов друг на друга. На рис. 2.3 показано, как это выглядит.

Обратите внимание на то, что совместные (*shared*) зависимости — это зависимости, которые являются общими между юнит-тестами, а не между тестируемыми классами (юнитами). В этом смысле зависимость-одиночка (*singleton*) не является совместной, если в каждом тесте создается новый ее экземпляр. Хотя в основном коде приложения существует только один экземпляр одиночки, тесты могут не следовать этому паттерну и не переиспользовать этот экземпляр. Такая зависимость будет приватной.

Например, обычно существует только один экземпляр класса конфигурации, который используется всем кодом приложения. Но если вы реализуете внедрение зависимостей (dependency injection) в тестируемом классе через конструктор, то в каждом тесте сможете создавать новый экземпляр класса конфигурации; вам не придется переиспользовать единственный экземпляр в тестах. С другой стороны, создать новую файловую систему или базу данных не получится — они должны либо переиспользоваться тестами, либо заменяться тестовыми заглушками.

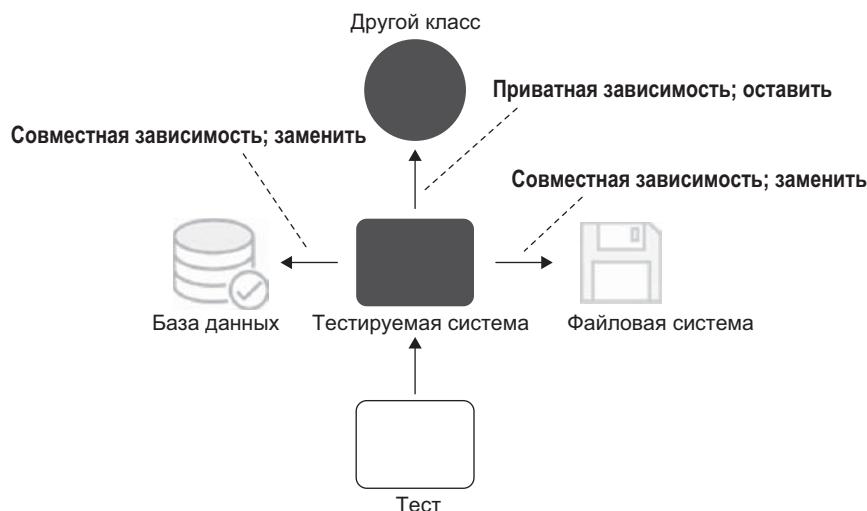


Рис. 2.3. Изоляция юнит-тестов подразумевает изоляцию тестируемого класса только от совместных (shared) зависимостей. Приватные зависимости могут использоваться в тестах как есть

Другая причина для замены совместных зависимостей моками — ускорение тестов. В отличие от приватных зависимостей, совместные зависимости почти всегда являются внепроцессными. По этой причине обращения к совместным зависимостям (например, базам данных или файловой системе) занимают больше времени, чем обращения к приватным зависимостям. А поскольку необходимость быстрого выполнения является вторым атрибутом в определении юнит-теста, такие обращения помещают тесты из категории юнит-тестирования в область интеграционного тестирования. Интеграционное тестирование будет более подробно рассмотрено позднее в этой главе.

Классический подход к вопросу изоляции также приводит к различию во взглядах на то, что должен из себя представлять юнит (тестируемый фрагмент кода). Юнит не обязан ограничиваться классом. Вы можете юнит-тестировать как один класс, так и несколько классов, при условии что ни один из них не является совместной зависимостью.

СОВМЕСТНЫЕ (SHARED) И НЕСТАБИЛЬНЫЕ (VOLATILE) ЗАВИСИМОСТИ

Существует и другой термин с похожим, хотя и не идентичным смыслом: нестабильная (volatile) зависимость. По теме управления зависимостями я рекомендую книгу «Dependency Injection: Principles, Practices, Patterns» Стивена ван Дьюрзена (Steven van Deursen) и Марка Симана (Mark Seemann) (Manning Publications, 2018).

Нестабильной называется зависимость, проявляющая одно из следующих свойств:

- ✓ Необходимость подготовки и настройки среды в дополнение к той, что устанавливается на машине разработчика по умолчанию. Хорошими примерами служат базы данных и сервисы различных API. Они требуют дополнительной настройки и не устанавливаются на машинах вашей организации по умолчанию.
- ✓ Наличие недетерминированного поведения. Пример — генератор случайных чисел или класс, возвращающий текущую дату и время. Такие зависимости являются недетерминированными, потому что они выдают разные результаты при каждом обращении.

Как видите, концепции совместных (shared) и нестабильных (volatile) зависимостей похожи. Например, база данных является одновременно и совместной, и нестабильной. Тем не менее с файловой системой это не так. Файловая система не является нестабильной, потому что она устанавливается на машине каждого разработчика и ведет себя детерминированно в подавляющем большинстве случаев. Однако файловая система является совместной зависимостью, так как через нее юнит-тесты могут влиять на результаты друг друга. Аналогичным образом генератор случайных чисел нестабилен, но поскольку каждому тесту можно предоставить его отдельный экземпляр, такая зависимость не является совместной.

2.2. Классическая и лондонская школы юнит-тестирования

Как видите, суть различий между лондонской и классической школой кроется в подходе к вопросу изоляции. Лондонская школа рассматривает его как изоляцию тестируемой системы от ее коллaborаторов, тогда как классическая школа рассматривает его как изоляцию самих юнит-тестов друг от друга.

Такое несущественное на первый взгляд различие привело к значительным расхождениям в подходе к юнит-тестированию, что, в свою очередь, привело к появлению двух школ юнит-тестирования: классической и лондонской. Эти расхождения можно разбить на три основных темы:

- вопрос изоляции;
- что собой представляет тестируемый фрагмент кода (юнит);
- работа с зависимостями.

В таблице 2.1 приведена краткая сводка различий.

Таблица 2.1. Различия между лондонской и классической школами юнит-тестирования, разбитые по трем темам: подход к изоляции, размер юнита и использование тестовых заглушек (моков)

	Изоляция	Юнит — это	Использование моков для
Лондонская школа	Юнитов	Класс	Коллaborаторов (любых изменяемых зависимостей)
Классическая школа	Юнит-тестов	Класс или набор классов	Совместных (shared) зависимостей

2.2.1. Работа с зависимостями в классической и лондонской школах

Несмотря на повсеместное использование моков, лондонская школа все же позволяет использовать в тестах некоторые зависимости без их замены на заглушки. Основное различие здесь в том, является ли зависимость изменяемой: лондонская школа допускает не использовать моки для неизменяемых объектов.

Как было показано в более ранних примерах, при рефакторинге тестов в лондонский стиль я не заменил экземпляры `Product` моками, а использовал реальные объекты (следующий код скопирован из листинга 2.2):

```
[Fact]
public void Purchase_fails_when_not_enough_inventory()
{
    // Arrange
    var storeMock = new Mock<IStore>();
    storeMock
        .Setup(x => x.HasEnoughInventory(Product.Shampoo, 5))
        .Returns(false);
    var customer = new Customer();

    // Act
    bool success = customer.Purchase(storeMock.Object, Product.Shampoo, 5);

    // Assert
    Assert.False(success);
    storeMock.Verify(
        x => x.RemoveInventory(Product.Shampoo, 5),
        Times.Never);
}
```

Из двух зависимостей класса `Customer` только `Store` содержит внутреннее состояние, которое может изменяться со временем. Экземпляры `Product` неизменяемы (сам тип `Product` — перечисление (enum) C#). Поэтому я заменил на мок только экземпляр `Store`. По этой же причине я не заменил на мок число 5 — оно тоже неизменяемо.

Такие неизменяемые объекты называются *объектами-значениями* (*value object*), или просто *значениями* (*value*). Их главная особенность заключается в том, что у них нет «личности»; они идентифицируются исключительно по своему содержимому. Если два объекта имеют одинаковое содержимое, неважно, с каким из них вы работаете: эти экземпляры взаимозаменяемы. Например, если у вас есть два целых числа 5, вы можете использовать любое из них. То же относится к товарам в нашем примере: можно использовать один экземпляр `Product.Shampoo` или объявить несколько экземпляров — это ни на что не повлияет. Такие экземпляры будут иметь одинаковое содержимое, а следовательно, могут использоваться как взаимозаменяемые.

Следует помнить, что концепция объекта-значения не привязана к какому-то конкретному языку программирования или фреймворку. Вы можете написать объект-значение (*value object*) на любом языке. Об объектах-значениях можно более подробно прочитать в моей статье «Entity vs. Value Object: The ultimate list of differences» по адресу <http://mng.bz/KE9O>.

На рис. 2.4 показана классификация зависимостей и как обе школы юнит-тестирования работают с ними. Зависимость может быть либо *совместной* (*shared*), либо *приватной* (*private*). Приватная зависимость, в свою очередь, может быть изменяемой или неизменяемой. В последнем случае она называется *объектом-значением*. Например, база данных является совместной зависимостью — ее внутреннее состояние

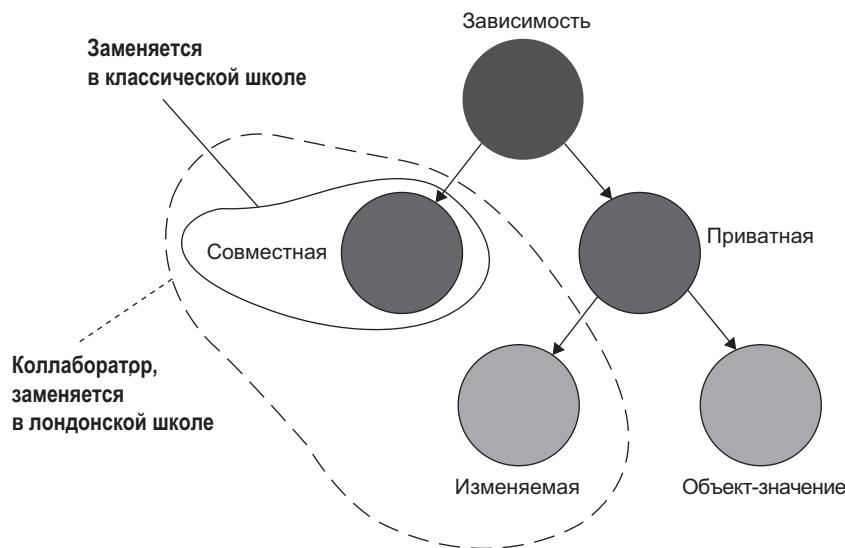


Рис. 2.4. Иерархия зависимостей. Классическая школа выступает за замену совместных зависимостей тестовыми заглушками. Лондонская школа выступает также за замену приватных зависимостей при условии, что они являются изменяемыми

совместно используется всеми тестами (которые не заменяют эту базу данных на мок). Экземпляр `Store` является изменяемой приватной зависимостью. С другой стороны, экземпляр `Product` (а также экземпляр числа 5) является примером неизменяемой приватной зависимости — то есть объекта-значения. Все совместные зависимости изменяемы, но чтобы изменяемая зависимость была совместной, она должна переиспользоваться тестами.

Повторю таблицу 2.1 со сводкой различий между школами.

	Изоляция	Юнит — это	Использование моков для
Лондонская школа	Юнитов	Класс	Коллабораторов (любых изменяемых зависимостей)
Классическая школа	Юнит-тестов	Класс или набор классов	Совместных (shared) зависимостей

Хочу еще раз подчеркнуть одно обстоятельство, касающееся типов зависимостей. Не все внепроцессные (out-of-process) зависимости относятся к категории совместных (shared) зависимостей. Совместная зависимость почти всегда работает вне процесса приложения, но обратное неверно (рис. 2.5). Чтобы внепроцессная зависимость была совместной, она должна предоставлять средства, при помощи которых юнит-тесты могут взаимодействовать друг с другом. Такое взаимодействие осуществляется путем изменения внутреннего состояния зависимости. Неизменяемая

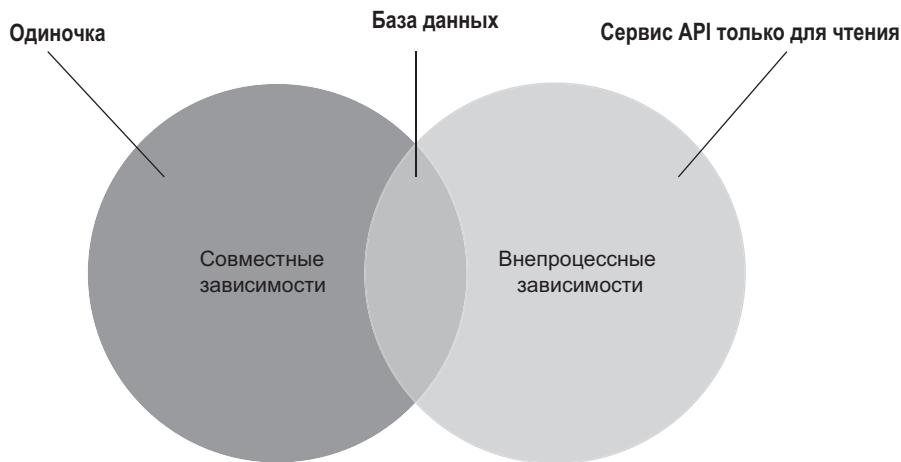


Рис. 2.5. Отношение между совместными и внепроцессными зависимостями. Пример совместной, но не внепроцессной зависимости — одиночка (экземпляр, переиспользуемый всеми тестами) или статическое поле класса. База данных является совместной и внепроцессной — она работает за пределами процесса приложения и является изменяемой. API только для чтения — внепроцессная, но не совместная зависимость, потому что тесты не могут ее изменить, а следовательно, не могут влиять друг на друга

внепроцессная зависимость не предоставляет таких средств. Тесты просто не могут ничего изменить в ней и таким образом не могут повлиять на работу друг друга.

Например, если где-то существует API, возвращающий каталог всех товаров, продаваемых организацией, это не может считаться совместной зависимостью, при условии что этот API не предоставляет функциональности для изменения каталога. Такая зависимость является нестабильной (*volatile*) и внепроцессной, но поскольку тесты не могут изменить возвращаемые ею данные, она не является совместной. Это не означает, что такая зависимость должна использоваться как есть в тестах. В большинстве случаев ее все равно нужно заменять на мок для ускорения теста. Но если внепроцессная зависимость достаточно быстра и соединение с ней стабильно, то использование ее в тестах в исходном виде — тоже приемлемый вариант.

КОЛЛАБОРАТОРЫ И ЗАВИСИМОСТИ

Коллaborатор (*collaborator*) — это изменяемая зависимость. Например, класс, предоставляющий доступ к базе данных, является коллaborатором (при условии что доступ не только для чтения). Store также является коллaborатором, потому что его состояние может изменяться со временем.

Product и число 5 также являются зависимостями, но при этом они не коллабораторы. Это значения (*values*) или объекты-значения (*value objects*).

Типичный класс может работать с зависимостями обоих типов: коллабораторами и значениями. Взгляните на следующий вызов метода:

```
customer.Purchase(store, Product.Shampoo, 5)
```

Здесь присутствуют три зависимости. Одна из них (*store*) является коллаборатором, а две другие (*Product.Shampoo*, 5) — нет.

В этой книге я использую термины «совместная зависимость» и «внепроцессная зависимость» как синонимы, если только в тексте явно не указано обратное. В реальных проектах редко встречаются совместные зависимости, которые не являются также внепроцессными. Если зависимость является внутрипроцессной, вы можете легко создать отдельный ее экземпляр для каждого теста; нет необходимости в ее переработке. Аналогичным образом внепроцессные зависимости, как правило, являются также и совместными, так как большинство внепроцессных зависимостей могут изменяться тестами.

Разобравшись с определениями и терминами, давайте теперь сравним две школы.

2.3. Сравнение классической и лондонской школ юнит-тестирования

Еще раз: главное различие между классической и лондонской школами в том, как они интерпретируют аспект изоляции в определении юнит-теста. В свою очередь,

это различие приводит к разнице в работе с зависимостями и в интерпретации того, что из себя должен представлять юнит (тестируемый фрагмент кода).

Я предпочитаю классическую школу юнит-тестирования. Она обычно приводит к тестам более высокого качества, а следовательно, лучше подходит для достижения цели юнит-тестирования — стабильного роста вашего проекта. Причина кроется в хрупкости: тесты, использующие моки, обычно бывают более хрупкими, чем классические тесты. Более подробно эту тему мы обсудим в главе 5, а пока возьмем основные привлекательные стороны лондонской школы и оценим их одну за одной.

Лондонская школа обладает следующими преимуществами:

- *Улучшенная детализация.* Тесты высокодетализированы и проверяют только один класс за раз.
- *Упрощение юнит-тестирования большого графа взаимосвязанных классов.* Так как все коллaborаторы заменяются тестовыми заглушками, вам не придется беспокоиться о них при написании теста.
- *Если тест падает, вы точно знаете, в какой функциональности произошел сбой.* Так как все коллaborаторы заменены на заглушки, не может быть других подозреваемых, кроме самого тестируемого класса. Конечно, все еще возможны ситуации, в которых тестируемая система использует объект-значение, и изменение в этом объекте-значении приводит к падению теста. Однако такие случаи встречаются не так часто, потому что все остальные зависимости устраниены в тестах.

2.3.1. Юнит-тестирование одного класса за раз

Первый пункт (улучшение детализации) связан с обсуждением того, что собой представляет юнит в юнит-тестировании. Лондонская школа считает, что это должен быть класс. Разработчики с опытом объектно-ориентированного программирования обычно рассматривают классы как атомарные элементы, из которых складывается фундамент любой кодовой базы. Это естественным образом приводит к тому, что классы также начинают рассматриваться как атомарные единицы для проверки в тестах. Такая тенденция понятна, но ошибочна.

СОВЕТ

Тесты не должны проверять единицы кода (*units of code*). Вместо этого они должны проверять единицы поведения (*units of behavior*) — нечто имеющее смысл для предметной области, а в идеале — нечто такое, полезность которого будет понятна бизнесу. Количество классов, необходимых для реализации такой единицы поведения, не имеет значения. Тест может охватывать как несколько классов, так и только один класс или даже всего один маленький метод.

Таким образом, повышение детализации тестируемого кода само по себе не является чем-то полезным. Если тест проверяет одну единицу поведения, это хороший тест. Стремление к тому, чтобы охватить что-то меньшее, может повредить вашим юнит-тестам, так как становится сложнее понять, что же именно эти тесты проверяют. В идеале тест должен рассказывать о проблеме, решаемой кодом проекта, и этот рассказ должен быть связным и понятным даже для непрограммиста.

Пример связного рассказа:

Когда я зову свою собаку, она идет ко мне.

Теперь сравните со следующим рассказом:

Когда я зову свою собаку, она сначала выставляет вперед левую переднюю лапу, потом правую переднюю лапу, поворачивает голову, начинает вилять хвостом...

Второй рассказ не кажется особо вразумительным. Для чего нужны все эти движения? Собака идет ко мне? Или убегает? Сходу не скажешь. Так начинают выглядеть ваши тесты, когда вы ориентируетесь на отдельные классы (лапы, голова, хвост) вместо фактического поведения (собака идет к хозяину). В главе 5 мы вернемся к разговору о наблюдаемом поведении и о том, как отличить его от деталей внутренней реализации.

2.3.2. Юнит-тестирование большого графа взаимосвязанных классов

Использование моков вместо реальных колабораторов может упростить тестирование класса — особенно при наличии сложного графа объектов, в котором тестируемый класс имеет зависимости, каждая из которых имеет свои зависимости, и т. д. на несколько уровней в глубину. С тестовыми заглушками вы можете устраниТЬ непосредственные зависимости тестируемого класса и таким образом разделить граф объектов, что может значительно сократить объем подготовки, необходимой для юнит-тестирования. Если же следовать канонам классической школы, то необходимо будет воссоздать полный граф объектов (кроме совместных зависимостей) просто ради того, чтобы подготовить тестируемую систему, что может потребовать значительной работы.

И хотя все это правда, такие рассуждения фокусируются не на той проблеме. Вместо того чтобы искать способы тестирования большого сложного графа взаимосвязанных классов, следует сконцентрироваться на том, чтобы у вас изначально не было такого графа классов. Как правило, большой граф классов является результатом плохого проектирования кода.

На самом деле тот факт, что тесты подчеркивают эту проблему, является преимуществом. Как обсуждалось в главе 1, сама возможность юнит-тестирования

кода служит хорошим негативным признаком — она позволяет определить плохое качество кода с относительно высокой точностью. Если вы видите, что для юнит-тестирования класса необходимо увеличить фазу подготовки теста сверх любых разумных пределов, это указывает на определенные проблемы с нижележащим кодом. Использование моков только скрывает эту проблему, не пытаясь справиться с ее корневой причиной. О том, как решаются проблемы с проектированием кода, будет рассказано в части II.

2.3.3. Выявление точного местонахождения ошибки

Если в систему с тестами в лондонском стиле будет внесена ошибка, то, как правило, упадут только те тесты, у которых тестируемая система содержит ошибку. С другой стороны, при классическом подходе также могут падать и тесты, проверяющие клиентов неправильно функционирующего класса. Это приводит к каскадному эффекту: одна ошибка может вызвать тестовые сбои во всей системе. В результате усложняется поиск корневой проблемы и требуется дополнительное время для отладки.

Это хороший довод в пользу лондонской школы, но, на мой взгляд, не является большой проблемой для классической школы. Если вы регулярно запускаете тесты (в идеале после каждого изменения в коде приложения), то знаете, что стало причиной ошибки — это тот код, который вы редактировали в последний раз, и поэтому найти ошибку будет не так трудно. Также вам не нужно просматривать все непрошедшие тесты. Исправление одного автоматически исправляет все остальные.

Более того, в каскадном распространении сбоев по всем тестам есть некоторые плюсы. Если ошибка ведет к сбою не только одного теста, но сразу многих, это показывает, что только что сломанный код чрезвычайно ценен — от него зависит вся система. Это полезная информация, которую следует учитывать при работе с этим кодом.

2.3.4. Другие различия между классической и лондонской школами

Остаются еще два различия между классической и лондонской школами:

- подход к проектированию системы на базе методологии разработки через тестирование (TDD, Test-Driven Development);
- проблема излишней спецификации (over-specification).

Лондонский стиль юнит-тестирования ведет к методологии TDD по схеме «снаружи внутрь» (outside-in): вы начинаете с тестов более высокого уровня, которые задают ожидания для всей системы. Используя моки, вы указываете, с какими коллегами система должна взаимодействовать для достижения ожидаемого результата. Затем вы проходите по графу классов, пока не реализуете их все. Моки делают этот

процесс разработки возможным, потому что вы можете сосредоточиться на одном классе за раз. Вы можете отсечь всех колабораторов тестируемой системы и таким образом отложить реализацию этих колабораторов.

Классическая школа такой возможности не дает, потому что вам приходится иметь дело с реальными объектами в тестах. Вместо этого обычно используется подход по схеме «изнутри наружу» (*inside-out*). В этом стиле вы начинаете с модели предметной области, а затем накладываете на нее дополнительные слои, пока программный код не станет пригодным для конечного пользователя.

Но, пожалуй, самое принципиальное различие между школами — проблема излишней спецификации (*over-specification*), то есть привязки теста к деталям имплементации тестируемой системы. Лондонский стиль приводит к тестам, завязанным на детали имплементации, чаще, чем классический стиль. И это становится главным аргументом против повсеместного использования моков и лондонского стиля в целом.

Начиная с главы 4 я постепенно расскажу обо всем, что относится к теме применения моков.

РАЗРАБОТКА ЧЕРЕЗ ТЕСТИРОВАНИЕ

Разработка через тестирование (Test-Driven Development, TDD) — процесс разработки ПО, в котором тесты управляют ходом разработки проекта. Процесс состоит из трех (четырех, по мнению некоторых авторов) стадий, которые повторяются для каждого тестового сценария:

1. Написать падающий тест, который покажет, какую функциональность необходимо добавить и каким поведением она должна обладать.
2. Написать код, минимально достаточный для того, чтобы тест проходил. На этой стадии код не обязан быть элегантным или чистым.
3. Провести рефакторинг кода. Вы можете безопасно «чистить» код, защищенный написанными ранее тестами, — сделать его более читаемым и простым в сопровождении.

Хороший источник информации по этой теме — две книги, упоминавшиеся ранее: «Test-Driven Development: By Example» Кента Бека (Kent Beck) и «Growing Object-Oriented Software, Guided by Tests» Стива Фримена (Steve Freeman) и Ната Прайса (Nat Pryce).

2.4. Интеграционные тесты в двух школах

Лондонская и классическая школы также расходятся в определении интеграционного теста. Такое расхождение естественным образом вытекает из различий в их взглядах на вопрос изоляции.

В лондонской школе любой тест, в котором используется реальный объект-коллaborатор, рассматривается как интеграционный. Большинство тестов, написанных

в классическом стиле, будут считаться интеграционными тестами сторонниками лондонской школы. Например, в листинге 2.1 я впервые представил два теста, покрывающих функциональность покупки товара клиентом. Этот код является типичным юнит-тестом с классической точки зрения, но для последователя лондонской школы он будет интеграционным тестом.

В этой книге я использую классические определения как юнит-, так и интеграционного тестирования. Напомню, что юнит-тестом называется автоматизированный тест, который:

- проверяет правильность работы небольшого фрагмента кода (также называемого юнитом);
- делает это быстро
- и поддерживая изоляцию от другого кода.

Теперь, когда я объяснил смысл первого и третьего атрибутов, я перепишу их с точки зрения классической школы. Юнит-тестом называется тест, который:

- проверяет *одну единицу поведения*;
- делает это быстро
- и в изоляции *от других тестов*.

Таким образом, интеграционным тестом называется тест, который не удовлетворяет хотя бы одному из этих критериев. Например, тест, который обращается к совместной зависимости — скажем, базе данных, — не может выполняться в изоляции от других тестов. Изменение состояния базы данных одним тестом приведет к изменению результатов всех остальных тестов, зависящих от той же базы и выполняемых параллельно. Вам придется предпринять дополнительные действия, для того чтобы избежать такого пересечения. В частности, такие тесты должны выполняться последовательно, чтобы каждый тест ожидал своей очереди для работы с совместной зависимостью.

Аналогичным образом обращение к внепроцессной зависимости замедляет тест. Обращение к базе данных добавляет сотни миллисекунд, возможно — до секунды дополнительного времени работы теста. На первый взгляд, миллисекунды особой роли не играют, но когда количество тестов становится достаточно большим, каждая секунда на счету.

Теоретически можно написать медленный тест, работающий только с объектами в памяти, но это не так просто. Взаимодействия между объектами, находящимися в одном процессе, проходят намного быстрее, чем взаимодействия между разными процессами. Даже если тест работает с сотнями объектов в памяти, взаимодействия между ними все равно будут происходить быстрее, чем обращение к базе данных.

Наконец, тест является интеграционным, когда он проверяет две или более единицы поведения. Часто это является результатом попыток оптимизации скорости

тестов. Если у вас имеются два медленных теста, которые выполняют похожие действия, но проверяют разные единицы поведения, может показаться разумным объединить их: один тест, проверяющий две похожие ситуации, работает быстрее, чем два отдельных теста. Но с другой стороны, два исходных теста уже были бы интеграционными тестами (из-за их медленной скорости), так что этот признак обычно не является решающим.

Интеграционный тест также может проверить, как два или более модуля, разработанных разными командами, работают вместе. Такие тесты тоже попадают в третью категорию тестов, проверяющих сразу несколько единиц поведения. Но поскольку подобная интеграция обычно требует внепроцессной зависимости, такой тест будет не удовлетворять сразу трем критериям, а не только одному.

Интеграционное тестирование играет важную роль в обеспечении качества кода приложения за счет проверки всей системы целиком. Интеграционное тестирование более подробно рассматривается в части III.

2.4.1. Сквозные (end-to-end) тесты как подмножество интеграционных тестов

Итак, интеграционным тестом называется тест, который проверяет, что ваш код работает в интеграции с совместными зависимостями, внепроцессными зависимостями или кодом, разработанным другими командами в организации. Также существует отдельная концепция сквозного теста. *Сквозные* (end-to-end) тесты составляют подмножество интеграционных тестов. Они тоже проверяют, как ваш код работает с внепроцессными зависимостями. Сквозные тесты отличаются от интеграционных прежде всего тем, что сквозные тесты обычно включают большее число таких зависимостей.

Граница между этими двумя типами тестов не очень четкая, но в общем случае интеграционный тест работает только с одной или двумя внепроцессными зависимостями. Сквозной же тест обычно работает со всеми внепроцессными зависимостями или с подавляющим их большинством. Отсюда и определение «сквозной» (end-to-end) — оно означает, что тест проверяет систему с точки зрения конечного пользователя, включая все внешние приложения, с которыми интегрирована система (рис. 2.6).

Также встречаются такие термины, как *UI-тесты* (UI — User Interface, то есть пользовательский интерфейс), *GUI-тесты* (GUI — Graphical User Interface, то есть графический пользовательский интерфейс) и *функциональные тесты*. Точной терминологии, к сожалению, нет, но обычно все эти термины считаются синонимами.

Предположим, ваше приложение работает с тремя внепроцессными зависимостями: базой данных, файловой системой и платежным шлюзом. Типичный интеграционный

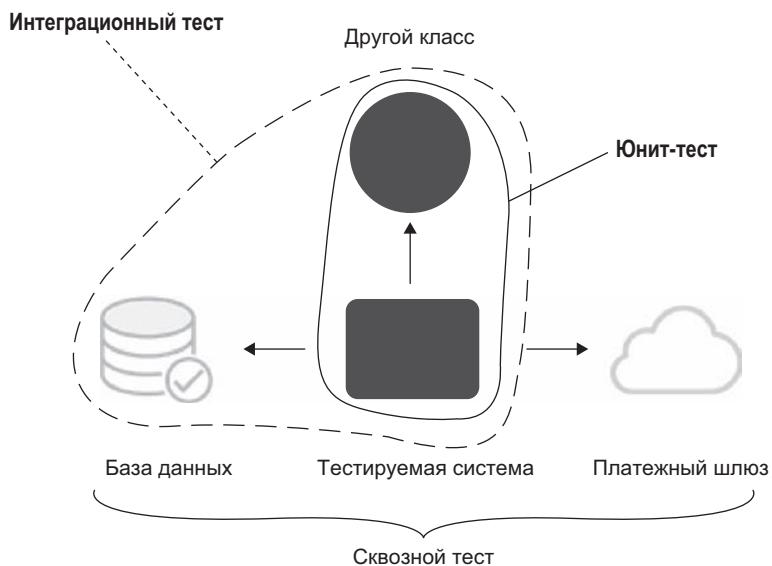


Рис. 2.6. Сквозные тесты обычно включают все или почти все внепроцессные зависимости. Интеграционные тесты проверяют только одну или две такие зависимости — те, которые проще настраиваются автоматически (например, база данных или файловая система)

тест будет включать только базу данных и файловую систему, а платежный шлюз будет заменен тестовой заглушкой. База данных и файловая система находятся под вашим полным контролем, а следовательно, вы можете легко привести их к нужному состоянию в тестах, тогда как над платежным шлюзом у вас такого контроля нет. Возможно, вам придется обратиться к компании, занимающейся обработкой платежей, для создания специальной тестовой учетной записи. Вам также придется время от времени обращаться к этой учетной записи, чтобы вручную удалять платежи, оставшиеся от прошлых тестовых прогонов.

Так как сквозные тесты оказываются наиболее затратными в отношении сопровождения, лучше выполнять их на более поздней стадии билд-процесса, после прохождения всех юнит- и интеграционных тестов. Возможно даже их выполнение только на билд-сервере, а не на машинах отдельных разработчиков.

Помните, что даже со сквозными тестами не всегда удается решить проблемы со всеми внепроцессными зависимостями. У некоторых зависимостей может не быть тестовой версии или может оказаться невозможно привести эти зависимости в необходимое состояние автоматически. Тогда вам все равно придется использовать тестовые заглушки, а это подчеркивает тот факт, что между интеграционными и сквозными тестами не существует четкой границы.

Итоги

- В этой главе было уточнено определение юнит-теста. Юнит-тест:
 - проверяет правильность работы одной единицы поведения;
 - делает это быстро
 - и в изоляции от других тестов.
- Больше всего разногласий вызывает аспект изоляции. Эти разногласия привели к формированию двух школ юнит-тестирования: классической (детройтской) и лондонской (мокистской). Различия во взглядах на изоляцию также влияют на представления о том, что собой представляет юнит, а также как нужно работать с зависимостями тестируемой системы.
 - Лондонская школа считает, что изолированы друг от друга должны быть юниты (*units under test*) — единицы кода, чаще всего класс. Все его зависимости, за исключением неизменяемых, должны быть заменены тестовыми заглушками в тестах.
 - Классическая школа считает, что изолированы друг от друга должны быть сами юнит-тесты, а не юниты. Кроме того, тестируется единица поведения, а не единица кода. Таким образом, только совместные (*shared*) зависимости должны заменяться тестовыми заглушками. Совместными называются зависимости, предоставляющие тестам возможность влиять на результаты друг друга.
- Основные преимущества лондонской школы — улучшенная детализация, простота тестирования больших графов взаимосвязанных классов и простота нахождения функциональности, содержащей ошибку, при отказе теста.
- Преимущества лондонской школы на первый взгляд кажутся привлекательными. Тем не менее они создают ряд потенциальных проблем. Во-первых, концентрация на единицах кода ошибочна: тесты должны проверять единицы поведения, а не кода. Кроме того, невозможность юнит-тестирования фрагмента кода является хорошим показателем проблем с кодом. Тестовые заглушки не решают эти проблемы, а только скрывают их. И наконец, хотя простота нахождения ошибочного кода полезна, это, как правило, не является серьезной проблемой для классической школы, так как вы почти всегда знаете, что вызвало ошибку, — это тот код, который вы редактировали последним.
- Самой большой проблемой лондонской школы юнит-тестирования является проблема излишней спецификации — привязка тестов к деталям имплементации тестируемой системы.
- Интеграционный тест — это тест, который не удовлетворяет как минимум одному критерию юнит-теста. Сквозные (*end-to-end*) тесты составляют подмножество интеграционных тестов; они проверяют систему с точки зрения

конечного пользователя. Сквозные тесты обращаются напрямую ко всем или почти ко всем внепроцессным зависимостям, с которыми работает ваше приложение.

- Канонический источник информации по классическому стилю — книга Кента Бека (Kent Beck) «Test-Driven Development: By Example»¹. За информацией о лондонском стиле обращайтесь к книге «Growing Object-Oriented Software, Guided by Tests» Стива Фримена (Steve Freeman) и Ната Прайса (Nat Pryce). По теме управления зависимостями я рекомендую книгу «Dependency Injection: Principles, Practices, Patterns» Стивена ван Дьюрзена (Steven van Deursen) и Марка Симана (Mark Seemann)².

¹ *Бек К.* Экстремальное программирование: разработка через тестирование. — СПб.: Питер, 2021. — 224 с.: ил.

² *Симан Марк.* Внедрение зависимостей в .NET. — СПб.: Питер, 2014. — 464 с.: ил.

Анатомия юнит-теста

В этой главе:

- ✓ Структура юнит-теста.
- ✓ Правила выбора имен в юнит-тестировании.
- ✓ Работа с параметризованными тестами.
- ✓ Fluent Assertions.

В этой последней главе части I я расскажу о некоторых базовых вещах. Мы рассмотрим структуру типичного юнит-теста, которая обычно описывается паттерном AAA (*arrange, act, assert* — подготовка, действие и проверка). Также я опишу фреймворк для юнит-тестирования *xUnit* и объясню, почему я использую именно его, а не одного из конкурентов.

Далее речь пойдет о выборе имен в юнит-тестах. На этот счет есть несколько конкурирующих точек зрения, и к сожалению, многие из них работают во вред юнит-тестам. В этой главе я опишу распространенные советы по именованию юнит-тестов и покажу, почему не согласен с ними. Вместо них я опишу альтернативу — простые доступные рекомендации по именованию тестов, которые будут понятными не только программисту, написавшему эти тесты, но и любому другому человеку, знакомому с предметной областью приложения.

Наконец, я опишу некоторые возможности фреймворка, которые способствуют упрощению процесса юнит-тестирования. Эта информация будет специфична для C# и .NET, но большинство фреймворков юнит-тестирования предоставляет аналогичную функциональность независимо от языка программирования. Если вы изучили один фреймворк, то сможете без особых проблем работать с другими.

3.1. Структура юнит-теста

В этом разделе показано, как структурировать юнит-тесты в соответствии с паттерном AAA, каких подводных камней следует избегать и как сделать ваши тесты более читаемыми.

3.1.1. Паттерн AAA

В паттерне AAA каждый тест разбивается на три части: arrange (подготовка), act (действие) и assert (проверка). Также иногда этот паттерн называется 3A. Возьмем для примера класс `Calculator` с методом для вычисления суммы двух чисел:

```
public class Calculator
{
    public double Sum(double first, double second)
    {
        return first + second;
    }
}
```

В листинге 3.1 приведен тест для проверки поведения класса, построенный по схеме AAA.

Листинг 3.1. Тест для метода `Sum` класса `Calculator`

```
public class CalculatorTests
{
    [Fact] ← Атрибут xUnit, обозначающий тест
    public void Sum_of_two_numbers() ← Название юнит-теста
    {
        // Arrange
        double first = 10; ← Секция подготовки
        double second = 20;
        var calculator = new Calculator();

        // Act
        double result = calculator.Sum(first, second); ← Секция действия

        // Assert
        Assert.Equal(30, result); ← Секция проверки
    }
}
```

Паттерн AAA предоставляет простую единообразную структуру для всех тестов в проекте. Это единообразие — одно из самых больших преимуществ паттерна: привыкнув к нему, вы сможете легко прочитать и понять любой тест. Структура теста выглядит так:

- в секции подготовки тестируемая система (system under test, SUT) и ее зависимости приводятся в нужное состояние;

- в секции действия вызываются методы SUT, передаются подготовленные зависимости и сохраняется выходное значение (если оно есть);
- в секции проверки проверяется результат, который может быть представлен возвращаемым значением, итоговым состоянием тестируемой системы и ее колабораторов или методами, которые тестируемая система вызывает у этих колабораторов.

Как правило, написание теста начинается с секции подготовки (`arrange`), так как она предшествует двум другим. Такой подход нормально работает в большинстве случаев, но начинать писать тесты можно также и с секции проверки (`assert`). Если вы практикуете разработку через тестирование (TDD) — то есть когда вы создаете непроходящий тест перед разработкой некоторой функциональности, — вы еще не знаете всего о поведении этой функциональности. Возможно, будет полезно сначала описать, чего вы ожидаете от поведения, а уже затем разбираться, как создать систему для удовлетворения этих ожиданий.

Такой подход может показаться странным, но на самом деле мы именно так подходим к решению задач. Сначала мы думаем о цели: что разрабатываемая функциональность должна делать для нас. Решение задачи начинается уже после этого. Написание проверок до всего остального — не более чем формализация этого мыслительного процесса. Но я еще раз подчеркну, что эта рекомендация применима только в том случае, когда вы практикуете TDD, то есть пишете тесты до рабочего кода. Если вы пишете основной код до кода тестов, то к тому моменту, когда вы доберетесь до теста, вы уже знаете, чего ожидать от поведения, и лучше будет начать с секции подготовки.

ПАТТЕРН «GIVEN-WHEN-THEN»

Возможно, вы также слышали о паттерне «Given-When-Then», похожем на AAA. Этот паттерн также рекомендует разбить тест на три части:

- ✓ Given — соответствует секции подготовки (`arrange`);
- ✓ When — соответствует секции действия (`act`);
- ✓ Then — соответствует секции проверки (`assert`).

В отношении построения теста эти два паттерна ничем не отличаются. Единственное отличие заключается в том, что структура «Given-When-Then» более понятна для не-программиста. Таким образом, она лучше подойдет для тестов, которые вы собираетесь показывать людям, не имеющим технической подготовки.

3.1.2. Избегайте множественных секций `arrange`, `act` и `assert`

Время от времени встречаются тесты с несколькими секциями `arrange` (подготовка), `act` (действие) или `assert` (проверка). Обычно они работают так, как показано на рис. 3.1.

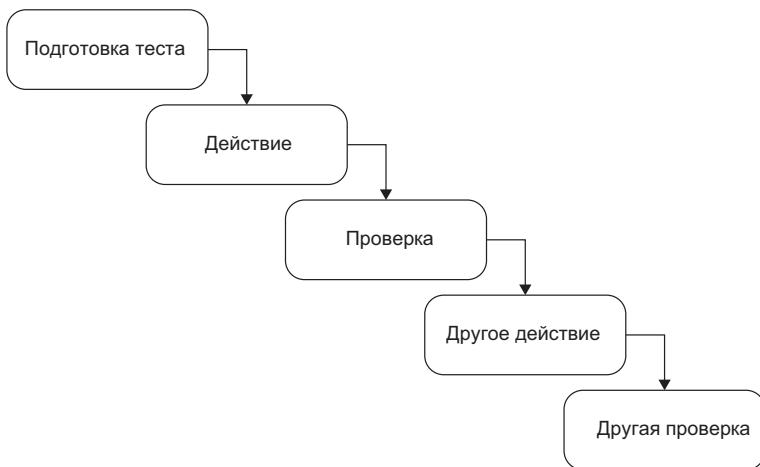


Рис. 3.1. Множественные секции подготовки, действий и проверки указывают на то, что тест пытается проверять слишком много всего. Проблема решается разбиением такого теста на несколько тестов

Когда вы видите несколько секций действий, разделенных секциями проверки и, возможно, секциями подготовки, это означает, что тест проверяет несколько единиц поведения. И как обсуждалось в главе 2, такой тест уже не является юнит-тестом — это интеграционный тест. Такой структуры тестов лучше избегать. Единственное действие гарантирует, что ваши тесты остаются юнит-тестами — то есть остаются простыми, быстрыми и понятными. Если вы видите тест, содержащий серию действий и проверок, отрефакторите его: выделите каждое действие в отдельный тест.

Иногда допустимо иметь несколько секций действий в интеграционных тестах. Как вы, вероятно, помните из предыдущей главы, интеграционные тесты могут быть медленными. Один из способов ускорить их заключается в том, чтобы сгруппировать несколько интеграционных тестов в один с несколькими секциями действий и проверок. Это особенно хорошо ложится на конечные автоматы (state machines), где состояния системы перетекают из одного в другое и когда одна секция действий одновременно служит секцией подготовки для следующей секции действий.

И снова следует подчеркнуть, что этот метод оптимизации применим только к интеграционным тестам — и не ко всем, а только к тем, которые работают медленно, и вы не хотите, чтобы они стали еще медленнее. Для юнит-тестов или интеграционных тестов, которые работают достаточно быстро, такая оптимизация не нужна. Тесты, проверяющие несколько единиц поведения, лучше разбивать на несколько тестов.

3.1.3. Избегайте команд `if` в тестах

Наряду с множественными секциями подготовки, действий и проверки иногда встречаются юнит-тесты, содержащие команду `if`. Это также является антипаттерном.

Тест — неважно, юнит- или интеграционный — должен представлять собой простую последовательность шагов без ветвлений.

Присутствие команды `if` означает, что тест проверяет слишком много всего. Следовательно, такой тест должен быть разбит на несколько тестов. Но в отличие от ситуации с множественными секциями AAA, здесь нет исключений для интеграционных тестов: ветвление в тестах не несет никаких преимуществ. Оно не дает ничего, кроме дополнительных затрат на сопровождение: команды `if` затрудняют чтение и понимание тестов.

3.1.4. Насколько большой должна быть каждая секция?

Типичный вопрос, который нередко задают разработчики при первом знакомстве с паттерном AAA: насколько большой должна быть каждая секция? И как насчет завершающей (teardown) секции — той, которая должна «прибирать» после каждого теста? Существуют несколько рекомендаций, касающихся размеров секций.

Секция подготовки — самая большая

Секция подготовки обычно является самой большой из трех. Ее размер может быть таким же, как секции действия и проверки вместе взятые. Если она становится значительно больше этого, лучше выделить отдельные операции подготовки либо в приватные методы того же класса теста, либо в отдельный класс-фабрику. Два популярных паттерна помогут вам организовать переиспользование кода в секциях подготовки: «Мать объектов» (Object Mother) и «Построитель тестовых данных» (Test Data Builder).

Избегайте секций действий, состоящих из нескольких строк

Секция действия обычно состоит всего из одной строки кода. Если действие состоит из двух и более строк, это может указывать на проблемы с API тестируемой системы.

Этот пункт лучше продемонстрировать на примере; я позаимствую этот пример из главы 2 и продублирую его в листинге 3.2. В этом примере клиент совершает покупку в интернет-магазине.

Листинг 3.2. Однострочная секция действий

```
[Fact]
public void Purchase_succeeds_when_enough_inventory()
{
    // Arrange
    var store = new Store();
    store.AddInventory(Product.Shampoo, 10);
    var customer = new Customer();
```

```
// Act  
bool success = customer.Purchase(store, Product.Shampoo, 5);  
  
// Assert  
Assert.True(success);  
Assert.Equal(5, store.GetInventory(Product.Shampoo));  
}
```

Обратите внимание: секция действия (*act*) в этом тесте состоит из вызова одного метода, что является признаком хорошо спроектированного API класса. Теперь сравните ее с версией из листинга 3.3: на этот раз секция действия состоит из двух строк. Это признак проблемы с API тестируемой системы: он требует, чтобы клиент помнил о необходимости второго вызова метода для завершения покупки, а следовательно, тестируемая система недостаточно инкапсулирована.

Листинг 3.3. Секция действия из двух строк

```
[Fact]  
public void Purchase_succeeds_when_enough_inventory()  
{  
    // Arrange  
    var store = new Store();  
    store.AddInventory(Product.Shampoo, 10);  
    var customer = new Customer();  
  
    // Act  
    bool success = customer.Purchase(store, Product.Shampoo, 5);  
    store.RemoveInventory(success, Product.Shampoo, 5);  
  
    // Assert  
    Assert.True(success);  
    Assert.Equal(5, store.GetInventory(Product.Shampoo));  
}
```

Вот что происходит в секции действий в листинге 3.3:

- в первой строке клиент пытается приобрести пять единиц шампуня в магазине;
- во второй строке товар удаляется со склада. Удаление происходит только в том случае, если предшествующий вызов *Purchase()* завершился успехом.

Недостаток новой версии заключается в том, что она требует двух вызовов для выполнения одной операции. Следует заметить, что это не является проблемой самого теста. Тест проверяет ту же единицу поведения: процесс покупки. Проблема кроется в API класса *Customer*. Он не должен требовать от клиента дополнительного вызова.

С точки зрения бизнеса успешная покупка имеет два результата: покупка продукта клиентом и уменьшение количества товара на складе. Оба результата должны достигаться вместе, что означает, что должен существовать один метод в API класса, решающий обе задачи. В противном случае может быть нарушена логическая

целостность, если клиентский код вызывает первый метод, но не вызывает второй; в этом случае клиент получит товар, но количество товара на складе при этом не уменьшится.

Такое нарушение логической целостности называется *нарушением инварианта* (*invariant violation*). Защита кода от потенциальных нарушений инвариантов называется *инкапсуляцией* (*encapsulation*). Когда нарушение логической целостности проникает в базу данных, оно становится серьезной проблемой; теперь вам не удастся сбросить состояние приложения простым перезапуском. Вам придется разбираться с поврежденными данными в базе и, возможно, связываться с клиентами и решать проблему с каждым из них по отдельности. Представьте, что произойдет, если приложение будет выдавать подтверждения о покупках без резервирования самого товара. Клиенты могут зарезервировать и даже оплатить большее количество товара, чем вы сможете приобрести в ближайшем будущем.

Проблема решается поддержанием инкапсуляции кода. В предыдущих примерах удаление запрашиваемого товара со склада должно быть частью метода `Purchase — customer` не должен полагаться на то, что клиентский код сделает это сам, вызвав метод `store.RemoveInventory`. Когда речь заходит о поддержании инвариантов в системе, вы должны устраниć любую потенциальную возможность нарушить эти инварианты.

Рекомендация о том, что секция действия должна быть не больше одной строки, применима к большинству кода, содержащего бизнес-логику, но в меньшей степени — к служебному или инфраструктурному коду. Иными словами, иногда это правило можно нарушить. Тем не менее отсматривайте каждый такой случай на возможные нарушения инкапсуляции.

3.1.5. Сколько проверок должна содержать секция проверки?

Наконец, остается секция проверки. Возможно, вы слышали о рекомендации, согласно которой каждый тест должен содержать только одну проверку. Она происходит от предпосылки, рассмотренной в главе 2: каждый тест должен покрывать минимально возможный фрагмент кода.

Как вы уже знаете, эта предпосылка неверна. Под «юнитом» в юнит-тестировании понимается единица поведения, а не единица кода. Одна единица поведения может приводить к нескольким результатам; проверять все эти результаты в одном teste вполне нормально.

Тем не менее будьте внимательны с секциями проверки, которые получаются слишком большими: это может быть признаком того, что в коде недостает какой-то абстракции. Например, вместо того чтобы по отдельности проверять все свойства объекта, возвращенного тестируемой системой, возможно, будет лучше добавить методы проверки равенства (*equality members*) в класс такого объекта. После этого объект можно будет сравнивать с ожидаемым значением всего одной командой.

3.1.6. Нужна ли завершающая (teardown) фаза?

Некоторые специалисты также выделяют четвертую – *завершающую (teardown)* – секцию, которая следует после секций подготовки, действия и проверки. Например, в завершающей секции можно удалить любые файлы, созданные в ходе теста, закрыть подключение к базе данных и т. д. Завершение обычно представляется отдельным методом, который переиспользуется всеми тестами в классе. По этой причине я не включил эту фазу в паттерн AAA.

Большинству юнит-тестов завершение не требуется. Юнит-тесты не взаимодействуют с внепроцессными зависимостями, а следовательно, не оставляют за собой ничего, что нужно было бы удалять. Вопрос очистки тестовых данных относится к области интеграционного тестирования. О том, как правильно выполнить завершающие действия после интеграционных тестов, будет рассказано в части III.

3.1.7. Выделение тестируемой системы

Тестируемая система (system under test, SUT) играет важную роль в тестах. Она предоставляет точку входа для поведения, которое вы хотите протестировать. Как обсуждалось в предыдущей главе, это поведение может охватывать несколько классов, а может ограничиваться одним методом. Тем не менее точка входа может быть только одна: один класс, инициирующий данное поведение.

А следовательно, важно отличать тестируемую систему от ее зависимостей (особенно если их достаточно много), чтобы вам не приходилось тратить слишком много времени, выясняя, что есть что в teste. Для этого всегда присваивайте тестируемой системе имя `sut`. В листинге 3.4 показано, как будет выглядеть `CalculatorTests` после переименования экземпляра `Calculator`.

Листинг 3.4. Отделение тестируемой системы от ее зависимостей

```
public class CalculatorTests
{
    [Fact]
    public void Sum_of_two_numbers()
    {
        // Arrange
        double first = 10;
        double second = 20;
        var sut = new Calculator(); ← Переменная calculator
                                            теперь называется sut

        // Act
        double result = sut.Sum(first, second);

        // Assert
        Assert.Equal(30, result);
    }
}
```

3.1.8. Удаление комментариев «arrange/act/assert» из тестов

Отделить тестируемую среду от ее зависимостей важно, но не менее важно отличать три секции (`arrange`, `act` и `assert`) друг от друга, чтобы вам не приходилось подолгу разбираться, к какой секции относится та или иная строка в teste. Для этого можно включить в начало каждой секции комментарий `// Arrange`, `// Act` и `// Assert`. Другой способ основан на разделении секций пустыми строками, как в листинге 3.5.

Листинг 3.5. Разделение секций пустыми строками

```
public class CalculatorTests
{
    [Fact]
    public void Sum_of_two_numbers()
    {
        double first = 10;
        double second = 20;
        var sut = new Calculator(); | Подготовка (arrange)

        double result = sut.Sum(first, second); ← Действие (act)

        Assert.Equal(30, result); ← Проверка (assert)
    }
}
```

Разделение секций пустыми строками хорошо работает в большинстве юнит-тестов. Этот способ позволяет выдержать баланс между краткостью и удобочитаемостью. Впрочем, он не столь эффективен в больших тестах, в которых в секцию подготовки включаются дополнительные пустые строки, разделяющие разные фазы конфигурации. Ситуация особенно характерна для интеграционных тестов, которые часто содержат сложную логику настройки. Таким образом:

- удаляйте комментарии секций в тестах, следующих паттерну AAA, если вы можете избежать вставки дополнительных пустых строк в секциях подготовки и проверки;
- в остальных случаях оставляйте комментарии секций.

3.2. Фреймворк тестирования xUnit

В этом разделе я приведу краткий обзор инструментов для юнит-тестирования, доступных в .NET, и опишу их возможности. Я использую xUnit (<https://github.com/xunit/xunit>) как фреймворк юнит-тестирования (обратите внимание: для запуска тестов xUnit из Visual Studio необходимо установить NuGet-пакет `xunit.runner.visualstudio`). Хотя этот фреймворк работает только в .NET, в каждом объектно-ориентированном языке (Java, C++, JavaScript и т. д.) существуют собственные фреймворки юнит-тестирования, и между ними есть много общего. Если вы работали

с одним таким фреймворком, то сможете без особых проблем работать со всеми остальными.

Даже на одной платформе .NET есть несколько альтернативных вариантов, таких как NUnit (<https://github.com/nunit/nunit>), и встроенный фреймворк MSTest от компании Microsoft. Лично я предпочитаю xUnit по причинам, которые будут описаны ниже, но вы также можете использовать NUnit; эти два фреймворка более или менее равнозначны в отношении функциональности. Тем не менее MSTest я не рекомендую: он не обладает такой гибкостью, как xUnit и NUnit. Даже сами работники Microsoft не используют MSTest. Например, команда ASP.NET Core использует xUnit.

Я предпочитаю xUnit, потому что это более компактная и элегантная версия NUnit. Возможно, вы заметили, что в приводившихся до настоящего момента тестах не было никаких атрибутов, относящихся к тест-фреймворку, кроме атрибута `[Fact]`. Этот атрибут помечает метод класса как юнит-тест, чтобы фреймворк юнит-тестирования знал, что его нужно выполнить. В этих примерах не было атрибутов `[TestFixture]`; в xUnit любой открытый класс может содержать юнит-тест. Не было и атрибутов `[SetUp]` и `[TearDown]`. Если вам нужно переиспользовать логику конфигурации между тестами, вы можете поместить ее в конструктор тест-класса. А если вам необходимо выполнить очистку тестовых данных, вы можете реализовать интерфейс `IDisposable`, как показано в листинге 3.6.

Листинг 3.6. Логика подготовки и завершения, общая для всех тестов

```
public class CalculatorTests : IDisposable
{
    private readonly Calculator _sut;

    public CalculatorTests()
    {
        _sut = new Calculator();           | Вызывается
                                            | до каждого
                                            | теста в классе
    }

    [Fact]
    public void Sum_of_two_numbers()
    {
        /* ... */
    }

    public void Dispose()              | Вызывается
    {
        _sut.CleanUp();                | после каждого
                                            | теста в классе
    }
}
```

Как видите, авторы xUnit постарались упростить работу с фреймворком. Многие концепции, которые ранее требовали дополнительной конфигурации (например,

атрибуты [`TestFixture`] или [`SetUp`]), теперь полагаются на соглашения (conventions) или встроенные языковые конструкции.

Мне особенно нравится атрибут [`Fact`] — именно тем, что он называется `Fact`, а не `Test`. Он подчеркивает правило, упоминавшееся в главе 2: каждый тест должен рассказывать историю. Эта история — отдельный атомарный сценарий или факт, относящийся к предметной области задачи, а прохождение теста показывает, что этот факт является истинным. Если тест не проходит, значит, факт перестал быть истинным и его нужно переписать, либо сама система нуждается в исправлении.

Я рекомендую пользоваться таким подходом при написании юнит-тестов. Ваши тесты не должны ограничиваться простым перечислением того, что делает рабочий код. Вместо этого они должны предоставлять высокоуровневое описание поведения приложения. В идеале это описание должно быть понятным не только программистам, но и бизнесу.

3.3. Переиспользование тестовых данных между тестами

Важно понимать, как и когда переиспользовать код между тестами. Переиспользование кода между секциями подготовки — хороший способ сокращения и упрощения ваших тестов. В этом разделе будет показано, как сделать это правильно.

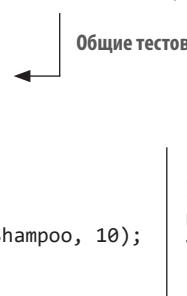
Ранее я упоминал, что подготовка тестовых данных часто занимает много места. Есть смысл выделить эту подготовку в отдельные методы или классы, которые затем переиспользуются между тестами. Существуют два способа реализации такого переиспользования, но я рекомендую использовать только один из них; второй способ приводит к повышению затрат на сопровождение теста.

Первый (неправильный) способ переиспользования тестовых данных — инициализация их в конструкторе теста (или методе, помеченном атрибутом [`SetUp`], если вы используете NUnit), как показано в листинге 3.7.

Листинг 3.7. Выделение кода инициализации в конструктор теста

```
public class CustomerTests
{
    private readonly Store _store;
    private readonly Customer _sut;

    public CustomerTests()
    {
        _store = new Store();
        _store.AddInventory(Product.Shampoo, 10);
        _sut = new Customer();
    }
}
```



Общие тестовые данные

Выполняется перед каждым тестом в классе

```
[Fact]
public void Purchase_succeeds_when_enough_inventory()
{
    bool success = _sut.Purchase(_store, Product.Shampoo, 5);

    Assert.True(success);
    Assert.Equal(5, _store.GetInventory(Product.Shampoo));
}

[Fact]
public void Purchase_fails_when_not_enough_inventory()
{
    bool success = _sut.Purchase(_store, Product.Shampoo, 15);

    Assert.False(success);
    Assert.Equal(10, _store.GetInventory(Product.Shampoo));
}
}
```

Два теста в листинге 3.7 имеют общую логику конфигурации. Они содержат одинаковые секции подготовки, а следовательно, эти секции можно полностью выделить в конструктор `CustomerTests` — именно это и было сделано выше.

Такой подход позволяет значительно сократить объем кода в тестах — вы можете избавиться от большинства (или даже от всех) конфигураций в тестах. Однако у этого подхода есть два серьезных недостатка:

- он создает сильную связность (high coupling) между тестами;
- он ухудшает читаемость тестов.

Обсудим эти недостатки более подробно.

3.3.1. Сильная связность (high coupling) между тестами как антипаттерн

В новой версии, приведенной в листинге 3.7, все тесты связаны друг с другом: изменение логики подготовки одного теста повлияет на все тесты в классе. Например, если заменить строку

```
_store.AddInventory(Product.Shampoo, 10);
```

строкой

```
_store.AddInventory(Product.Shampoo, 15);
```

то тесты, ожидающие 10 единиц шампуня на складе, начнут падать.

Тем самым нарушается важное правило: изменение одного теста не должно влиять на другие тесты. Это правило похоже на то, которое обсуждалось в главе 2:

тесты должны работать в изоляции друг от друга. Тем не менее это не одно и то же правило. В данном случае речь идет о независимом изменении тестов, а не об их независимом выполнении. Оба свойства являются важными атрибутами хорошо спроектированного теста.

Чтобы следовать этим правилам, необходимо избегать совместного состояния (shared state) в классах тестов. Следующие два приватных поля служат примерами такого совместного состояния:

```
private readonly Store _store;
private readonly Customer _sut;
```

3.3.2. Использование конструкторов в тестах ухудшает читаемость

Другой недостаток выделения кода подготовки в конструктор — ухудшение читаемости теста. С таким конструктором просмотр самого теста больше не дает вам полной картины. Чтобы понять, что делает тест, вам придется смотреть в два места: сам тест и конструктор тест-класса.

Даже если логика подготовки данных проста — допустим, только создание экземпляров `Store` и `Customer`, — ее все равно лучше разместить в самом teste. В противном случае вы будете задаваться вопросом, действительно ли здесь создаются только экземпляры тестируемых классов или же происходит также дополнительная их настройка. Автономный тест, не зависящий от конструктора тест-класса, не оставит вам подобной неопределенности.

3.3.3. Более эффективный способ переиспользования тестовых данных

Использование конструктора — не лучший подход к переиспользованию тестовых данных. Второй (правильный) способ — написать фабричные методы, как показано в листинге 3.8.

Листинг 3.8. Выделение общего кода инициализации в приватные фабричные методы

```
public class CustomerTests
{
    [Fact]
    public void Purchase_succeeds_when_enough_inventory()
    {
        Store store = CreateStoreWithInventory(Product.Shampoo, 10);
        Customer sut = CreateCustomer();

        bool success = sut.Purchase(store, Product.Shampoo, 5);

        Assert.True(success);
    }
}
```

```
        Assert.Equal(5, store.GetInventory(Product.Shampoo));
    }

    [Fact]
    public void Purchase_fails_when_not_enough_inventory()
    {
        Store store = CreateStoreWithInventory(Product.Shampoo, 10);
        Customer sut = CreateCustomer();

        bool success = sut.Purchase(store, Product.Shampoo, 15);

        Assert.False(success);
        Assert.Equal(10, store.GetInventory(Product.Shampoo));
    }

    private Store CreateStoreWithInventory(
        Product product, int quantity)
    {
        Store store = new Store();
        store.AddInventory(product, quantity);
        return store;
    }

    private static Customer CreateCustomer()
    {
        return new Customer();
    }
}
```

Выделяя общий код инициализации в приватные фабричные методы, можно сократить код теста с сохранением полного контекста того, что происходит в этом тесте. Более того, приватные методы не связывают тесты друг с другом, при условии что вы сделаете их достаточно гибкими (то есть позволите тестам указать, как должны создаваться тестовые данные).

К примеру, возьмем следующую строку:

```
Store store = CreateStoreWithInventory(Product.Shampoo, 10);
```

Здесь тест явно указывает, что магазин должен содержать 10 единиц шампуня. Код получается одновременно и читаемым, и пригодным для переиспользования. Он читаем, потому что вам не приходится изучать внутреннее устройство фабричного метода, для того чтобы понять атрибуты созданного магазина. Он пригоден для переиспользования, потому что этот метод также можно использовать в других тестах.

Обратите внимание: в этом конкретном примере писать фабричные методы не обязательно, так как логика подготовки весьма проста. Этот код приводится исключительно в демонстрационных целях.

У правила о переиспользовании тестовых данных есть одно исключение. Вы можете создавать тестовые данные в конструкторе в случае, если эти данные используются

всеми или почти всеми тестами в проекте. Такая ситуация часто встречается с интеграционными тестами, которые работают с базой данных. Все такие тесты требуют подключения к базе данных, код инициализации которой можно написать один раз и потом переиспользовать во всех интеграционных тестах. Но даже в этом случае будет разумнее добавить базовый класс и инициализировать базу данных в конструкторе этого класса, а не в отдельных классах тестов. Пример общего кода инициализации в базовом классе приведен в листинге 3.9.

Листинг 3.9. Общий код инициализации в базовом классе

```
public class CustomerTests : IntegrationTests
{
    [Fact]
    public void Purchase_succeeds_when_enough_inventory()
    {
        /* Здесь используется _database */
    }
}

public abstract class IntegrationTests : IDisposable
{
    protected readonly Database _database;

    protected IntegrationTests()
    {
        _database = new Database();
    }

    public void Dispose()
    {
        _database.Dispose();
    }
}
```

Обратите внимание на то, что класс `CustomerTests` остается без конструктора. Он получает доступ к экземпляру `_database`, наследуя его от базового класса `IntegrationTests`.

3.4. Именование юнит-тестов

Очень важно присваивать вашим тестам осмысленные имена. Правильное именование помогает понять, что проверяет тест и как работает система.

Как же выбрать имя для юнит-теста? Есть много рекомендаций на эту тему. Одна из самых распространенных (и, пожалуй, одна из наименее полезных) рекомендаций выглядит так:

[*ТестируемыйМетод*]_[*Сценарий*]_[*ОжидаемыйРезультат*]

где:

- *ТестируемыйМетод* — имя тестируемого метода;
- *Сценарий* — состояние системы, при котором тестируется метод;
- *ОжидаемыйРезультат* — что ожидается от тестируемого метода.

Такая схема неоптимальна, так как она заставляет вас сосредоточиться на деталях имплементации вместо поведения системы.

Простые фразы гораздо лучше подходят для этой задачи: они более выразительны и не ограничивают вас в выборе имен для тестов. Простыми фразами можно описать поведение системы в формулировках, которые будут понятны всем, в том числе заказчику или аналитику. Для примера теста с хорошим названием вернемся к коду из листинга 3.5:

```
public class CalculatorTests
{
    [Fact]
    public void Sum_of_two_numbers()
    {
        double first = 10;
        double second = 20;
        var sut = new Calculator();

        double result = sut.Sum(first, second);

        Assert.Equal(30, result);
    }
}
```

Как переписать имя теста (`Sum_of_two_numbers`) в схеме [*ТестируемыйМетод*]_[*Сценарий*]_[*ОжидаемыйРезультат*]? Вероятно, как-то так:

```
public void Sum_TwoNumbers_ReturnsSum()
```

Тестируемый метод — `Sum`, сценарий описывает два числа, а ожидаемым результатом является сумма этих двух чисел. Такое имя выглядит логично с точки зрения программиста, но как оно выглядит с точки зрения читаемости теста? Довольно плохо. Для непрограммиста это и вовсе китайская грамота. Только подумайте: почему `Sum` дважды встречается в имени теста? О чём говорит слово `Returns`, мы что-то куда-то возвращаем? Непонятно.

Можно возразить: неважно, что непрограммист не поймет названия этого имени. В конце концов, юнит-тесты пишутся программистами для программистов, а не для бизнеса или аналитиков. Программисты довольно хорошо справляются с расшифровкой непонятных имен — это их работа.

Это верно, но только до определенной степени. Непонятные названия создают дополнительную когнитивную нагрузку для всех, независимо от того, программисты они или

нет. Читателю приходится прикладывать дополнительные усилия, чтобы определить, что именно проверяет тест и как он связан с бизнес-требованиями. На первый взгляд это не кажется серьезной проблемой, но эта когнитивная нагрузка дает о себе знать в долгосрочной перспективе. Все это медленно, но верно увеличивает затраты на сопровождение ваших тестов. Это особенно заметно, когда вы возвращаетесь к тесту после того, как забыли подробности тестируемой функциональности, или попытаетесь разобраться в teste, написанном коллегой. Чтение чужого кода и без того является достаточно сложным делом, и любая помощь в этом приносит заметную пользу.

Еще раз приведу две версии:

```
public void Sum_of_two_numbers()
public void Sum_TwoNumbers_ReturnsSum()
```

Первое имя читается намного проще. Оно описывает тестируемое поведение доступным и понятным языком.

3.4.1. Рекомендации по именованию юнит-тестов

Для написания выразительных, читаемых имен тестов:

- не следуйте жесткой структуре именования тестов. Высокоуровневое описание сложного поведения не удастся втиснуть в узкие рамки такой структуры. Сохраняйте свободу самовыражения;
- выбирайте имя теста так, словно вы описываете сценарий не-программисту, знакомому с предметной областью задачи (например, бизнес-аналитику);
- разделяйте слова символами подчеркивания. Это поможет улучшить читаемость, особенно длинных имен.

Обратите внимание, что я не использую подчеркивания в имени класса теста `CalculatorTests`. Обычно имена классов имеют меньшую длину и normally читаются без подчеркиваний.

Также обратите внимание на то, что хотя я использую паттерн `[ИмяКласса]Tests` при выборе имен классов тестов, это не означает, что тесты ограничиваются проверкой только этого класса. Вспомните, что юнитом в юнит-тестировании является единица поведения, а не класс. Единица поведения может охватывать один или несколько классов; фактический размер не имеет значения. Рассматривайте класс в `[ИмяКласса]Tests` как точку входа — API, при помощи которого можно проверить единицу поведения.

3.4.2. Пример: переименование теста в соответствии с рекомендациями

Возьмем тест в качестве примера и попробуем постепенно доработать его название в соответствии с моими рекомендациями. В листинге 3.10 приведен тест, который

проверяет, что доставка с уже прошедшей датой недопустима. Имя теста записано в соответствии со структурой, ухудшающей читаемость теста.

Листинг 3.10. Тест с именем, заданным по жесткой схеме

```
[Fact]
public void IsDeliveryValid_InvalidDate_ReturnsFalse()
{
    DeliveryService sut = new DeliveryService();
    DateTime pastDate = DateTime.Now.AddDays(-1);
    Delivery delivery = new Delivery
    {
        Date = pastDate
    };

    bool isValid = sut.IsDeliveryValid(delivery);

    Assert.False(isValid);
}
```

Этот тест проверяет, что `DeliveryService` распознает доставку с некорректной датой как недопустимую. Как бы вы переписали название этого теста? Следующий вариант будет неплохой первой попыткой:

```
public void Delivery_with_invalid_date_should_be_considered_invalid()
```

Обратите внимание на две особенности новой версии:

- это имя является осмысленным для непрограммиста, а это означает, что программисту тоже будет проще понять его;
- название тестируемого метода — `IsDeliveryValid` — уже не является частью имени теста.

ИМЯ ТЕСТИРУЕМОГО МЕТОДА В НАЗВАНИИ ТЕСТА

Не включайте имя тестируемого метода в название теста.

Помните, что мы тестируем не код, а поведение системы. Следовательно, имя тестируемого метода не важно. Как упоминалось ранее, тестируемая система является точкой входа — средством активизации тестируемого поведения. Вы можете решить присвоить тестируемому методу другое имя — скажем, `IsDeliveryCorrect`, и оно никак не повлияет на поведение тестируемой системы. С другой стороны, если следовать исходным соглашениям об именах, тест придется переименовать. Это еще раз подчеркивает, что ориентация на код вместо поведения привязывает тесты к деталям имплементации этого кода, что отрицательно влияет на стоимость сопровождения тестов. Подробнее об этом в главе 5.

Единственное исключение из этого правила — работа со вспомогательным кодом. Такой код не содержит бизнес-логики — его поведение почти не выходит за рамки простой вспомогательной функциональности. В таких случаях допускается использование имен тестируемых методов.

Второй пункт является естественным следствием переписывания названия теста на естественном языке, поэтому его легко упустить из виду. Тем не менее это важное следствие, которое заслуживает отдельного внимания.

Но вернемся к нашему примеру. Новая версия названия теста — хорошее начало, но ее можно улучшить. Что именно означает некорректная дата доставки? Из теста в листинге 3.10 видно, что недопустимой считается любая дата в прошлом. И это разумно — система должна разрешать выбрать дату доставки в будущем, но не в прошлом.

Отразим эту информацию в названии теста:

```
public void Delivery_with_past_date_should_be_considered_invalid()
```

Уже лучше, но не идеально — слишком длинно. От слова «considered» можно избавиться без потери смысла:

```
public void Delivery_with_past_date_should_be_invalid()
```

Формулировку *should be* («должно быть») следует рассматривать как еще один распространенный антипаттерн. Ранее в этой главе я упоминал, что тест представляет собой атомарный факт, относящийся к единице поведения. При изложении факта нет места пожеланиям. Замените *should be* на *is*:

```
public void Delivery_with_past_date_is_invalid()
```

Наконец, не стоит игнорировать правила грамматики. Артикли упрощают чтение текста; включите артикль *a* в имя теста:

```
public void Delivery_with_a_past_date_is_invalid()
```

Вот и все. Итоговая версия представляет собой изложение факта, которое сразу переходит к сути дела и описывает один из аспектов тестируемого поведения, в этом конкретном случае — аспект определения возможности доставки.

3.5. Параметризованные тесты

Одного теста обычно оказывается недостаточно для полного описания единицы поведения. Обычно такая единица состоит из нескольких компонентов, каждый из которых должен быть отражен в отдельном тесте. Если поведение достаточно сложно, то количество описывающих его тестов может резко вырасти и стать сложным в сопровождении. К счастью, многие фреймворки юнит-тестирования предоставляют функциональность, которая позволяет группировать похожие тесты с использованием параметризованных тестов (рис. 3.2).

В этом разделе я сначала покажу все компоненты поведения, описанные отдельными тестами, а затем продемонстрирую, как эти тесты можно сгруппировать друг с другом.

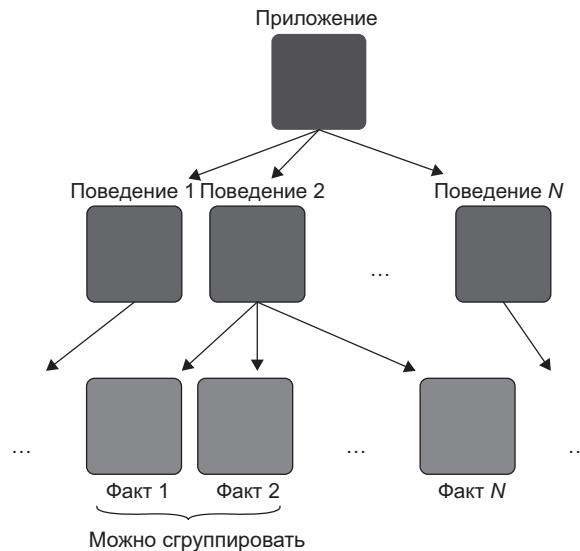


Рис. 3.2. Типичное приложение содержит несколько единиц поведения. Чем сложнее поведение, тем больше фактов требуется для его полного описания. Каждый факт представляется тестом. Похожие факты могут группироваться в одном тестовом методе при помощи параметризованных тестов

Допустим, функциональность доставки работает так, что ближайшая доставка возможна не менее через два дня. Очевидно, одного уже написанного нами теста недостаточно для описания этого поведения. Кроме теста, который проверяет прошлую дату доставки, нам также понадобятся тесты для проверки сегодняшней, завтрашней и послезавтрашней даты.

Существующий тест называется `Delivery_with_a_past_date_is_invalid`. К нему добавляются еще три:

```
public void Delivery_for_today_is_invalid()
public void Delivery_for_tomorrow_is_invalid()
public void The_soonest_delivery_date_is_two_days_from_now()
```

Но тогда у нас появятся четыре тестовых метода, которые различаются только датой доставки.

Правильнее будет сгруппировать эти тесты, чтобы сократить размер тестового кода. В xUnit (как и во многих других фреймворках) есть возможность *параметризации* тестов. В листинге 3.11 показано, как работает такая группировка. Каждый атрибут `InlineData` представляет отдельный факт о системе; это отдельный тестовый сценарий.

Каждый факт теперь представляется строкой `[InlineData]` вместо отдельного теста. Я также переименовал тестовый метод и присвоил ему более общее название; в нем уже не упоминается, какая дата должна считаться допустимой или недопустимой.

Листинг 3.11. Тест, охватывающий сразу несколько фактов

```
public class DeliveryServiceTests
{
    [InlineData(-1, false)]
    [InlineData(0, false)]
    [InlineData(1, false)]
    [InlineData(2, true)]
    [Theory]
    public void Can_detect_an_invalid_delivery_date(
        int daysFromNow,
        bool expected)
    {
        DeliveryService sut = new DeliveryService();
        DateTime deliveryDate = DateTime.Now
            .AddDays(daysFromNow);
        Delivery delivery = new Delivery
        {
            Date = deliveryDate
        };
        bool isValid = sut.IsDeliveryValid(delivery);
        Assert.Equal(expected, isValid);
    }
}
```

The diagram shows annotations explaining the code's behavior:

- Атрибут `InlineData` передает тестовому методу набор входных значений. Каждая строка представляет отдельный факт, относящийся к поведению.** This annotation points to the first four lines of the `[InlineData]` block.
- Параметры, которым атрибуты присваивают входные значения** This annotation points to the vertical bar between the `[InlineData]` block and the `daysFromNow` parameter in the `Can_detect_an_invalid_delivery_date` method.
- Использование этих параметров** This annotation points to the two arrows originating from the `daysFromNow` and `expected` parameters in the method signature, which point to their respective assignments in the `InlineData` block.

СОВЕТ

Обратите внимание на использование атрибута `[Theory]` вместо `[Fact]`. Теория (*theory*) — набор фактов, относящихся к поведению.

Параметризованные тесты позволяют значительно сократить объем тестового кода, но такой подход также имеет и недостатки. Становится труднее понять, какие факты описывает тестовый метод. И чем больше параметров в teste, тем сложнее это сделать. В качестве компромиссного решения можно выделить позитивный тестовый сценарий в отдельный тест и извлечь пользу из содержательного имени там, где это важнее всего — при определении критерия, по которому различаются допустимые и недопустимые даты поставки, как показано в листинге 3.12.

Листинг 3.12. Два теста для проверки позитивного и негативного сценария

```
public class DeliveryServiceTests
{
    [InlineData(-1)]
    [InlineData(0)]
    [InlineData(1)]
    [Theory]
    public void Can_detect_an_invalid_delivery_date(int daysFromNow)
    {
```

```
    /* ... */
}

[Fact]
public void The_soonest_delivery_date_is_two_days_from_now()
{
    /* ... */
}
}
```

Такой подход также упрощает негативные тестовые сценарии, так как из тестового метода можно исключить параметр `expected`. И конечно, позитивный тестовый метод тоже можно преобразовать в параметризованный тест для проверки разных дат.

Как видите, существует компромисс между объемом тестового кода и его читаемостью. Как правило, стоит объединять позитивные и негативные тестовые сценарии в одном методе только тогда, когда по входным параметрам можно легко определить, какой сценарий чему соответствует. В противном случае негативные и позитивные тестовые сценарии следует разделить. А если поведение становится слишком сложным, вообще не используйте параметризованные тесты. В таком случае каждый негативный и позитивный тестовый сценарий лучше описать отдельным тестом.

3.5.1. Генерирование данных для параметризованных тестов

При использовании параметризованных тестов (по крайней мере, в .NET) существует ряд нюансов, о которых следует знать. В листинге 3.11 я использовал параметр `daysFromNow` для передачи входных данных тестовому методу. Почему не реальную дату и время? К сожалению, следующий код работать не будет:

```
[InlineData(DateTime.Now.AddDays(-1), false)]
[InlineData(DateTime.Now, false)]
[InlineData(DateTime.Now.AddDays(1), false)]
[InlineData(DateTime.Now.AddDays(2), true)]
[Theory]
public void Can_detect_an_invalid_delivery_date(
    DateTime deliveryDate,
    bool expected)
{
    DeliveryService sut = new DeliveryService();
    Delivery delivery = new Delivery
    {
        Date = deliveryDate
    };

    bool isValid = sut.IsDeliveryValid(delivery);

    Assert.Equal(expected, isValid);
}
```

В C# содержимое всех атрибутов вычисляется во время компиляции. Вы должны использовать только значения, понятные компилятору, то есть:

- константы;
- литералы;
- выражения `typeof()`.

Вызов `DateTime.Now` зависит от исполнительной системы .NET, и поэтому недопустим.

Впрочем, эту проблему можно обойти. В xUnit есть функциональность для генерации данных, передаваемых тесту: `[MemberData]`. Листинг 3.13 показывает, как переписать предыдущий тест с использованием этой функциональности.

Листинг 3.13. Генерирование сложных данных для параметризованного теста

```
[Theory]
[MemberData(nameof(Data))]
public void Can_detect_an_invalid_delivery_date(
    DateTime deliveryDate,
    bool expected)
{
    /* ... */
}

public static List<object[]> Data()
{
    return new List<object[]>
    {
        new object[] { DateTime.Now.AddDays(-1), false },
        new object[] { DateTime.Now, false },
        new object[] { DateTime.Now.AddDays(1), false },
        new object[] { DateTime.Now.AddDays(2), true }
    };
}
```

`MemberData` получает имя статического метода, генерирующего коллекцию входных данных (компилятор преобразует `nameof(Data)` в литерал "Data"). Каждый элемент коллекции сам по себе является коллекцией, соответствующей двум входным параметрам: `deliveryDate` и `expected`. С помощью этой функциональности в тест можно передавать параметры любого типа.

3.6. Использование библиотек для дальнейшего улучшения читаемости тестов

Также для улучшения читаемости теста можно воспользоваться assertion-библиотекой. Лично я предпочитаю библиотеку Fluent Assertions (<https://fluentassertions.com>), но в .NET есть и другие библиотеки с подобным функционалом.

Главное преимущество assertion-библиотеки в том, что она делает проверки в тестах более читаемыми. Возьмем один из предшествующих тестов:

```
[Fact]
public void Sum_of_two_numbers()
{
    var sut = new Calculator();

    double result = sut.Sum(10, 20);

    Assert.Equal(30, result);
}
```

Теперь сравните со следующим фрагментом, использующим Fluent Assertions:

```
[Fact]
public void Sum_of_two_numbers()
{
    var sut = new Calculator();

    double result = sut.Sum(10, 20);

    result.Should().Be(30);
}
```

Проверка из второго теста (`result.Should().Be(30)`) читается намного проще. Мы предпочитаем усваивать информацию в форме рассказов. Все рассказы строятся по определенной схеме:

[субъект] [действие] [объект].

Например:

Боб открыл дверь.

Здесь *Боб* — субъект, *открыл* — действие, а *дверь* — объект. То же правило применяется к коду. `result.Should().Be(30)` читается лучше, чем `Assert.Equal(30, result)` именно потому, что это выражение строится как рассказ. Это простой рассказ, в котором `result` — субъект, `should be` — действие, а `30` — объект.

ПРИМЕЧАНИЕ

Одним из факторов успеха парадигмы объектно-ориентированного программирования (ООП) стала именно улучшенная читаемость кода. В ООП вы тоже можете структурировать свой код так, чтобы он читался как рассказ.

Библиотека Fluent Assertions также предоставляет многочисленные вспомогательные методы для проверки чисел, строк, коллекций, даты и времени и т. д. Единственный недостаток заключается в том, что такая библиотека становится дополнительной

зависимостью, которую необходимо включать в проект (хотя она включается только на стадии разработки и не будет поставляться в продуктив).

Итоги

- Все юнит-тесты должны строиться по схеме AAA: подготовка (Arrange), действие (Act), проверка (Assert). Если тест состоит из нескольких секций подготовки, действий или проверки, это указывает на то, что тест проверяет сразу несколько единиц поведения. Если этот тест — юнит-тест, разбейте его на несколько тестов: по одному для каждого действия.
- Секция действия, содержащая более одной строки, — признак проблем с API тестируемой системы. Клиент должен не забывать выполнять эти действия совместно, чтобы не привести к нарушению логической целостности. Такие нарушения называются *нарушениями инвариантов*. Защита вашего кода от потенциальных нарушений инвариантов называется инкапсуляцией.
- Чтобы выделить тестируемую систему в тестах, присвойте ей имя `sut`. Чтобы разделить три секции теста, либо включите в них соответствующий комментарий (`Arrange`, `Act`, `Assert`), либо вставьте пустые строки между секциями.
- Переиспользование кода инициализации тестовых данных должно осуществляться с помощью фабричных методов (вместо конструктора тест-класса). Такой подход поддерживает изоляцию между тестами и улучшает читаемость.
- Не используйте жесткую структуру именования тестов. Присваивайте имена тестам так, как если бы вы описывали сценарий непрограммисту, знакомому с предметной областью. Разделяйте слова в имени подчеркиваниями и не включайте имя тестируемого метода в название теста.
- Параметризованные тесты помогают сократить объем кода, необходимого для похожих тестов. Недостаток параметризованных тестов — ухудшение читаемости, так как тесты становятся более общими.
- Assertion-библиотеки помогают улучшить читаемость кода за счет реструктуризации порядка слов в проверках в тестах.

Часть II

Обеспечение эффективной работы ваших тестов

Итак, теперь вы знаете, для чего нужно юнит-тестирование, и мы можем перейти к сути — выяснить, какими свойствами обладает хороший тест, и научиться проводить рефакторинг тестов для повышения их эффективности. В главе 4 рассматриваются четыре аспекта, на которых строятся хорошие юнит-тесты. Эти четыре аспекта формируют общую систему координат, которая будет использоваться для анализа юнит-тестов и методов тестирования в остальной части книги.

Глава 5 использует систему координат, установленную в главе 4, и рассказывает о моках и о том, почему их использование так часто приводит к хрупким тестам.

В главе 6 та же система координат используется для анализа трех стилей юнит-тестирования. Вы узнаете, какие из этих стилей обычно приводят к написанию тестов наилучшего качества и почему.

В главе 7 материал глав 4–6 применяется на практике. Вы узнаете, как провести рефакторинг переусложненных тестов, с тем чтобы улучшить их эффективность и снизить затраты на сопровождение.

Четыре аспекта хороших юнит-тестов

В этой главе:

- ✓ Нахождение баланса между различными аспектами хороших юнит-тестов.
- ✓ Что такое идеальный тест.
- ✓ Пирамида тестирования.
- ✓ Тестирование по принципу «черного ящика» и «белого ящика».

В главе 1 были перечислены свойства хороших юнит-тестов.

- Они интегрированы в цикл разработки. Пользу приносят только те тесты, которые вы активно используете; иначе писать их нет смысла.
- Они тестируют только самые важные части вашего кода. Не весь рабочий код заслуживает одинакового внимания. Важно отличать бизнес-логику приложения (его доменную модель) от всего остального. Эта тема рассматривается в главе 7.
- Они дают максимальную защиту от багов с минимальными затратами на сопровождение. Для этого вы должны уметь:
 - распознавать эффективные тесты (и по аналогии — тесты с низкой эффективностью);
 - писать эффективные тесты.

Как обсуждалось в главе 1, умение распознать эффективный тест и написать его — два разных навыка. Для второго навыка необходимо сначала освоить первый, поэтому

в этой главе я покажу, как распознать эффективный тест. Вы познакомитесь с системой координат, с помощью которой можно анализировать любой тест в проекте. Затем эта система координат будет использована для рассмотрения некоторых популярных концепций юнит-тестирования: пирамиды тестирования и тестирования по принципу «черного ящика»/«белого ящика».

4.1. Четыре аспекта хороших юнит-тестов

Хороший юнит-тест должен обладать следующими четырьмя атрибутами:

- защита от багов;
- устойчивость к рефакторингу;
- быстрая обратная связь;
- простота поддержки.

Эти четыре атрибута фундаментальны. Они могут использоваться для анализа любых автоматизированных тестов, будь то юнит-, интеграционные или сквозные (end-to-end) тесты. В каждом teste до той или иной степени проявляется каждый из четырех атрибутов. В этом разделе я расскажу о первых двух; в разделе 4.2 я опишу связь между ними.

4.1.1. Первый аспект: защита от багов

Начнем с первого атрибута хорошего юнит-теста: *защиты от багов*. Как упоминалось в главе 1, баг (или регрессия) — это программная ошибка. Как правило, такие ошибки возникают после внесения изменений в код — обычно после написания новой функциональности.

Баги сами по себе могут привести к неприятным последствиям, но хуже всего то, что чем больше функциональности вы разрабатываете, тем выше вероятность того, что вы внесете баг в новую версию. Вопреки распространенному мнению, *код — это не актив, а обязательство*. Чем больше кода в проекте, тем больше вероятность того, что в нем найдется ошибка. Вот почему так важно разработать хорошую защиту от багов. Без такой защиты вы не сможете обеспечить рост проекта в долгосрочной перспективе из-за постоянно увеличивающегося количества ошибок.

Чтобы оценить, насколько хорошо тест проявляет себя в отношении защиты от багов, необходимо принять во внимание следующее:

- объем кода, выполняемого тестом;
- сложность этого кода;
- важность этого кода с точки зрения бизнес-логики.

Как правило, чем больше кода тест выполняет, тем выше вероятность выявить в нем баг (если, конечно, он там есть). Само собой, тест также должен иметь актуальный набор проверок (assertions), просто выполнить код недостаточно.

Важен не только объем кода, но и его сложность и важность с точки зрения бизнес-логики. Код, содержащий сложную бизнес-логику, важнее инфраструктурного кода — ошибки в критичной для бизнеса функциональности наносят наибольший ущерб.

Как следствие, тестирование тривиального кода обычно не имеет смысла. Этот код слишком простой и не содержит сколько-нибудь значительного объема бизнес-логики. В тестах, покрывающих тривиальный код, вероятность нахождения ошибок невелика. Примером тривиального кода служит одностороннее свойство следующего вида:

```
public class User
{
    public string Name { get; set; }
}
```

Более того, помимо вашего кода также должен учитываться код, который был написан не вами: например, библиотеки, фреймворки и любые внешние системы, используемые приложением. Этот код влияет на работу вашего кода почти в такой же степени, как и ваш собственный код. Для обеспечения оптимальной защиты тест должен проверять, как ваш код работает в комбинации с этими библиотеками, фреймворками и внешними системами.

СОВЕТ

Чтобы максимизировать метрику защиты от багов, тест должен выполнять максимально возможный объема кода.

4.1.2. Второй аспект: устойчивость к рефакторингу

Второй атрибут хорошего юнит-теста — устойчивость к рефакторингу. Эта устойчивость определяет, насколько хорошо тест может пережить рефакторинг тестируемого им кода без выдачи ошибок.

ОПРЕДЕЛЕНИЕ

Рефакторингом называется модификация существующего кода без изменения его наблюдаемого поведения. Обычно рефакторинг проводится для улучшения нефункциональных характеристик кода: читаемости и простоты. Примеры рефакторинга — переименование метода или выделение фрагмента кода в новый класс.

Представьте себе такую ситуацию. Вы разработали новую функциональность, тесты проходят, все хорошо. Теперь вы решаете подчистить код. Вы проводите рефакторинг, вносите небольшие изменения тут и там, и все выглядит еще лучше, чем прежде...

Кроме одного обстоятельства — тесты перестали проходить. Вы начинаете смотреть, что же именно сломалось при рефакторинге, но оказывается, что все в порядке. Новая функциональность работает так же хорошо, как и прежде. Это тесты были написаны так, что они падают при любом изменении тестируемого кода. И это происходит независимо от того, внесли вы ошибку в этот код или нет.

Такая ситуация называется *ложным срабатыванием*. Это ложный сигнал тревоги: тест показывает, что функциональность не работает, тогда как в действительности все работает как положено. Такие ложные срабатывания обычно происходят при рефакторинге кода, когда вы изменяете имплементацию, но оставляете поведение приложения без изменений. Отсюда и название этого атрибута: устойчивость к рефакторингу.

Чтобы оценить, насколько тест хорош в плане устойчивости к рефакторингу, смотрите на то, сколько этот тест выдает ложных срабатываний: чем меньше, тем лучше.

Почему столько внимания уделяется ложным срабатываниям? Потому что они могут иметь серьезные последствия для всего приложения. Как говорилось в главе 1, целью юнит-тестирования является обеспечение устойчивого роста проекта. Устойчивый рост становится возможным благодаря тому, что тесты позволяют добавлять новую функциональность и проводить регулярный рефакторинг без внесения ошибок в код. Здесь имеются два конкретных преимущества:

- *Тесты становятся системой раннего предупреждения при поломке существующей функциональности.* Благодаря таким ранним предупреждениям вы можете устранить ошибки задолго до того, как ошибочный код будет развернут в продуктиве, где исправление ошибок потребует значительно больших усилий.
- *Вы получаете уверенность в том, что изменения в вашем коде не приведут к багам.* Без такой уверенности вы будете проводить гораздо меньше рефакторинга, что в свою очередь приведет к постепенному ухудшению качества кода проекта.

Ложные срабатывания негативно влияют на оба эти преимущества:

- Если тесты падают без веской причины, они притупляют вашу готовность реагировать на проблемы в коде. Со временем вы привыкаете к таким сбоям и перестаете обращать на них внимание. А это может привести к игнорированию настоящих ошибок, которые затем попадают в продуктив.
- С другой стороны, при частых ложных срабатываниях вы начинаете все меньше и меньше доверять вашим тестам. Они уже не воспринимаются как что-то, на что вы можете положиться. Отсутствие доверия приводит к уменьшению рефакторинга, так как вы пытаетесь свести к минимуму потенциальные ошибки.

Эта история типична для большинства проектов с хрупкими тестами. Сначала разработчики серьезно относятся к падениям тестов и стараются их починить. Через какое-то время они устают от того, что тесты постоянно поднимают тревогу, и все чаще игнорируют их. Рано или поздно наступает момент, когда в продуктив попадают настоящие ошибки, так как разработчики проигнорировали их вместе с ложными срабатываниями.

В такой ситуации не следует полностью отказываться от любых попыток рефакторинга. Правильный подход здесь — проанализировать тесты в проекте и приступить к снижению их хрупкости. Эта тема рассматривается в главе 7.

ИСТОРИЯ С ПОЛЕЙ

Однажды я работал над проектом с богатой историей. Проект был не слишком старым — два или три года; но за это время разработка успела серьезно изменить направление. С этими изменениями появилась проблема: в кодовой базе стали накапливаться большие блоки унаследованного кода, который никому не хватало смелости удалить или переработать. Компании была не нужна функциональность, предоставлявшаяся этим кодом, но некоторые его части использовались в новой функциональности, и полностью избавиться от старого кода было невозможно.

Проект имел хорошее тестовое покрытие. Но каждый раз, когда кто-то пытался провести рефакторинг старой функциональности и отделить части, которые продолжали использоваться, от всего остального, тесты начинали падать. И не только старые тесты, которые были отключены уже давно, но и новые тоже. Некоторые падения были оправданными, но большей частью это были ложные срабатывания.

Поначалу разработчики пытались справиться с этими падениями. Тем не менее, поскольку большинство из них были ложными, ситуация дошла до того, что разработчики стали игнорировать такие срабатывания и отключать непроходящие тесты.

Какое-то время это работало — пока в продуктив не попала серьезная ошибка. Один из тестов правильно выявил эту ошибку, но никто не обратил внимания; тест был отключен вместе со всеми остальными. После этого случая разработчики вообще перестали прикасаться к старому коду.

4.1.3. Что приводит к ложным срабатываниям?

Итак, что же становится причиной ложных срабатываний? И как их избежать?

Количество ложных срабатываний, выданных тестом, напрямую связано со структурой этого теста. Чем сильнее тест связан с деталями имплементации тестируемой системы (*system under test*, SUT), тем больше ложных срабатываний он порождает. Уменьшить количество ложных срабатываний можно только одним способом: отвязав тест от деталей имплементации тестируемой системы. Тест должен проверять конечный результат — наблюдаемое поведение тестируемой системы, а не действия, которые она совершает для достижения этого результата. Тесты должны подходить к проверке SUT с точки зрения конечного пользователя и проверять только результат, имеющий смысл для этого пользователя. Все остальное следует отбросить (подробнее об этом в главе 5).

Лучший вариант структурирования теста — тот, при котором он рассказывает историю о предметной области. Если такой тест не проходит, это означает, что между историей и фактическим поведением приложения существует разрыв. Только такие

падения тестов полезны — они всегда несут полезную информацию о том, что пошло не так. Все остальные сбои — отвлекающий шум.

Рассмотрим пример из листинга 4.1. Класс `MessageRenderer` генерирует HTML-представление для сообщений, состоящих из заголовка, тела и колонтитула (footer).

Листинг 4.1. Генерирование HTML-представления сообщения

```
public class Message
{
    public string Header { get; set; }
    public string Body { get; set; }
    public string Footer { get; set; }
}

public interface IRenderer
{
    string Render(Message message);
}

public class MessageRenderer : IRenderer
{
    public IReadOnlyList<IRenderer> SubRenderers { get; }

    public MessageRenderer()
    {
        SubRenderers = new List<IRenderer>
        {
            new HeaderRenderer(),
            new BodyRenderer(),
            new FooterRenderer()
        };
    }

    public string Render(Message message)
    {
        return SubRenderers
            .Select(x => x.Render(message))
            .Aggregate("", (str1, str2) => str1 + str2);
    }
}
```

Класс `MessageRenderer` содержит несколько подгенераторов, которым он поручает работу над частями сообщения. Затем результат объединяется в документ HTML. Подгенераторы дополняют текст тегами HTML. Например:

```
public class BodyRenderer : IRenderer
{
    public string Render(Message message)
    {
        return $"<b>{message.Body}</b>";
    }
}
```

Как протестировать `MessageRenderer`? Первый вариант здесь — проанализировать алгоритм класса.

Листинг 4.2. Проверка правильности структуры `MessageRenderer`

```
[Fact]
public void MessageRenderer_uses_correct_sub_renderers()
{
    var sut = new MessageRenderer();

    IReadOnlyList<IRenderer> renderers = sut.SubRenderers;

    Assert.Equal(3, renderers.Count);
    Assert.IsAssignableFrom<HeaderRenderer>(renderers[0]);
    Assert.IsAssignableFrom<BodyRenderer>(renderers[1]);
    Assert.IsAssignableFrom<FooterRenderer>(renderers[2]);
}
```

Тест проверяет, что все подгенераторы имеют ожидаемый тип и присутствуют в правильном порядке. Предполагается, что если это так, то и результат работы `MessageRenderer` тоже должен быть правильным. Тест на первый взгляд выглядит нормально, но действительно ли он проверяет наблюдаемое поведение класса `MessageRenderer`? Что будет, если мы переставим подгенераторы местами или заменим один из них другим? Приведет ли это к ошибке?

Необязательно. Композицию подгенераторов можно изменить таким образом, что полученный HTML-документ останется неизменным. Например, можно заменить `BodyRenderer` подгенератором `BoldRenderer`, который делает то же, что и `BodyRenderer`. Или вы можете вообще избавиться от всех подгенераторов и реализовать генерирование разметки прямо в `MessageRenderer`.

Тем не менее при любых попытках такого рода тест будет падать, даже если конечный результат не изменится. Это связано с тем, что тест привязывается к деталям имплементации тестируемой системы, а не к генерируемому ею результату. Этот тест анализирует алгоритм и ожидает увидеть одну конкретную имплементацию, не учитывая равноправных альтернативных имплементаций (рис. 4.1).

Любой значительный рефакторинг класса `MessageRenderer` приведет к падению теста. Рефакторинг — это изменение деталей имплементации без изменения наблюдаемого поведения. Падение происходит именно из-за того, что тест завязывается на эти детали имплементации.

Отсюда следует, что тесты, завязанные на детали имплементации тестируемой системы, не устойчивы к рефакторингу. Такие тесты обладают всеми недостатками, которые я описывал ранее.

Они не предоставляют раннего предупреждения в случае ошибок — вы просто проигнорируете эти предупреждения из-за их неактуальности.

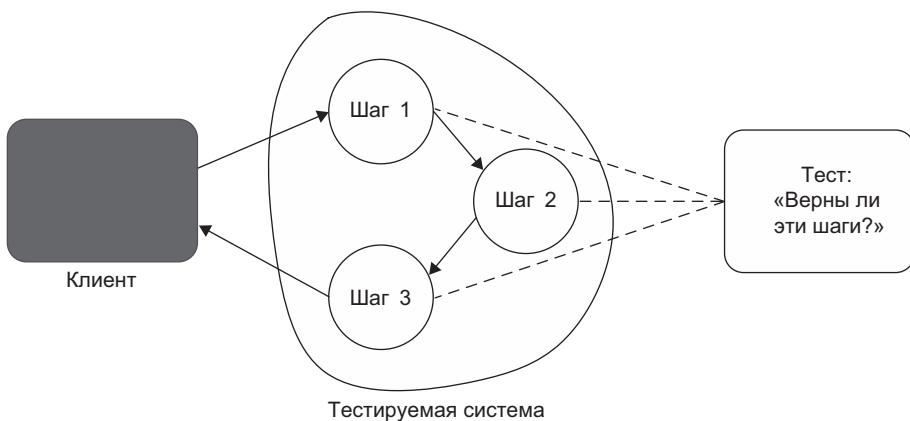


Рис. 4.1. Тест, привязанный к алгоритму тестируемой системы. Такой тест рассчитан на одну конкретную имплементацию (конкретные шаги, которые должна выполнить тестируемая система для выдачи результата), а следовательно, является хрупким. Любой рефакторинг тестируемой системы приводит к падению теста

Они уменьшают желание проводить рефакторинг. И неудивительно — кто хотел бы проводить рефакторинг, зная, что тесты не помогут выявить потенциальные ошибки?

В листинге 4.3 показан пример хрупкости тестов, доведенный до крайности. В этом примере тест читает исходный код класса `MessageRenderer` и сравнивает его с «правильной» реализацией.

Листинг 4.3. Проверка исходного кода класса `MessageRenderer`

```
[Fact]
public void MessageRenderer_is_implemented_correctly()
{
    string sourceCode = File.ReadAllText(@"[path]\MessageRenderer.cs");

    Assert.Equal(@""
public class MessageRenderer : IRenderer
{
    public IReadOnlyList<<IRenderer>> SubRenderers { get; }
    public MessageRenderer()
    {
        SubRenderers = new List<<IRenderer>>
        {
            new HeaderRenderer(),
            new BodyRenderer(),
            new FooterRenderer()
        };
    }
    public string Render(Message message) { /* ... */ }
}", sourceCode);
}
```

Конечно, этот тест никуда не годится: он будет падать при изменении даже мельчайших деталей класса `MessageRenderer`. В то же время он не так сильно отличается от теста, приведенного выше. Обе версии настаивают на конкретной имплементации, не принимая во внимание наблюдаемое поведение тестируемой системы. И оба теста будут падать каждый раз, когда вы изменяете эту имплементацию. Впрочем, нужно признать, что тест из листинга 4.3 будет падать чаще теста из листинга 4.2.

4.1.4. Тестирование конечного результата вместо деталей имплементации

Как упоминалось ранее, избежать хрупкости в тестах и повысить их устойчивость к рефакторингу можно только одним способом — отвязав их от деталей имплементации тестируемой системы. Тесты должны находиться как можно дальше от внутренних механизмов кода и проверять только конечный результат. Давайте отрефакторим тест из листинга 4.2, чтобы уменьшить его хрупкость.

Для начала необходимо понять, что является конечным результатом работы `MessageRenderer`. Это представление сообщения в формате HTML. И это единственное, что имеет смысл проверять в тестах. До тех пор пока HTML не меняется, нет смысла беспокоиться о том, как именно он был сгенерирован, — такие подробности имплементации не важны. В листинге 4.4 приведена новая версия теста.

Листинг 4.4. Проверка результата `MessageRenderer`

```
[Fact]
public void Rendering_a_message()
{
    var sut = new MessageRenderer();
    var message = new Message
    {
        Header = "h",
        Body = "b",
        Footer = "f"
    };
    string html = sut.Render(message);
    Assert.Equal("<h1>h</h1><b>b</b><i>f</i>", html);
}
```

Этот тест рассматривает `MessageRenderer` как «черный ящик» и интересуется только его наблюдаемым поведением. В результате тест становится намного более устойчивым к рефакторингу — его не интересует, какие изменения вносятся в тестируемую систему, при условии что конечный HTML остается неизменным (рис. 4.2).

Обратите внимание на принципиальное улучшение этого теста по сравнению с исходной версией. Он соотносится с требованиями бизнеса, проверяя только

результат, который имеет смысл для конечного пользователя, — то, как сообщение отображается в браузере. Падения таких тестов всегда указывают на проблему: они сообщают об изменении поведения приложения, что может отразиться на пользователе, и поэтому должны быть рассмотрены разработчиком. Такой тест выдает минимум ложных срабатываний.

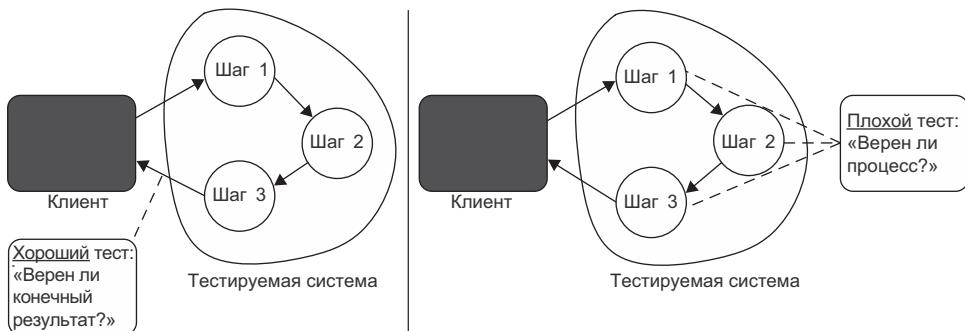


Рис. 4.2. Тест слева связан с наблюдаемым поведением SUT, а не с деталями реализации. Такой тест устойчив к рефакторингу — он вызовет несколько ложных срабатываний, если таковые будут

Почему «минимум», а не «нуль»? Потому что в `MessageRenderer` все еще могут быть внесены изменения, которые нарушают работу теста. Например, можно добавить дополнительный параметр в метод `Render()`, и это приведет к ошибке компиляции. С технической точки зрения такая ошибка тоже будет считаться ложным срабатыванием, ведь падение теста произошло не из-за изменения в поведении приложения.

Впрочем, такие ложные срабатывания легко исправить. Просто следуйте рекомендациям компилятора и добавьте новый параметр во все тесты, вызывающие метод `Render()`. Хуже обстоит дело с ложными срабатываниями, которые не приводят к ошибкам компиляции. С такими ложными срабатываниями справиться сложнее всего — они выглядят так, словно указывают на настоящую ошибку, и на разбирательства с ними уходит намного больше времени.

4.2. Связь между первыми двумя атрибутами

Как упоминалось ранее, между первыми двумя аспектами хорошего юнит-теста (защита от багов и устойчивость к рефакторингу) существует связь. Оба атрибута вносят вклад в точность тестов, хотя и с противоположных позиций. Эти два атрибута также по-разному влияют на проект с течением времени: важно иметь хорошую защиту от багов сразу же после запуска проекта, но необходимость в устойчивости к рефакторингу возникает позднее.

В этом разделе рассматриваются следующие темы:

- максимизация точности тестов;
- важность ложных и ложноотрицательных срабатываний.

4.2.1. Максимизация точности тестов

Давайте рассмотрим более широкую картину того, что собой представляют результаты тестовых прогонов. В том, что касается правильности кода и результатов тестирования, возможны четыре варианта, представленных на рис. 4.3. Тесты могут проходить или не проходить (строки таблицы), а сама функциональность может работать либо правильно, либо неправильно (столбцы таблицы).

Ситуация, когда тест проходит, а тестируемая функциональность работает правильно, называется истинным отрицательным срабатыванием: тест правильно определяет состояние системы (отсутствие в ней ошибок).

Если тест не выявляет ошибку, значит, возникла проблема. Ситуация соответствует правому верхнему квадранту: *ложноотрицательное срабатывание*. И именно ее помогает избежать первый атрибут хорошего теста — защита от багов. Тесты с хорошей защитой от багов помогают минимизировать количество ложноотрицательных срабатываний — ошибок II типа.

Типы ошибок		Функциональность		Защита от багов
		Работает правильно	Работает неправильно	
Тест	...проходит	Истинное отрицательное срабатывание	Ошибка II рода (ложноотрицательное срабатывание)	Защита от багов
	...не проходит	Ошибка I рода (ложное срабатывание)	Истинное срабатывание	

Устойчивость к рефакторингу

Рис. 4.3. Отношение между защитой от багов и устойчивостью к рефакторингу. Защита от багов предохраняет от ложноотрицательных срабатываний (ошибки II типа). Устойчивость к рефакторингу минимизирует количество ложных срабатываний (ошибки I типа)

С другой стороны, существует симметричная ситуация: функциональность работает правильно, но тест сообщает об ошибке. Это *ложное срабатывание*, то есть «ложная тревога». И с ней помогает второй атрибут — устойчивость к рефакторингу.

Количества ложных и ложноотрицательных срабатываний образуют метрику точности теста: чем меньше таких срабатываний, тем точнее тест. Первые два аспекта хорошего юнит-теста относятся именно к точности. Защита от багов и устойчивость к рефакторингу направлены на максимизацию точности тестов. Сама метрика точности состоит из двух компонентов:

- насколько хорошо тест выявляет *присутствие* ошибок (отсутствие ложноотрицательных срабатываний, сфера защиты от багов);
- насколько хорошо тест выявляет *отсутствие* ошибок (отсутствие ложных срабатываний, сфера устойчивости к рефакторингу).

Ложные и ложноотрицательные срабатывания можно также рассматривать в контексте отношения «сигнал/шум». Как видно из формулы на рис. 4.4, улучшить точность теста можно двумя способами. Первый — повышение числителя (*сигнал*), то есть повышение вероятности выявления ошибок. Второй — уменьшение знаменателя (*шум*), то есть уменьшения вероятности ложных срабатываний.

$$\text{Точность теста} = \frac{\text{Сигнал (количество обнаруженных ошибок)}}{\text{Шум (количество ложных срабатываний)}}$$

Рис. 4.4. Тест точен, если он выдает хороший сигнал (способен находить ошибки) с минимально возможным шумом (не выдает ложных срабатываний)

Оба параметра очень важны. Тест, который не может находить ошибки, бесполезен, даже если он не выдает ложных срабатываний. Аналогичным образом точность теста, который выдает большое количество шума, падает до нуля, даже если этот тест может найти все ошибки в коде. Найденные таким тестом ошибки попросту теряются в море нерелевантной информации.

4.2.2. Важность ложных и ложноотрицательных срабатываний: динамика

В краткосрочной перспективе ложные срабатывания не так важны, как ложноотрицательные срабатывания. В начале проекта получить неверное предупреждение об ошибке не так страшно, как не получить предупреждения вообще. Но с ростом проекта ложные срабатывания начинают все сильнее влиять на проект (рис. 4.5).

Почему ложные срабатывания менее важны на начальной стадии? Потому что важность рефакторинга тоже не проявляется немедленно; она возрастает постепенно, со временем. В начале проекта не нужно проводить многочисленные чистки кода. Только что написанный код часто безупречен. К тому же этот код еще не успел выветриться из вашей памяти, так что вы можете легко провести его рефакторинг, даже если тесты выдают ложные срабатывания.

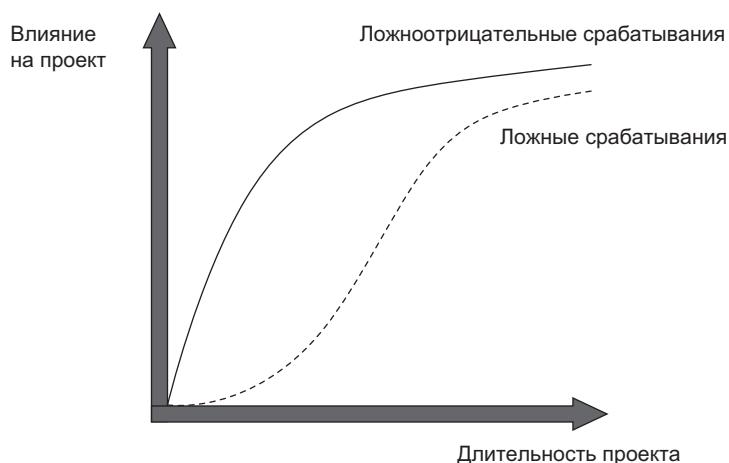


Рис. 4.5. Ложные срабатывания (ложные сигналы тревоги) не оказывают заметного отрицательного влияния в начале проекта. С ростом проекта они начинают играть все более важную роль — столь же важную, как и ложноотрицательные срабатывания (незамеченные ошибки)

Но с течением времени качество кода постепенно ухудшается. Он становится все более сложным и дезорганизованным. Необходимо проводить регулярный рефакторинг, чтобы не допустить дальнейшего ухудшения качества кода. В противном случае затраты на разработку новой функциональности в таком коде возрастают слишком сильно.

С ростом необходимости в рефакторинге растет и важность устойчивости тестов к рефакторингу. Как объяснялось ранее, рефакторинг невозможен, если тесты постоянно поднимают ложную тревогу, и вы получаете предупреждения о несуществующих ошибках. Доверие к тестам быстро теряется, и они перестают рассматриваться как надежный источник обратной связи.

Несмотря на важность защиты кода от ложных срабатываний, особенно на более поздних стадиях проекта, лишь немногие разработчики понимают эту важность. Большинство людей обычно концентрируется на улучшении первого атрибута хороших юнит-тестов — защите от багов, чего недостаточно для построения эффективных тестов.

Такое отношение объясняется тем, что лишь небольшая часть проектов добирается до более поздних стадий — чаще всего из-за того, что сами эти проекты невелики и разработка завершается до того, как они становятся слишком большими. Разработчики сталкиваются с проблемой незамеченных ошибок намного чаще, чем с ложными срабатываниями, которые препятствуют рефакторингу. Тем не менее, если вы работаете над средним или большим проектом, вам необходимо уделять равное внимание как ложноотрицательным срабатываниям (пропущенным ошибкам), так и ложным срабатываниям (ложным сигналам тревоги).

4.3. Третий и четвертый аспекты: быстрая обратная связь и простота поддержки

В этом разделе речь пойдет о двух оставшихся аспектах хорошего юнит-теста:

- быстрая обратная связь;
- простота поддержки.

Как говорилось в главе 2, быстрая обратная связь является одним из важнейших свойств юнит-теста. Чем быстрее работают тесты, тем больше их можно включить в проект и тем чаще вы их сможете запускать.

Быстро выполняемые тесты сильно ускоряют обратную связь. В идеальном случае тесты начинают предупреждать вас об ошибках сразу же после их внесения, в результате чего затраты на исправление этих ошибок уменьшаются почти до нуля. С другой стороны, медленные тесты увеличивают время, в течение которого ошибки остаются необнаруженными, что приводит к увеличению затрат на их исправление. Дело в том, что медленные тесты отбивают у разработчика желание часто запускать их, поэтому в итоге он тратит больше времени, двигаясь в ошибочном направлении.

Наконец, четвертый аспект хороших юнит-тестов — *простота поддержки* — оценивает затраты на сопровождение кода. Метрика состоит из двух компонентов:

- *Насколько сложно тест понять.* Этот компонент связан с размером теста. Чем меньше кода в тесте, тем проще он читается. Также небольшие тесты проще изменить при необходимости. Качество кода тестов не менее важно, чем качество рабочего кода. Не пренебрегайте качеством кода тестов; относитесь к коду тестов как к полноценному коду.
- *Насколько сложно тест запустить.* Если тест работает с внепроцессными зависимостями, вам придется тратить время на то, чтобы поддерживать эти зависимости в рабочем состоянии: перезагружать сервер базы данных, решать проблемы с сетью и т. д.

4.4. В поисках идеального теста

Еще раз перечислю четыре атрибута хороших юнит-тестов:

- защита от багов;
- устойчивость к рефакторингу;
- быстрая обратная связь;
- простота поддержки.

Произведение этих четырех атрибутов определяет эффективность теста. И в данном случае я использую термин «произведение» в математическом смысле: если один из атрибутов равен нулю, то ценность всего теста тоже обращается в нуль:

Эффективность теста = $[0..1] * [0..1] * [0..1] * [0..1]$

СОВЕТ

Чтобы тест был эффективным, он должен демонстрировать результативность в каждой из четырех категорий.

Конечно, точно измерить эти атрибуты невозможно. Не существует утилиты, которой можно передать на вход тест и получить на выходе значение эффективности. Тем не менее тест можно достаточно точно оценить и узнать его эффективность по каждому из четырех атрибутов. В свою очередь, эта оценка дает сводную оценку полезности теста, по которой можно решить, стоит ли оставлять тест в проекте.

Напомню, что весь код, включая код тестов, является обязательством, а не активом. Установите достаточно высокий порог для минимальной требуемой эффективности и включайте в проект только те тесты, которые проходят этот порог. Небольшой набор высокоэффективных тестов намного лучше справится с задачей поддержания роста проекта, чем большое количество посредственных тестов.

Вскоре я приведу примеры. А пока попробуем понять, возможно ли создать идеальный тест.

4.4.1. Возможно ли создать идеальный тест?

Идеальный тест получает максимальные оценки по всем четырем атрибутам. Если считать, что минимальное и максимальное значение каждого атрибута равны 0 и 1, идеальный тест должен получить 1 по всем четырем атрибутам.

К сожалению, создать такой тест невозможно. Дело в том, что первые три атрибута — защита от багов, устойчивость к рефакторингу и быстрая обратная связь — являются взаимоисключающими. Невозможно довести их до максимума одновременно: одним из трех придется пожертвовать для максимизации двух остальных.

Более того, из-за принципа умножения (см. формулу эффективности выше) выдержать баланс еще сложнее. Нельзя просто обнулить один атрибут, чтобы сосредоточиться на остальных. Как упоминалось ранее, тест с нулевым значением в одной из четырех категорий бесполезен. Следовательно, атрибуты нужно максимизировать так, чтобы ни один из них не падал слишком низко.

Рассмотрим примеры тестов, которые стараются максимизировать два из трех атрибутов за счет третьего. Как упоминалось раньше, эффективность таких тестов близка к нулю.

4.4.2. Крайний случай № 1: сквозные (end-to-end) тесты

Первый пример — сквозные (end-to-end) тесты. Как вы, возможно, помните из главы 2, сквозные тесты рассматривают систему с точки зрения конечного пользователя. Они обычно проходят через все компоненты системы, включая пользовательский интерфейс, базу данных и внешние приложения.

Так как сквозные тесты задействуют большой объем кода, они обеспечивают наилучшую защиту от багов. Из всех возможных типов тестов именно сквозные тесты задействуют наибольший объем кода — как ваш код, так и код, написанный не вами, но используемый в проекте: внешние библиотеки, фреймворки и сторонние приложения.

Сквозные тесты практически не выдают ложных срабатываний, а следовательно, обладают хорошей устойчивостью к рефакторингу. Правильно проведенный рефакторинг не изменяет наблюдаемого поведения системы, а следовательно, не влияет на сквозные тесты. Это другое преимущество таких тестов: они не настаивают на какой-то конкретной имплементации. Единственное, на что смотрят сквозные тесты, — поведение приложения с точки зрения конечного пользователя. Они настолько отделены от деталей имплементации, насколько это возможно.

Тем не менее наряду с преимуществами у сквозных тестов имеется крупный недостаток: они очень медленные. Любой проект, который полагается исключительно на такие тесты, не сможет получить быструю обратную связь. Именно поэтому невозможно обеспечить покрытие кода только сквозными тестами.

На рис. 4.6 показано, какое место занимают сквозные тесты по отношению к первым трем метрикам юнит-тестирования. Такие тесты предоставляют отличную защиту от багов и ложных срабатываний, но им не хватает скорости.



Рис. 4.6. Сквозные тесты обеспечивают превосходную защиту как от багов, так и от ложных срабатываний, но плохо проявляют себя в быстроте обратной связи

4.4.3. Крайний случай № 2: тривиальные тесты

Другой пример максимизации двух из трех атрибутов за счет третьего — тривиальный тест. Такие тесты покрывают простой фрагмент кода, вероятность сбоя в котором невелика. Это показано в листинге 4.5.

Листинг 4.5. Тривиальный тест, покрывающий простой фрагмент кода

```
public class User
{
    public string Name { get; set; } ← Вероятность ошибки
}                                         в подобных односторонних
[Fact]                                     методах низка
public void Test()
{
    var sut = new User();

    sut.Name = "John Smith";

    Assert.Equal("John Smith", sut.Name);
}
```

В отличие от сквозных тестов, тривиальные тесты предоставляют быструю обратную связь. Кроме того, вероятность ложных срабатываний также мала, поэтому они обладают хорошей устойчивостью к рефакторингу. Тем не менее тривиальные тесты вряд ли смогут выявить какие-либо ошибки, потому что покрываемый ими код слишком прост.

Тривиальные тесты, доведенные до крайности, ведут к появлению *тавтологических* тестов. Такие тесты ничего не тестируют — они построены так, что всегда проходят или содержат бессмысленные проверки.

На рис. 4.7 показано, какое место занимают тривиальные тесты. Они обладают хорошей устойчивостью к рефакторингу и предоставляют быструю обратную связь, но не защищают от багов.

4.4.4. Крайний случай № 3: хрупкие тесты

Также достаточно легко написать тест, который работает быстро и хорошо выявляет ошибки в коде, но делает это с большим количеством ложных срабатываний. Такие тесты называются *хрупкими*: они падают при любом рефакторинге тестируемого кода независимо от того, изменилась тестируемая ими функциональность или нет.

Пример хрупкого теста уже встречался в листинге 4.2. В листинге 4.6 представлен другой пример.



Рис. 4.7. Тривиальные тесты обладают хорошей устойчивостью к рефакторингу и обеспечивают быструю обратную связь, но не защищают от багов

Листинг 4.6. Тест, проверяющий, какая команда SQL была выполнена

```
public class UserRepository
{
    public User GetById(int id)
    {
        /* ... */
    }

    public string LastExecutedSqlStatement { get; set; }
}

[Fact]
public void GetById_executes_correct_SQL_code()
{
    var sut = new UserRepository();

    User user = sut.GetById(5);

    Assert.Equal(
        "SELECT * FROM dbo.[User] WHERE UserID = 5",
        sut.LastExecutedSqlStatement);
}
```

Этот тест проверяет, генерирует ли класс `UserRepository` правильную команду SQL при выборке пользователя от базы данных. Может ли этот тест обнаружить ошибку? Может. Например, разработчик может ошибиться в SQL-коде и использовать `ID` вместо `UserID`; тест упадет, сообщив об этой ошибке. Но обладает ли этот тест хорошей устойчивостью к рефакторингу? Нет. Существует несколько разновидностей команды SQL, которые приводят к одному и тому же результату:

```
SELECT * FROM dbo.[User] WHERE UserID = 5
SELECT * FROM dbo.User WHERE UserID = 5
SELECT UserID, Name, Email FROM dbo.[User] WHERE UserID = 5
SELECT * FROM dbo.[User] WHERE UserID = @UserID
```

Тест в листинге 4.6 упадет при замене SQL-кода любой из этих разновидностей, хотя сама функциональность остается работоспособной. Это еще один пример привязывания теста к деталям имплементации тестируемой системы. Тест фокусируется на том, *как* работает приложение, что препятствует дальнейшему рефакторингу. Вместо этого тест должен проверять, *что* делает это приложение.

На рис. 4.8 показано, что хрупкие тесты попадают в третью категорию. Такие тесты выполняются быстро и обеспечивают хорошую защиту от багов, но обладают недостаточной устойчивостью к рефакторингу.

4.4.5. В поисках идеального теста: результаты

Первые три атрибута хорошего юнит-теста (защита от багов, устойчивость к рефакторингу и быстрая обратная связь) являются взаимоисключающими. Создать тест, который максимизирует два из этих трех атрибутов, несложно, но это можно сделать только за счет третьего атрибута. Тем не менее эффективность такого теста будет практически нулевой из-за правила умножения. К сожалению, создать идеальный тест с максимальными значениями всех трех атрибутов невозможно (рис. 4.9).



Рис. 4.8. Хрупкие тесты быстро выполняются и обеспечивают хорошую защиту от багов, но обладают недостаточной устойчивостью к рефакторингу

Четвертый атрибут — *простота поддержки* — не так сильно связан с первыми тремя, за исключением сквозных (end-to-end) тестов. Сквозные тесты обычно имеют

больший размер из-за необходимости подготовки всех зависимостей, к которым могут обращаться такие тесты. Они также требуют дополнительных усилий для поддержания этих зависимостей в работоспособном состоянии. Таким образом, сквозные тесты требуют больших затрат на сопровождение.



Рис. 4.9. Невозможно создать идеальный тест с идеальными значениями всех трех атрибутов

Выдержать баланс между атрибутами хорошего теста сложно. Тест не может иметь максимальных значений в каждой из первых трех категорий; также приходится учитывать аспект простоты поддержки. А значит, вам придется идти на компромиссы. Более того, на компромиссы придется идти так, чтобы ни один конкретный атрибут не оказался равным нулю. Уступки должны быть частичными и стратегическими.

Как должны выглядеть эти уступки? Из-за взаимоисключающего характера защиты от багов, устойчивости к рефакторингу и быстрой обратной связи можно подумать, что оптимальной стратегией будет незначительное снижение каждого атрибута — ровно настолько, чтобы хватило места для всех трех атрибутов.

На практике же устойчивость к рефакторингу не должна быть предметом для уступок. Вы должны постараться довести ее до максимума, при условии что ваши тесты остаются достаточно быстрыми и вы не переходите на использование исключительно сквозных тестов. Таким образом, компромисс сводится к выбору соотношения между тем, насколько хорошо ваши тесты будут справляться с поиском ошибок, и насколько быстро они будут это делать. Другими словами, между защитой от багов и быстрой обратной связью. Представьте себе это как ползунок, который может свободно перемещаться по шкале между защитой от багов и быстрой обратной связью. Чем больше усиливается один атрибут, тем больше вы теряете в другом (рис. 4.10).

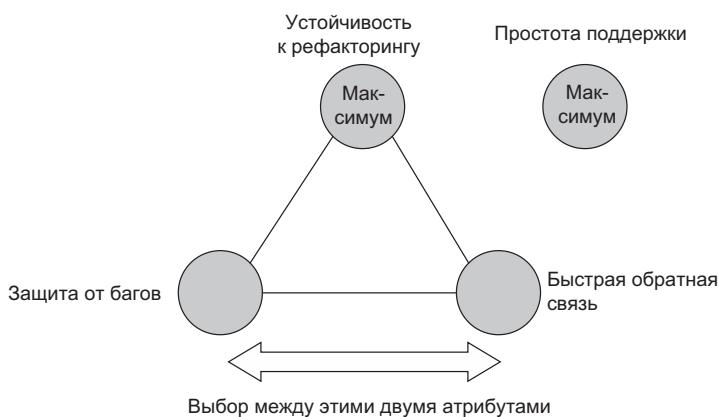


Рис. 4.10. Лучшие тесты демонстрируют максимально возможную простоту поддержки и устойчивость к рефакторингу; всегда старайтесь максимизировать эти два атрибута. Компромисс сводится к выбору между защитой от багов и быстрой обратной связи

TEOPEMA CAP

Компромисс между первыми тремя атрибутами хорошего юнит-теста напоминает теорему CAP. Эта теорема утверждает, что распределенное хранилище данных не может предоставить более двух из трех следующих гарантий одновременно:

- ✓ согласованность (*consistency*) данных: при каждой операции чтения будут получены данные от последней операции записи или ошибка;
 - ✓ доступность (*availability*): для каждого запроса будет получен ответ — кроме сбоев, распространяющихся на все узлы системы;
 - ✓ устойчивость к разделению (*partition tolerance*): система сохраняет работоспособность даже при расщеплении сети, то есть потере связи между ее узлами.

Сходство является двойным:

- ✓ в CAP вы тоже можете выбрать максимум два атрибута из трех;
 - ✓ устойчивость к разделению в крупномасштабных распределенных системах также не является предметом для компромиссов. Большое приложение — такое как, например, веб-сайт Amazon — не может работать на одной машине. Вариант с достижением согласованности данных и доступности за счет устойчивости к разделению просто не рассматривается — объем данных Amazon слишком велик для хранения на одном сервере, каким бы мощным ни был этот сервер.

А значит, этот выбор тоже сводится к компромиссу между согласованностью и доступностью. В некоторых частях системы предпочтительно пожертвовать небольшой частью согласованности для повышения доступности. Например, при выводе каталога продуктов обычно нормально, если какие-то части каталога содержат устаревшие данные. Доступность в таком сценарии имеет более высокий приоритет. С другой стороны, при обновлении описания продуктов согласованность данных важнее доступности: узлы сети должны иметь консенсус относительно того, какая версия этого описания является самой актуальной, для предотвращения конфликтов при одновременном редактировании.

Почему же устойчивость к рефакторингу не должна быть предметом для компромиссов? Потому что этот атрибут в основном сводится к бинарному выбору: тест либо устойчив к рефакторингу, либо нет. Между этими двумя состояниями почти нет промежуточных ступеней. А значит, пожертвовать небольшой частью устойчивости к рефакторингу не получится; придется потерять ее *полностью*. С другой стороны, метрики защиты от багов и быстрой обратной связи более эластичны. В следующем разделе будет показано, какие компромиссы возможны при выборе одного атрибута в ущерб другому.

СОВЕТ

Борьба с хрупкостью тестов (с ложными срабатываниями) становится первоочередной задачей на пути к построению эффективных тестов.

4.5. Известные концепции автоматизации тестирования

Четыре атрибута хороших юнит-тестов, представленные выше, являются фундаментальными. Все существующие и хорошо известные концепции автоматизации могут быть сведены к этим четырем атрибутам. В этом разделе будут рассмотрены две такие концепции: пирамида тестирования и тестирование по принципу «черного ящика»/«белого ящика».

4.5.1. Пирамида тестирования

Концепция пирамиды тестирования предписывает определенное соотношение разных типов тестов в проекте (рис. 4.11):



Рис. 4.11. Пирамида тестирования предписывает определенное соотношение юнит-, интеграционных и сквозных тестов

- юнит-тесты;
- интеграционные тесты;
- сквозные тесты.

Пирамида тестирования часто изображается состоящей из трех типов тестов. Ширина уровней пирамиды обозначает относительную долю тестов определенного типа в проекте. Чем шире уровень, тем больше тестов. Высота уровня показывает, насколько близки эти тесты к эмуляции поведения конечного пользователя. Сквозные тесты расположены на вершине — они ближе всего имитируют конечного пользователя. Разные типы тестов в пирамиде выбирают разные компромиссы между быстротой обратной связи и защитой от багов. Тесты более высоких уровней пирамиды отдают предпочтение защите от багов, тогда как тесты нижних уровней выводят на первый план скорость выполнения (рис. 4.12).

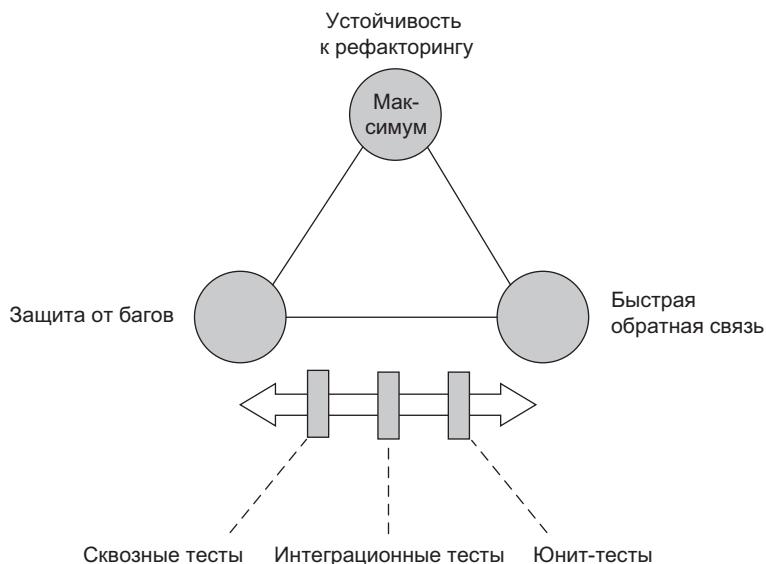


Рис. 4.12. Разные типы тестов в пирамиде принимают разные решения относительно быстрой обратной связи и защиты от багов. Сквозные тесты отдают предпочтение защите от багов, юнит-тесты — быстроте обратной связи. Интеграционные тесты находятся посередине

Обратите внимание: ни один из уровней не делает устойчивость к рефакторингу предметом для компромисса. Конечно, сквозные и интеграционные тесты более устойчивы к рефакторингу, чем юнит-тесты, но только как побочный эффект от того, что они не работают так тесно с рабочим кодом, как юнит-тесты. Тем не менее даже юнит-тесты не должны делать уступок в отношении устойчивости к рефакторингу. Все они должны выдавать как можно меньше ложных срабатываний, независимо

от того, насколько тесно они работают с рабочим кодом. (О том, как этого добиться, рассказано в следующей главе.)

Точное соотношение между типами тестов будет разным для разных команд и проектов. Но в общем случае должно сохраняться соотношение пирамиды: сквозные тесты составляют меньшинство; юнит-тесты — большинство; интеграционные тесты лежат где-то в середине.

Причина, по которой сквозных тестов меньше всего, кроется в правиле умножения из раздела 4.4. Для сквозных тестов характерна исключительно низкая скорость выполнения. Они также не отличаются простотой в поддержке: такие тесты обычно занимают много места и требуют дополнительных усилий для поддержания задействованных внепроцессных зависимостей. Таким образом, сквозные тесты имеет смысл применять только к самой критической функциональности — функциональности, в которых нежелательны любые ошибки — и только когда вы не можете добиться аналогичного уровня защиты с юнит- и интеграционными тестами. Не используйте сквозные тесты в других случаях. Юнит-тесты более сбалансированы, поэтому их обычно больше.

У пирамиды тестирования есть исключения. Например, если ваше приложение содержит только базовые операции создания, чтения, обновления и удаления (CRUD) с минимумом бизнес-правил, ваша пирамида тестирования будет больше напоминать прямоугольник с равным количеством юнит- и интеграционных тестов, без сквозных тестов.

Юнит-тесты менее полезны в ситуациях, в которых отсутствует алгоритмическая или бизнес-сложность, — они быстро вырождаются в тривиальные тесты. В то же время интеграционные тесты полезны даже в таких случаях; каким бы простым код ни был, важно проверить, как он работает в интеграции с другими подсистемами (например, базой данных). В результате в CRUD-приложениях у вас будет меньше юнит-тестов и больше интеграционных. В самых тривиальных случаях интеграционных тестов может быть даже больше, чем юнит-тестов.

Другое исключение из пирамиды тестирования — API, обращающиеся к единственной внепроцессной зависимости (например, базе данных). В таких приложениях логично задействовать больше сквозных тестов. Так как пользовательский интерфейс отсутствует, сквозные тесты будут выполняться достаточно быстро. Затраты на сопровождение тоже будут не особенно велики, потому что вы работаете только с одной внешней зависимостью — базой данных. В такой среде сквозные тесты по сути неотличимы от интеграционных. Отличается только точка входа: сквозные тесты требуют, чтобы приложение было где-то размещено для полной эмуляции конечного пользователя, тогда как интеграционные тесты обычно запускают приложение в том же процессе. Мы вернемся к пирамиде тестирования в главе 8, когда речь пойдет об интеграционном тестировании.

4.5.2. Выбор между тестированием по принципу «черного ящика» и «белого ящика»

Также хорошо известна концепция автоматизации тестирования по принципу «черного ящика» и «белого ящика». В этом разделе я объясню, когда следует использовать каждый из двух методов.

- Тестирование по принципу «черного ящика» проверяет функциональность системы без знания ее внутренней структуры. Такое тестирование обычно строится на основе спецификаций и требований. Оно проверяет, что должно делать приложение, а не то, как оно это делает.
- Тестирование по принципу «белого ящика» работает по противоположному принципу. Этот метод тестирования проверяет внутренние механизмы приложения. Тесты строятся на основе исходного кода, а не на основе требований или спецификаций.

У обоих методов есть как достоинства, так и недостатки. Тестирование по принципу «белого ящика» обычно получается более тщательным. Анализируя исходный код, можно выявить множество ошибок, которые часто упускаются, когда вы полагаетесь исключительно на внешние спецификации. С другой стороны, результаты тестирования по принципу «белого ящика» часто оказываются хрупкими, поскольку они часто завязаны на детали имплементации тестируемого кода. Такие тесты генерируют много ложных срабатываний и поэтому не имеют хорошей устойчивости к рефакторингу. Также их редко получается связать с важным для бизнеса поведением — верный признак того, что тесты являются хрупкими и не добавляют особой ценности. Тестирование по принципу «черного ящика» обладает противоположными достоинствами и недостатками (табл. 4.1).

Таблица 4.1. Достоинства и недостатки тестирования по принципу «черного ящика» и «белого ящика»

	Защита от багов	Устойчивость к рефакторингу
Тестирование по принципу «белого ящика»	Хорошая	Плохая
Тестирование по принципу «черного ящика»	Плохая	Хорошая

Как вы помните из раздела 4.4.5, нельзя делать уступки в отношении устойчивости тестов к рефакторингу: тест либо хрупок, либо нет. Всегда отдавайте предпочтение тестированию по принципу «черного ящика». Тесты — неважно, юнит-, интеграционные или сквозные — должны рассматривать систему как «черный ящик» и проверять поведение, имеющее смысл с точки зрения бизнес-логики. Если тест не удается связать с бизнес-требованием, это является признаком хрупкости теста. Отрефакторите или удалите этот тест; не включайте его в проект в исходном виде.

Единственное исключение составляют тесты, содержащие служебный код с высокой алгоритмической сложностью (подробнее об этом в главе 7).

Хотя тестирование по принципу «черного ящика» предпочтительнее тестирования по принципу «белого ящика», метод «белого ящика» может применяться при *анализе* тестов. Используйте утилиты, показывающие покрытие кода для выявления непротестированных частей приложения, но затем тестируйте их так, словно вам ничего не известно о внутренней структуре этого кода. Такая комбинация методов «черного ящика» и «белого ящика» работает лучше всего.

Итоги

- Хороший юнит-тест должен обладать четырьмя фундаментальными атрибутами, которые могут использоваться для анализа любых автоматизированных тестов (юнит-, интеграционных или сквозных):
 - защита от багов;
 - устойчивость к рефакторингу;
 - быстрая обратная связь;
 - простота поддержки.
- Защита от багов показывает, несколько хорошо тест справляется с выявлением ошибок (ретрессий). Чем больше кода проверяет тест (как вашего, так и кода библиотек и фреймворков, задействованных в проекте), тем выше вероятность того, что тест обнаружит ошибку.
- Устойчивость к рефакторингу определяет, насколько тест хрупок: может ли он перенести рефакторинг рабочего кода, не выдавая ложных срабатываний.
- Ложное срабатывание представляет собой «ложную тревогу»: тест падает, но покрываемая им функциональность работает. Ложные срабатывания негативно влияют на проект:
 - если тесты падают без веской причины, они притупляют вашу готовность реагировать на проблемы в коде. Со временем вы привыкаете к таким сбоям и перестаете обращать на них внимание. А это может привести к игнорированию настоящих ошибок, которые затем попадают в продуктив;
 - с другой стороны, при частых ложных срабатываниях вы начинаете все меньше и меньше доверять вашим тестам. Они уже не воспринимаются как что-то, на что вы можете положиться. Отсутствие доверия приводит к уменьшению рефакторинга, так как вы пытаетесь свести к минимуму потенциальные ошибки.
- Ложные срабатывания появляются в результате привязки тестов к деталям имплементации тестируемой системы. Чтобы избежать такой привязки, тест должен проверять конечный результат, а не действия, которые для этого потребовались.

- Защита от багов и устойчивость к рефакторингу составляют метрику точности теста. Тест точен, когда он выдает хороший сигнал (способен находить ошибки) с минимально возможным шумом (ложных срабатываний).
- Ложные срабатывания не оказывают заметного отрицательного влияния в начале проекта. С ростом проекта они начинают играть все более важную роль — столь же важную, как ложноотрицательные срабатывания (незамеченные ошибки).
- Быстрая обратная связь — мера того, насколько быстро выполняется тест.
- Простота поддержки состоит из двух компонентов:
 - сложность понимания теста. Чем меньше тест, тем проще он читается;
 - сложность выполнения теста. Чем меньше внешнепроцессных зависимостей, тем проще поддерживать их в работоспособном состоянии.
- Эффективность теста определяется произведением этих четырех атрибутов. Если один из атрибутов равен нулю, то эффективность всего теста тоже равна нулю.
- Невозможно создать тест, который имеет максимальные показатели по всем четырем атрибутам, потому что первые три — защита от багов, устойчивость к рефакторингу и быстрая обратная связь — являются взаимоисключающими. Тест может максимизировать только два атрибута из трех.
- Устойчивость к рефакторингу не должна быть предметом для компромиссов, потому что этот атрибут в основном сводится к бинарному выбору: тест либо устойчив к рефакторингу, либо нет. Компромисс между атрибутами сводится к выбору между защитой от багов и быстрой обратной связью.
- Пирамида тестирования предписывает определенное соотношение юнит-, интеграционных и сквозных тестов: сквозных тестов должно быть меньше всего, юнит-тестов — больше всего, а интеграционных тестов — где-то посередине.
- Разные типы тестов в пирамиде принимают разные компромиссы между быстрой обратной связью и защиты от багов. Сквозные тесты отдают предпочтение защите от багов, а юнит-тесты — быстрой обратной связью.
- Используйте тестирование по принципу «черного ящика» при написании тестов. Используйте тестирование по принципу «белого ящика» при анализе тестов.

5

Моки и хрупкость тестов

В этой главе:

- ✓ Отличия моков от стабов.
- ✓ Разница между наблюдаемым поведением и деталями имплементации.
- ✓ Понимание связи между моками и хрупкости тестов.
- ✓ Использование моков без вреда для устойчивости к рефакторингу.

В главе 4 была представлена система координат, которая может использоваться для анализа тестов и методов юнит-тестирования. В этой главе эта система координат будет показана в действии: мы воспользуемся ею для анализа темы моков.

Использование моков (*mocks*) в тестах — неоднозначная тема. Некоторые применяют их в большинстве своих тестов. Другие утверждают, что моки приводят к хрупким тестам, и стараются обходиться без них. Истина, как обычно, лежит где-то посередине. В этой главе я покажу, что моки действительно нередко становятся причиной хрупкости тестов — тестов с низкой устойчивостью к рефакторингу. Тем не менее существуют ситуации, в которых применение моков возможно и даже предпочтительно.

Для лучшего понимания этой главы вам необходимо ознакомиться с различиями между лондонской и классической школами юнит-тестирования из главы 2. Вкратце, различия между школами обусловлены разницей в их представлении об аспекте изоляции тестов. Лондонская школа предписывает изоляцию тестируемого кода (юнитов) и замену всех изменяемых зависимостей (коллaborаторов) на тестовые заглушки (например, моки) для поддержания этой изоляции.

Классическая школа предписывает изоляцию самих юнит-тестов, чтобы они могли выполнять независимо друг от друга. Эта школа использует тестовые заглушки только для зависимостей, переиспользуемых тестами.

Между моками и хрупкостью тестов существует глубокая и почти неизбежная связь. В этой главе я расскажу об этой связи. Вы также научитесь пользоваться моками без вреда для устойчивости теста к рефакторингу.

5.1. Отличия моков от стабов

В главе 2 я кратко упомянул о том, что *мок* (mock) — тестовая заглушка, которая позволяет проверять взаимодействия между тестируемой системой (SUT) и ее коллегами. Также существует другая разновидность тестовых заглушек — *стаб* (stub). Давайте рассмотрим, что такое мок и чем он отличается от стаба.

5.1.1. Разновидности тестовых заглушек

Тестовая заглушка (test double) — общий термин, который описывает все разновидности фиктивных зависимостей в тестах. Английский вариант термина происходит от *stunt double* — дублеров актеров на съемках. Тестовые заглушки используются прежде всего для упрощения тестирования; они передаются тестируемой системе вместо реальных зависимостей, настройка или сопровождение которых могут быть сопряжены с определенными сложностями.

Джерард Месарош (Gerard Meszaros) выделяет пять разновидностей тестовых заглушки: пустышки (dummy), стабы (stub), шпионы (spy), моки (mock) и фейки¹. Такое разнообразие может показаться устрашающим, но в действительности все разновидности можно разделить на два типа: моки и стабы (рис. 5.1).



Рис. 5.1. Все разновидности тестовых заглушки можно разделить на два типа: моки и стабы

Различия между двумя типами сводятся к следующему:

- Моки помогают эмулировать и проверять *выходные* взаимодействия — то есть вызовы, совершаемые тестируемой системой к ее зависимостям для изменения их состояния.

¹ См. *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley, 2007. (На русском языке: *Месарош Джерард. Шаблоны тестирования xUnit: рефакторинг кода тестов.* : Пер. с англ. — М. : ООО «И.Д. Вильямс», 2016. — 832 с.: ил. — Примеч. ред.)

- Стабы помогают эмулировать *входные* взаимодействия — то есть вызовы, совершаемые тестируемой системой к ее зависимостям для получения входных данных (рис. 5.2).

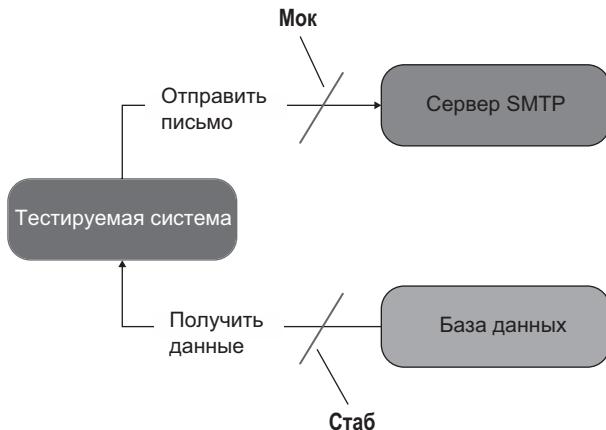


Рис. 5.2. Отправка письма является выходным взаимодействием — то есть взаимодействием, которое приводит к изменению состояния сервера SMTP. Тестовая заглушка, эмулирующая такое взаимодействие, является моком. Получение информации из базы данных является входным взаимодействием; оно не приводит к изменениям. Соответствующая тестовая заглушка является стабом

Все остальные различия между пятью разновидностями являются незначительными. Например, шпионы выполняют ту же функцию, что и моки, а отличаются от них лишь тем, что шпионы пишутся вручную, тогда как моки создаются при помощи мок-фреймворков.

С другой стороны, различия между стабом, пустышкой и фейком в том, насколько они сложны. *Пустышка* представляет собой простое, зашитое в код значение (к примеру, `null` или произвольная строка). Она используется для удовлетворения сигнатуры метода тестируемой системы и не влияет на результат. *Стаб* имеет более сложное устройство. Это полноценная зависимость, которая настраивается для воз-вращения разных значений для разных сценариев. Наконец, *фейк* в большинстве случаев — то же самое, что и стаб. Различие — в цели их создания: фейк обычно создается для замены зависимостей, которых еще не существует.

Обратите внимание на различия между моками и стабами (не считая направления взаимодействий). Моки помогают *эмодировать* и *проверять* взаимодействия между тестируемой системой и ее зависимостями, тогда как стабы только *эмодируют* эти взаимодействия. Это принципиальное отличие — вскоре вы поймете почему.

5.1.2. Мок-инструмент и мок — тестовая заглушка

Термин «мок» может означать разные вещи в разных обстоятельствах. В главе 2 я упоминал о том, что он часто используется для обозначения любых тестовых заглушек, тогда как на самом деле моки составляют лишь подмножество тестовых заглушек. Однако у термина существует и другой смысл. Классы мок-библиотек также могут называться моками. Такие классы помогают создавать моки, но сами по себе моками не являются. В листинге 5.1 приведен пример.

Листинг 5.1. Применение класса Mock из мок-библиотеки для создания мока

```
[Fact]
public void Sending_a_greetings_email()
{
    var mock = new Mock<IEmailGateway>();
    var sut = new Controller(mock.Object);

    sut.GreetUser("user@email.com");

    mock.Verify(
        x => x.SendGreetingsEmail(
            "user@email.com"),
        Times.Once);
}
```

Мок-инструмент используется для создания мока — тестовой заглушки

Проверяет вызов от
тестируемой системы
к тестовой заглушки

Тест в листинге 5.1 использует класс `Mock` из мок-библиотеки `Moq`. Этот класс позволяет создать тестовую заглушку — мок. Другими словами, класс `Mock` (или `Mock<IEmailGateway>`) — мок-инструмент, тогда как экземпляр этого класса (`mock`) — мок — тестовая заглушка. Важно не путать эти два понятия, потому что мок-инструмент может использоваться для создания обоих типов тестовых заглушек: моков и стабов.

Тест в листинге 5.2 также использует класс `Mock`, но экземпляр этого класса уже является не моком, а стабом.

Листинг 5.2. Применение класса Mock для создания стаба

```
[Fact]
public void Creating_a_report()
{
    var stub = new Mock<IDatabase>();
    stub.Setup(x => x.GetNumberOfUsers())
        .Returns(10);
    var sut = new Controller(stub.Object);

    Report report = sut.CreateReport();

    Assert.Equal(10, report.NumberOfUsers);
}
```

Мок-инструмент используется для создания стаба

Настраивает ответ

Эта тестовая заглушка эмулирует входное взаимодействие — вызов, предоставляющий тестируемой системе входные данные. С другой стороны, в предыдущем примере (листинг 5.1) вызов `SendGreetingsEmail()` является выходным взаимодействием. Его единственной целью является создание побочного эффекта — отправки электронной почты. Побочный эффект (*side effect*) — это изменение состояния чего-либо; в примере из листинга 5.1 — изменение состояния сервера SMTP.

5.1.3. Не проверяйте взаимодействия со стабами

Как упоминалось в разделе 5.1.1, моки помогают эмулировать и проверять выходные взаимодействия между тестируемой системой и ее зависимостями, тогда как стабы помогают только эмулировать входные взаимодействия, но не проверять их. Это различие является следствием следующего правила: *никогда не проверяйте взаимодействия со стабами*. Вызов от тестируемой системы к стабу не является частью конечного результата этой тестируемой системы. Такой вызов — всего лишь средство для получения результата; стаб предоставляет входные данные, по которым тестируемая система затем генерирует результат.

ПРИМЕЧАНИЕ

Проверка взаимодействий со стабами — часто встречающийся антипаттерн, приводящий к хрупкости тестов.

Как вы, возможно, помните из главы 4, избежать ложных срабатываний (а следовательно, улучшить устойчивость тестов к рефакторингу) можно только одним способом: эти тесты должны проверять конечный результат (который в идеале должен быть осмысленным для непрограммиста), а не детали имплементации. В листинге 5.1 проверка

```
mock.Verify(x => x.SendGreetingsEmail("user@email.com"))
```

является частью конечного результата, и этот результат имеет смысл для эксперта в предметной области: отправка приветственного сообщения по электронной почте — операция понятная для представителей бизнеса. В то же время вызов `GetNumberOfUsers()` в листинге 5.2 вообще не является результатом. Это внутренняя деталь имплементации, касающаяся того, как тестируемая система собирает данные, необходимые для создания отчета. Таким образом, проверка этого вызова в teste сделает этот тест хрупким: тест не должен интересоваться, как тестируемая система генерирует конечный результат, при условии что этот результат правилен. В листинге 5.3 показан пример такого хрупкого теста.

Эта практика проверки того, что не является частью конечного результата, также называется *излишней спецификацией* (*overspecification*). Чаще всего излишняя спецификация встречается при проверке взаимодействий. Проверка взаимодействий

со стабами — ошибка, которую довольно легко обнаружить, потому что тесты не должны проверять *никакие* взаимодействия со стабами. С моками дело обстоит сложнее; не все, но многие использований моков приводят к хрупкостям тестов. Позднее в этой главе я покажу, почему это происходит.

Листинг 5.3. Проверка взаимодействия со стабом

```
[Fact]
public void Creating_a_report()
{
    var stub = new Mock<IDatabase>();
    stub.Setup(x => x.GetNumberOfUsers()).Returns(10);
    var sut = new Controller(stub.Object);

    Report report = sut.CreateReport();

    Assert.Equal(10, report.NumberOfUsers());
    stub.Verify(
        x => x.GetNumberOfUsers(),
        Times.Once);    | Проверяет взаимодействие
                      | со стабом
}
```

5.1.4. Использование моков вместе со стабами

Иногда требуется создать тестовую заглушку, которая имеет свойства как мока, так и стаба. Для примера возьмем тест из главы 2, который использовался для демонстрации лондонского стиля юнит-тестирования.

Листинг 5.4. storeMock: одновременно и мок, и стаб

```
[Fact]
public void Purchase_fails_when_not_enough_inventory()
{
    var storeMock = new Mock<IStore>();
    storeMock
        .Setup(x => x.HasEnoughInventory(
            Product.Shampoo, 5))
        .Returns(false);    | Настраивает
                           | ответ
    var sut = new Customer();

    bool success = sut.Purchase(
        storeMock.Object, Product.Shampoo, 5);

    Assert.False(success);
    storeMock.Verify(
        x => x.RemoveInventory(Product.Shampoo, 5),
        Times.Never);    | Проверяет вызов
                           | из тестируемой системы
}
```

Тест использует `storeMock` для двух целей: возвращает фиксированный ответ и проверяет вызов из тестируемой системы. Обратите внимание, что здесь задействованы

два разных метода: тест настраивает ответ от `HasEnoughInventory()`, но затем проверяет вызов `RemoveInventory()`. Таким образом, правило о недопустимости проверки взаимодействий со стабами здесь не нарушается.

Когда тестовая заглушка является одновременно и моком, и стабом, она все равно называется моком, а не стабом. В основном это объясняется тем, что нужно выбрать одно имя, но отчасти и тем, что факт принадлежности к мокам важнее принадлежности к стабам.

5.1.5. Связь моков и стабов с командами и запросами

Концепции моков и стабов связаны с принципом CQS (Command Query Separation, то есть «разделение команд и запросов»). Принцип CQS утверждает, что каждый метод должен быть либо командой, либо запросом, но не и тем и другим одновременно. Как показано на рис. 5.3, команды — методы, которые производят побочные эффекты и не возвращают никакого значения (возвращают `void`). Примеры побочных эффектов: изменение состояния объекта, изменение файла в файловой системе и т. д. Запросы определяются противоположным образом — они не производят побочных эффектов и возвращают значение.

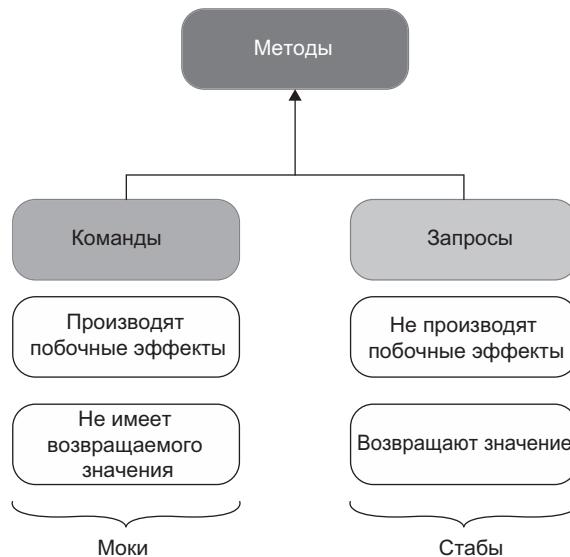


Рис. 5.3. В принципе разделения команд и запросов (CQS) команды соответствуют мокам, а запросы — стабам

Чтобы не нарушать этот принцип, следите за тем, чтобы методы, производящие побочный эффект, возвращали `void`. А если метод возвращает значение, он не должен

производить побочных эффектов. Другими словами, задание вопроса не должно изменять ответа. Код с таким четким разделением проще читается. Чтобы определить, что делает метод, достаточно посмотреть на его сигнатуру, не углубляясь в подробности реализации.

Конечно, полное следование принципу CQS не всегда возможно. Существуют методы, для которых оправдано как наличие побочного эффекта, так и возвращение значения. Классический пример — `stack.Pop()`. Этот метод одновременно и удаляет верхний элемент стека, и возвращает его вызывающей стороне. Тем не менее там, где это возможно, желательно придерживаться принципа CQS.

Тестовые заглушки, заменяющие команды, становятся моками. Аналогичным образом тестовые заглушки, заменяющие запросы, становятся стабами. Посмотрите еще раз на тесты из листингов 5.1 и 5.2:

```
var mock = new Mock<IEmailGateway>();
mock.Verify(x => x.SendGreetingsEmail("user@email.com"));

var stub = new Mock<IDatabase>();
stub.Setup(x => x.GetNumberOfUsers()).Returns(10);
```

`SendGreetingsEmail()` — команда, побочным эффектом которой является отправка сообщения. Тестовая заглушка, заменяющая эту команду, является моком. С другой стороны, `GetNumberOfUsers()` — запрос, который возвращает значение и не изменяет состояние базы данных. Соответствующая тестовая заглушка является стабом.

5.2. Наблюдаемое поведение и детали имплементации

В разделе 5.1 показано, что такое мок. Следующим шагом на пути к объяснению связи между моками и хрупкостью тестов будет разбор того, что же становится причиной этой хрупкости.

Как говорилось в главе 4, хрупкость тестов соответствует второму атрибуту хорошего юнит-теста: устойчивости к рефакторингу. (Напомню все четыре атрибута: защита от багов, устойчивость к рефакторингу, быстрая обратная связь и простота поддержки.) Метрика устойчивости к рефакторингу самая важная, так как наличие у юнит-теста этой устойчивости — по большей части бинарный выбор (она либо есть, либо нет). Следовательно, эту метрику желательно держать на максимально возможном уровне, при условии что тест остается юнит-тестом и не переходит в категорию сквозных тестов. Последние, несмотря на лучшие показатели в области устойчивости к рефакторингу, обычно создают гораздо больше проблем с сопровождением.

В главе 4 также было показано, что главная причина, по которой тесты генерируют ложные срабатывания (и таким образом теряют устойчивость к рефакторингу), заключается в их привязке к деталям имплементации тестируемого кода. Избежать

такой привязки можно только одним способом: проверять конечный результат, (наблюдаемое поведение системы) и отделять тесты от деталей имплементации настолько, насколько это возможно. Другими словами, тесты должны сосредоточиться на том, что система делает, а не на том, как она это делает. Что же именно является деталью имплементации и чем она отличается от наблюдаемого поведения?

5.2.1. Наблюдаемое поведение — не то же самое, что публичный API

Весь рабочий код можно классифицировать по двум измерениям:

- публичный или приватный API (API — программный интерфейс);
- наблюдаемое поведение или детали имплементации.

Категории в этих измерениях не пересекаются. Метод не может принадлежать как к публичному, так и приватному API; он относится либо к одной, либо к другой категории. Аналогичным образом код является либо внутренней деталью имплементации, либо частью наблюдаемого поведения системы, но не и тем и другим сразу.

Многие языки программирования предоставляют простые механизмы для разделения публичных и приватных API в коде. Например, в C# можно пометить любой компонент класса ключевым словом `private` и таким образом скрыть его от клиентского кода, делая частью приватного API класса. То же относится к классам: их легко можно объявить приватными при помощи ключевого слова `private` или `internal`.

Различия между наблюдаемым поведением и деталями имплементации менее очевидны. Чтобы код мог считаться частью наблюдаемого поведения системы, он должен решать одну из следующих задач:

- предоставлять операцию, которая помогает клиенту достичь одну из его целей. *Операция* — метод, который выполняет вычисление и/или создает побочный эффект;
- предоставлять доступ к состоянию системы, которое помогает клиенту достичь одну из его целей. *Состояние* — текущее состояние системы.

Любой код, который не делает ни того ни другого, является деталью имплементации.

Вопрос о том, относится ли код к наблюдаемому поведению, зависит от того, кто является клиентом и каковы его цели. Чтобы код был частью наблюдаемого поведения, этот код должен быть непосредственно связан хотя бы с одной такой целью. Термином «клиент» могут обозначаться разные понятия в зависимости от того, где размещается код. Типичные примеры — клиентский код из той же кодовой базы, внешнее приложение или пользовательский интерфейс.

В идеале публичный API системы должен совпадать с ее наблюдаемым поведением, а все детали имплементации должны быть скрыты от клиента. Такая система будет иметь хорошо спроектированный API (рис. 5.4).

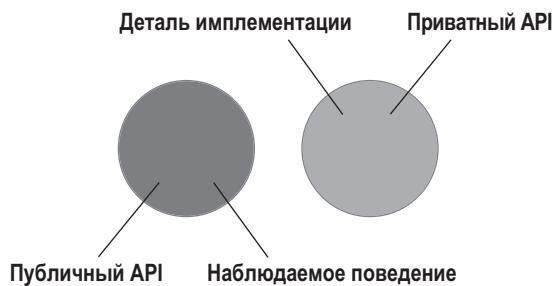


Рис. 5.4. В хорошо спроектированном API наблюдаемое поведение совпадает с публичным API, тогда как все детали имплементации скрываются за приватным API

Однако публичный API системы часто выходит за рамки наблюдаемого поведения и начинает раскрывать детали имплементации. Происходит утечка этих деталей в публичный API (рис. 5.5).

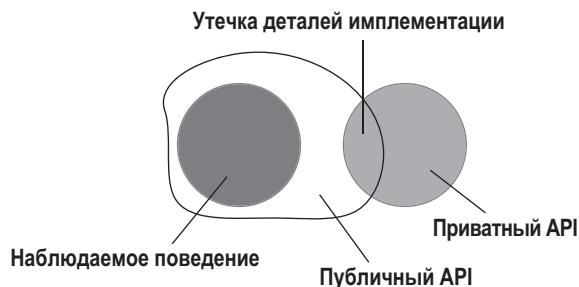


Рис. 5.5. Утечка деталей имплементации при распространении открытого API за пределы наблюдаемого поведения

5.2.2. Утечка деталей имплементации: пример с операцией

Рассмотрим примеры кода, в котором происходит утечка деталей имплементации в публичный API. В листинге 5.5 показан класс `User` с публичным API, состоящим из двух компонентов: свойства `Name` и метода `NormalizeName()`. У класса также имеется инвариант: имена пользователей не должны быть длиннее 50 символов; более длинные имена усекаются.

Клиентом здесь является `UserController`. Он использует класс `User` в своем методе `RenameUser`. Цель этого метода, как вы, вероятно, догадались, — изменение имени пользователя.

Итак, почему же API `User`-а спроектирован плохо? Еще раз взгляните на его компоненты: свойство `Name` и метод `NormalizeName`. Оба компонента являются публичными.

Следовательно, чтобы API класса можно было назвать хорошо спроектированным, эти компоненты должны быть частью наблюдаемого поведения. В свою очередь, для этого они должны решать одну из двух задач:

- предоставлять операцию, которая помогает клиенту достичь одну из его целей;
- предоставить доступ к состоянию, которое помогает клиенту достичь одну из его целей.

Листинг 5.5. Класс User с утечкой деталей имплементации

```
public class User
{
    public string Name { get; set; }
    public string NormalizeName(string name)
    {
        string result = (name ?? "").Trim();

        if (result.Length > 50)
            return result.Substring(0, 50);

        return result;
    }
}

public class UserController
{
    public void RenameUser(int userId, string newName)
    {
        User user = GetUserFromDatabase(userId);

        string normalizedName = user.NormalizeName(newName);
        user.Name = normalizedName;

        SaveUserToDatabase(user);
    }
}
```

Только свойство `Name` удовлетворяет этому требованию. Оно предоставляет `set`-метод — операцию, которая позволяет `UserController` достичь своей цели по изменению имени пользователя. Метод `NormalizeName` также является операцией, но он не связан напрямую с целью клиента. `UserController` вызывает этот метод только по одной причине: для удовлетворения инварианта `User`. Следовательно, `NormalizeName` является деталью имплементации, которая проникает в публичный API класса (рис. 5.6).

Чтобы исправить ситуацию и сделать API класса хорошо спроектированным, класс `User` должен скрыть `NormalizeName()` и вызвать его во внутренней реализации как часть `set`-метода свойства, не полагаясь на то, что это будет сделано в клиентском коде. Этот подход продемонстрирован в листинге 5.6.

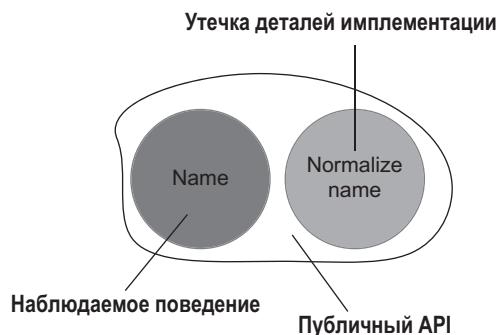


Рис. 5.6. API класса User не является хорошо спроектированным; он открывает доступ к методу NormalizeName, который не является частью наблюдаемого поведения

Листинг 5.6. Версия User с хорошо спроектированным API

```
public class User
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = NormalizeName(value);
    }

    private string NormalizeName(string name)
    {
        string result = (name ?? "").Trim();

        if (result.Length > 50)
            return result.Substring(0, 50);

        return result;
    }
}

public class UserController
{
    public void RenameUser(int userId, string newName)
    {
        User user = GetUserFromDatabase(userId);
        user.Name = newName;
        SaveUserToDatabase(user);
    }
}
```

API класса User в листинге 5.6 хорошо спроектирован: открыто только наблюдаемое поведение (свойство Name), а детали имплементации (метод NormalizeName) скрыты за приватным API (рис. 5.7).

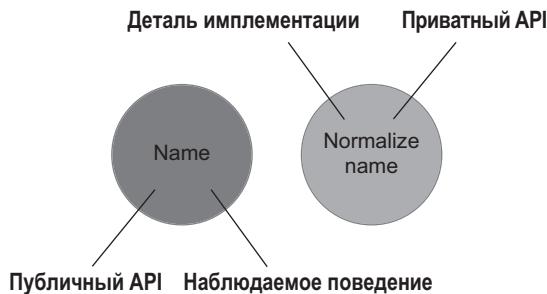


Рис. 5.7. Класс User с хорошо спроектированным API. Открытым является только наблюдаемое поведение; подробности реализации стали приватными

Существует хорошее эмпирическое правило, которое поможет вам определить, происходит ли в классе утечка деталей имплементации. Если количество операций, которые должны быть вызваны клиентом для достижения одной цели, больше 1, то в классе, скорее всего, происходит утечка деталей имплементации. В идеале каждая отдельная цель должна достигаться одной операцией. Так, в листинге 5.5 классу UserController приходится использовать две операции User:

```
string normalizedName = user.NormalizeName(newName);
user.Name = normalizedName;
```

После рефакторинга количество операций сократилось до одной:

```
user.Name = newName;
```

По моему опыту, это эмпирическое правило выполняется в подавляющем большинстве случаев, в которых задействована бизнес-логика, но возможны исключения. Анализируйте каждую ситуацию, в которой ваш код нарушает это правило, на предмет утечки деталей имплементации.

ПРИМЕЧАНИЕ

Строго говоря, get-метод Name тоже должен быть приватным, потому что он не используется в UserController. Однако на практике вам почти всегда требуется прочитать внесенные изменения. Таким образом, в реальном проекте наверняка будет другой сценарий, требующий получения текущих имен пользователей get-методом Name.

5.2.3. Хорошо спроектированный API и инкапсуляция

Поддержание хорошо спроектированного API напрямую связано с понятием инкапсуляции. Как говорилось в главе 3, *инкапсуляцией* называется защита вашего кода от нарушений логической целостности, также называемых *нарушениями инвариантов*. Инвариантом называется условие, которое должно поддерживаться

вашим приложением постоянно. У класса `User` из предыдущего примера есть один такой инвариант: длина имени пользователя не должна превышать 50 символов.

Утечка деталей имплементации идет рука об руку с нарушениями инвариантов — первое часто ведет ко второму. К примеру, в исходной версии `User` происходила не только утечка деталей имплементации — эта версия также не обеспечивала нормальной инкапсуляции. Это позволяло клиенту обойти инвариант и присвоить новое имя пользователю без предварительной нормализации этого имени.

Инкапсуляция играет важную роль в сопровождаемости кодовой базы в долгосрочной перспективе. Причина кроется в сложности. Сложность кода — одна из самых больших проблем, возникающих в процессе разработки. Чем сложнее становится кодовая база, тем труднее с ней работать; в свою очередь, это замедляет разработку и повышает количество ошибок.

Без инкапсуляции справиться с постоянным ростом сложности кода практически невозможно. Если API не указывает вам на то, что разрешено или не разрешено делать в коде, вам придется держать в голове большое количество информации, чтобы не внести ошибки и логические несоответствия при новых изменениях. Это создает дополнительную когнитивную нагрузку при программировании. Вы должны по возможности избавляться от этой нагрузки. *Нельзя быть уверенными в том, что вы каждый раз будете использовать код правильно, поэтому исключите саму возможность использовать его неправильно.* Для этого всегда поддерживайте инкапсуляцию кода, чтобы он не давал вам возможности внести ошибку. Инкапсуляция в конечном счете служит той же цели, что и юнит-тестирование: она обеспечивает стабильный рост проекта в долгосрочной перспективе.

Существует похожий принцип: tell-don't-ask. Он был сформулирован Мартином Фаулером (<https://martinfowler.com/bliki/TellDontAsk.html>) и означает упаковку данных с функциями, работающими с этими данными. Этот принцип может рассматриваться как следствие из принципа инкапсуляции. Инкапсуляция кода является целью, тогда как совместная упаковка данных и функций, а также сокрытие деталей имплементации — средствами для достижения этой цели:

- сокрытие деталей имплементации убирает внутреннее устройство класса от клиентов, таким образом снижая риск его повреждения;
- упаковка данных вместе с операциями помогает предотвратить нарушение инвариантов класса этими операциями.

5.2.4. Утечка деталей имплементации: пример с состоянием

В листинге 5.5 была продемонстрирована операция (метод `NormalizeName`), которая была деталью имплементации, проникшей в публичный API. Теперь рассмотрим пример с состоянием. В листинге 5.7 приведен класс `MessageRenderer`, уже

упоминавшийся в главе 4. Он использует коллекцию подгенераторов для построения сообщения, содержащего заголовок, тело и колонтитул (footer), в формате HTML.

Листинг 5.7. Состояние как деталь имплементации

```
public class MessageRenderer : IRenderer
{
    public IReadOnlyList<IRenderer> SubRenderers { get; }

    public MessageRenderer()
    {
        SubRenderers = new List<IRenderer>
        {
            new HeaderRenderer(),
            new BodyRenderer(),
            new FooterRenderer()
        };
    }

    public string Render(Message message)
    {
        return SubRenderers
            .Select(x => x.Render(message))
            .Aggregate("", (str1, str2) => str1 + str2);
    }
}
```

Коллекция подгенераторов публична. Но является ли она частью наблюдаемого поведения? Если считать, что целью клиента является построение сообщения HTML, ответ будет отрицательным. Единственный компонент класса, который понадобится такому клиенту, — это сам метод `Render`. Таким образом, `SubRenderers` также является утечкой деталей имплементации.

Я снова привожу этот пример не просто так. Как вы помните, я использовал его для демонстрации хрупкости тестов. Этот тест был хрупким именно потому, что был привязан к деталям имплементации — он проверял структуру коллекции. Хрупкость была исправлена перенаправлением теста на метод `Render`. Новая версия теста проверяла итоговое сообщение — единственный результат, который важен для клиентского кода, это наблюдаемое поведение.

Как видите, между хорошими юнит-тестами и хорошо спроектированным API существует связь. Делая все детали имплементации приватными, вы не оставляете своим тестам другого выбора, кроме как проверять наблюдаемое поведение кода, что автоматически повышает их устойчивость к рефакторингу.

СОВЕТ

Переход к хорошо спроектированному API автоматически улучшает юнит-тесты.

Существует еще одна рекомендация, которая следует из определения хорошо спроектированного API: необходимо раскрывать минимальное количество операций и состояния. Открытым должен становиться только тот код, который непосредственно помогает клиентам достигать их целей. Все остальное относится к деталям имплементации, а следовательно, должно быть скрыто за приватным API.

Обратите внимание: не существует такой проблемы, как утечка наблюдаемого поведения, которая была бы симметрична проблеме утечки деталей имплементации. Хотя вы можете раскрыть детали имплементации (метод или класс, не предназначенный для использования клиентом), скрыть наблюдаемое поведение невозможно. Такой метод или класс уже не будет иметь прямой связи с целями клиента, потому что клиент не сможет напрямую использовать его. Таким образом, этот код по определению перестанет быть частью наблюдаемого поведения (см. таблицу 5.1).

Таблица 5.1. Связь между публичностью и назначением кода. Избегайте раскрытия деталей имплементации

	Наблюдаемое поведение	Детали имплементации
Публичный код	Хорошо	Плохо
Приватный код	–	Хорошо

5.3. Связь между моками и хрупкостью тестов

В предыдущих разделах было приведено определение мока и продемонстрированы различия между наблюдаемым поведением и деталями имплементации. В этом разделе рассматривается гексагональная архитектура, различия между внутренними и внешними взаимодействиями, а также (наконец-то!) связь между моками и хрупкостью тестов.

5.3.1. Определение гексагональной архитектуры

Типичное приложение состоит из двух слоев: предметной области (domain layer) и сервисов приложения (application services) (рис. 5.8). Слой *предметной области* находится в середине диаграммы, потому что он является центральной частью вашего приложения. Он содержит *бизнес-логику* — функциональность, ради которой строится ваше приложение. Слой предметной области и его бизнес-логика — это то, что отличает ваше приложение от других, обеспечивая конкурентное преимущество для компании.

Слой сервисов приложения располагается поверх слоя предметной области и координирует взаимодействие между этим слоем и внешним миром. Например, если ваше приложение использует REST API, то все запросы к этому API сначала попадают в слой сервисов приложения. Затем этот слой координирует работу между

классами предметной области и внепроцессными зависимостями. Как пример такой координации, сервис приложения может:

- обращаться к базе данных и использовать полученные данные для создания экземпляра доменного класса (класса из слоя предметной области);
- вызвать операцию в этом экземпляре;
- сохранить результаты обратно в базу данных.

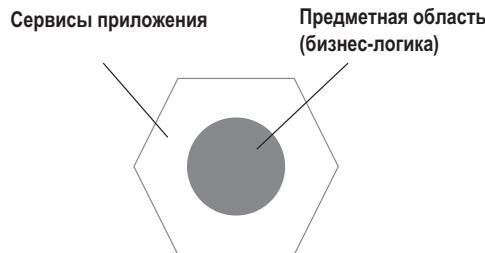


Рис. 5.8. Типичное приложение состоит из слоя предметной области и слоя сервисов приложения. Слой предметной области содержит бизнес-логику приложения; сервисы приложения связывают эту логику с бизнес-сценариями (use cases)

Комбинация слоя сервисов приложения и слоя предметной области образует *гексагон*, который представляет ваше приложение. Он может взаимодействовать с другими приложениями, представленными другими гексагонами (рис. 5.9). Другими приложениями могут быть SMTP-сервисы, сторонние системы, шины сообщений и т. д. Группа взаимодействующих гексагонов образует *гексагональную архитектуру*.



Рис. 5.9. Гексагональная архитектура состоит из набора взаимодействующих приложений, изображаемых в виде гексагонов

Термин «гексагональная архитектура» был предложен Алистером Кокберном (Alistair Cockburn). Он подчеркивает три важных принципа:

- *Разделение обязанностей между слоем предметной области и слоем сервисов приложения.* Бизнес-логика является важнейшей частью приложения. Следовательно, слой предметной области должен отвечать только за эту бизнес-логику и быть избавленным от любых других обязанностей. Эти обязанности — такие как взаимодействие с внешними приложениями и чтение информации из базы данных — должны быть возложены на сервисы приложения. И наоборот, сервисы приложения не должны содержать никакой бизнес-логики. Их обязанностью является адаптация уровня предметной области путем преобразования входных запросов в операции на доменных классах и последующее сохранение результатов или возвращение их вызывающей стороне. Слой предметной области может рассматриваться как коллекция знаний о предметной области (они отвечают на вопрос «как?»), а уровень сервисов приложения — как коллекция бизнес-сценариев (они отвечают на вопрос «что?»).
- *Взаимодействия внутри приложения.* Гексагональная архитектура предписывает односторонние зависимости — от слоя сервисов приложения к слою предметной области. Доменные классы должны зависеть только друг от друга, они не должны зависеть от классов слоя сервисов приложения. Это правило вытекает из предыдущего. Разделение обязанностей между слоем сервисов приложения и слоем предметной области означает, что первый располагает информацией о втором, но не наоборот. Слой предметной области должен быть полностью изолирован от внешнего мира.
- *Взаимодействия между приложениями.* Внешние приложения связываются с вашим приложением через общий интерфейс, поддерживаемый слоем сервисов приложения. Никто не может обращаться напрямую к слою предметной области. Каждая сторона гексагона представляет собой соединение к приложению извне или из приложения вовне. Хотя у гексагона шесть сторон, это не означает, что ваше приложение может быть связано только с шестью другими приложениями. Количество соединений не имеет значения. Важно то, что таких соединений может быть несколько.

Каждый слой приложения имеет свое наблюдаемое поведение и содержит собственный набор деталей имплементации. Например, наблюдаемое поведение слоя предметной области является совокупностью операций и состояний данного слоя, которая помогает слою сервисов приложения достигнуть по крайней мере одной из его целей. Принципы хорошо спроектированного API имеют фрактальную природу: они в равной степени применимы как ко всему слою, так и к его отдельным классам.

Когда вы делаете API каждого слоя хорошо спроектированным (то есть скрываете его детали имплементации), ваши тесты также обретают фрактальную структуру: они проверяют поведение, которое помогает достигнуть одинаковых целей, но на

разных уровнях. Тест, покрывающий слой сервисов приложения, проверяет, каким образом этот сервис достигает общей цели, поставленной внешним клиентом. В то же время тест, работающий с доменным классом, проверяет промежуточную цель, которая является частью общей цели (рис. 5.10).

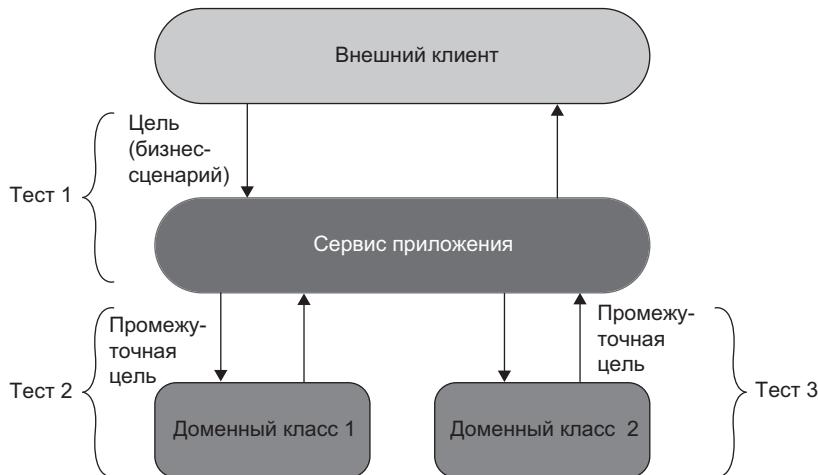


Рис. 5.10. Тесты, работающие на разных уровнях, имеют фрактальную природу: они проверяют одно и то же поведение на разных уровнях. Тест сервисов приложения проверяет, как выполняется бизнес-сценарий в целом. Тест, работающий со слоем предметной области, проверяет промежуточную цель на пути к выполнению этого бизнес-сценария

В предыдущих главах я уже упоминал о том, что любой тест должен ассоциироваться с конкретным бизнес-требованием. Каждый тест должен рассказывать историю, осмысленную для эксперта предметной области, а если он этого не делает — это верный признак того, что тест завязан на детали имплементации (а следовательно, является хрупким). Надеюсь, теперь понятно, почему это так.

Наблюдаемое поведение «проходит» от внешних слоев к центру. Общая цель, поставленная внешним клиентом, преобразуется в промежуточные цели, достижимые отдельными классами предметной области. Таким образом, каждый фрагмент наблюдаемого поведения в слое предметной области сохраняет связь с конкретным бизнес-сценарием. Эту связь можно проследить рекурсивно от внутреннего слоя (слоя предметной области) к слою сервисов приложения, а затем к потребностям внешнего клиента. Возможность отследить «текущее» наблюдаемого поведения следует из определения этого термина. Чтобы код был частью наблюдаемого поведения, он должен помогать клиенту в достижении одной из его целей. Для класса предметной области клиентом является сервис приложения; для сервиса приложения — это сам внешний клиент.

Тесты, проверяющие код с хорошо спроектированным API, тоже связаны с бизнес-требованиями, потому что эти тесты завязываются только на наблюдаемое поведение кода. Хорошим примером станут классы `User` и `UserController` из листинга 5.8.

Листинг 5.8. Класс предметной области с сервисом приложения

```
public class User
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = NormalizeName(value);
    }

    private string NormalizeName(string name)
    {
        /* Усечение имени до 50 символов */
    }
}

public class UserController
{
    public void RenameUser(int userId, string newName)
    {
        User user = GetUserFromDatabase(userId);
        user.Name = newName;
        SaveUserToDatabase(user);
    }
}
```

`UserController` в этом примере является сервисом приложения. Если предположить, что внешний клиент не имеет цели нормализации имен пользователей, а все имена нормализуются исключительно из-за ограничений самого приложения, метод `NormalizeName` в классе `User` не связан с потребностями клиента. Следовательно, он является деталью имплементации, и его следует сделать приватным (это уже было сделано ранее в этой главе). Кроме того, тесты не должны проверять этот метод напрямую. Они должны проверять его только как часть наблюдаемого поведения класса — `set`-метода свойства `Name` в данном примере.

Правило о том, что открытый API кода всегда должен прослеживаться до бизнес-требований, применимо к подавляющему большинству доменных классов и классов сервисов приложения, но в меньшей степени — к служебному и инфраструктурному коду. Задачи, решаемые таким кодом, часто оказываются слишком низкоуровневыми, чтобы связать их с конкретным бизнес-сценарием.

5.3.2. Внутрисистемные и межсистемные взаимодействия

В типичном приложении встречаются взаимодействия двух видов: внутрисистемные и межсистемные. *Внутрисистемные взаимодействия* — это взаимодействия между

классами внутри вашего приложения. *Межсистемные* взаимодействия — это взаимодействия между вашим приложением и внешними приложениями (рис. 5.11).

Внутрисистемные взаимодействия являются деталями имплементации, потому что взаимодействия, через которые проходят классы предметной области для выполнения какой-то задачи, не являются частью их наблюдаемого поведения. Эти взаимодействия не связаны напрямую с целью клиента. Таким образом, привязка тестов к таким взаимодействиям делает их хрупкими.

ПРИМЕЧАНИЕ

Внутрисистемные взаимодействия являются деталями имплементации; межсистемные взаимодействия — нет.

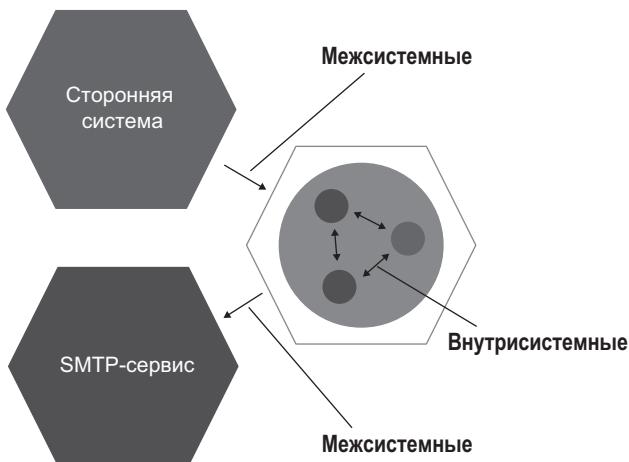


Рис. 5.11. Взаимодействия делятся на два типа: внутрисистемные (между классами в приложении) и межсистемные (между приложениями)

Межсистемные взаимодействия — совсем другое дело. В отличие от взаимодействий между классами внутри приложения, взаимодействие вашей системы с внешним миром формирует наблюдаемое поведение системы в целом. Это часть контракта, который должен постоянно соблюдаться приложением (рис. 5.12).

Это свойство межсистемных взаимодействий происходит из того, как разные приложения эволюционируют друг с другом. Один из основных принципов такой эволюции — сохранение обратной совместимости. Независимо от того, сколько рефакторинга делается внутри системы, взаимодействия этой системы с внешними приложениями всегда должны оставаться неизменными. Например, сообщения, передаваемые вашим приложением по шине, должны сохранять свою структуру; вызовы к сервису SMTP должны иметь то же количество и те же типы параметров и т. д.

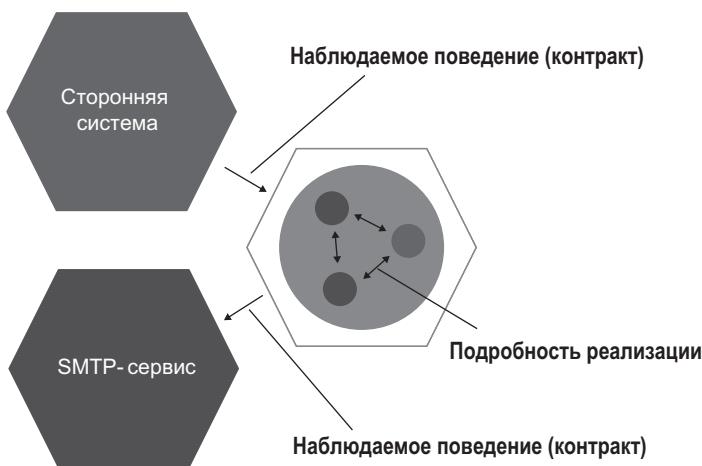


Рис. 5.12. Межсистемные взаимодействия формируют наблюдаемое поведение вашего приложения.
Внутрисистемные взаимодействия являются деталями имплементации

Моки эффективны при проверке взаимодействий между вашей системой и внешними приложениями. И наоборот, использование моков для проверки взаимодействий между классами внутри вашей системы приводит к хрупким тестам, так как они завязываются на детали имплементации и, как следствие, обладают низкой устойчивостью к рефакторингу.

5.3.3. Внутрисистемные и межсистемные взаимодействия: пример

Чтобы продемонстрировать различия между внутрисистемными и межсистемными взаимодействиями, мы доработаем пример с классами `Customer` и `Store`, который встречался в главе 2 и ранее в этой главе. Представьте следующий бизнес-сценарий:

- Заказчик пытается приобрести товар в магазине.
- Если запасы товара на складе достаточны, то:
 - товар удаляется со склада;
 - заказчику отправляется уведомление по электронной почте;
 - возвращается подтверждение.

Также будем считать, что приложение ограничивается API без пользовательского интерфейса.

В листинге 5.9 класс `CustomerController` представляет сервис приложения, координирующий работу между классами предметной области (`Customer`, `Product`, `Store`) и внешним приложением (`EmailGateway`, которое, в свою очередь, является прокси к SMTP-сервису).

Листинг 5.9. Связывание доменной модели с внешними приложениями

```
public class CustomerController
{
    public bool Purchase(int customerId, int productId, int quantity)
    {
        Customer customer = _customerRepository.GetById(customerId);
        Product product = _productRepository.GetById(productId);

        bool isSuccess = customer.Purchase(
            _mainStore, product, quantity);

        if (isSuccess)
        {
            _emailGateway.SendReceipt(
                customer.Email, product.Name, quantity);
        }

        return isSuccess;
    }
}
```

Проверка входных параметров опущена для краткости. В методе `Purchase()` `customer` проверяет, достаточен ли запас товара на складе, и если достаточен — уменьшает его на величину заказа.

Акт совершения покупки является бизнес-сценарием, в течение которого происходят как внутрисистемные, так и межсистемные взаимодействия. Межсистемные взаимодействия происходят между сервисом приложения `CustomerController` и двумя внешними системами: сторонним приложением (клиент, инициирующий бизнес-сценарий) и почтовым шлюзом. Внутрисистемные взаимодействия происходят между классами предметной области `Customer` и `Store` (рис. 5.13).

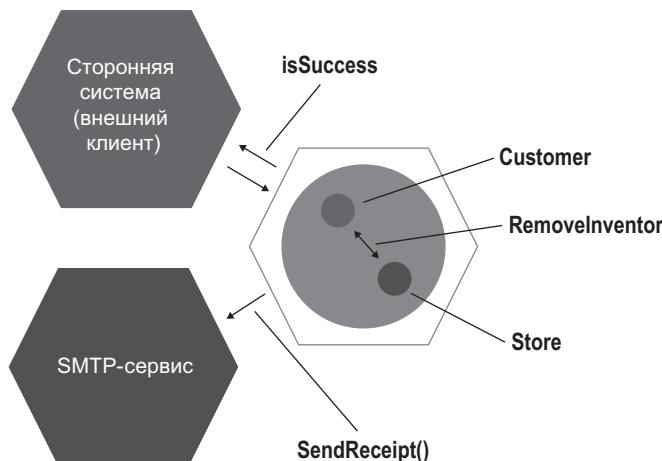


Рис. 5.13. Пример из листинга 5.9, представленный в гексагональной архитектуре. Взаимодействия между гексагонами являются межсистемными, а взаимодействия внутри гексагона — внутрисистемными

В данном примере обращение к сервису SMTP является побочным эффектом, который виден внешнему миру, а следовательно, является частью наблюдаемого поведения приложения.

Это обращение также напрямую связано с целями клиента. Клиентом приложения является сторонняя система. Цель системы — приобретение товара, и она ожидает, что заказчик получит подтверждающее сообщение как часть успешного результата.

Обращение к сервису SMTP — уважительная причина для применения моков. Оно не делает тест хрупким, потому что такие взаимодействия должны оставаться без изменения даже после рефакторинга. Использование моков — хороший способ убедиться в этом.

В листинге 5.10 представлен пример обоснованного применения моков.

Листинг 5.10. Применение мока не делает тест хрупким

```
[Fact]
public void Successful_purchase()
{
    var mock = new Mock<IEmailGateway>();
    var sut = new CustomerController(mock.Object);

    bool isSuccess = sut.Purchase(
        customerId: 1, productId: 2, quantity: 5);

    Assert.True(isSuccess);
    mock.Verify(
        x => x.SendReceipt(
            "customer@email.com", "Shampoo", 5),
        Times.Once);
}
```

Проверяет, что система отправила подтверждение после покупки

Обратите внимание: флаг `isSuccess` также виден для внешнего клиента и также нуждается в проверке. Однако мок для этого флага не нужен, простого сравнения значения будет достаточно.

Теперь давайте рассмотрим тест, который моделирует взаимодействие между `Customer` и `Store` с использованием мока.

В отличие от взаимодействия между `CustomerController` и SMTP-сервиса, вызов метода `RemoveInventory()` от `Customer` к `Store` не пересекает границу приложения: и вызывающая сторона, и получатель находятся внутри приложения. Кроме того, этот метод не является ни операцией, ни состоянием, которые помогают клиенту достигать его целей. Клиентом двух этих доменных классов является `CustomerController`, цель которого — приобретение товара. С этой целью непосредственно связаны только два компонента: `customer.Purchase()` и `store.GetInventory()`. Метод `Purchase()` инициирует покупку, а `GetInventory()` показывает состояние системы после завершения

этой покупки. Вызов метода `RemoveInventory()` является промежуточным шагом на пути к цели клиента — то есть деталью имплементации.

Листинг 5.11. Применение мока делает тест хрупким

```
[Fact]
public void Purchase_succeeds_when_enough_inventory()
{
    var storeMock = new Mock<IStore>();
    storeMock
        .Setup(x => x.HasEnoughInventory(Product.Shampoo, 5))
        .Returns(true);
    var customer = new Customer();

    bool success = customer.Purchase(
        storeMock.Object, Product.Shampoo, 5);

    Assert.True(success);
    storeMock.Verify(
        x => x.RemoveInventory(Product.Shampoo, 5),
        Times.Once);
}
```

5.4. Еще раз о различиях между классической и лондонской школами юнит-тестирования

В таблице 5.2 приведена сводка различий между классической и лондонской школами юнит-тестирования из главы 2 (таблица 2.1).

Таблица 5.2. Различия между лондонской и классической школами юнит-тестирования в отношении подхода к изоляции, размеру юнита и использованию тестовых заглушек

	Изоляция	Юнит — это	Использование тестовых заглушек для
Лондонская школа	Юнитов	Класс	Коллабораторов (любых изменяющихся зависимостей)
Классическая школа	Юнит-тестов	Класс или набор классов	Совместных (<i>shared</i>) зависимостей

В главе 2 я упоминал о том, что предпочитаю классическую школу юнит-тестирования, и надеюсь, теперь вы видите почему. Лондонская школа рекомендует использовать моки для всех зависимостей, кроме неизменяемых, и не делает различий между внутрисистемными и межсистемными взаимодействиями. В результате тесты проверяют взаимодействия между классами внутри приложения так же часто, как и взаимодействия между вашим приложением и внешними системами.

Именно такое неизбирательное применение моков является причиной того, почему лондонская школа часто приводит к появлению тестов, завязанных на детали

имплементации — и как следствие, не обладающих устойчивостью к рефакторингу. Как упоминалось в главе 4, метрика устойчивости к рефакторингу (в отличие от трех остальных) в основном является бинарным выбором: тест либо обладает этой устойчивостью, либо нет. При любых уступках в этой метрике тест становится практически бесполезным.

Классическая школа гораздо лучше справляется с этой проблемой, потому что она предписывает замену на моки только тех зависимостей, которые совместно используются тестами, что почти всегда означает внепроцессные зависимости: SMTP-сервисы, шину сообщений и т. д. Впрочем, классическая школа тоже не идеальна в отношении межсистемных взаимодействий. Она тоже поощряет избыточное использование моков, хотя и не в такой степени, как лондонская школа.

5.4.1. Не все внепроцессные зависимости должны заменяться моками

Прежде чем переходить к обсуждению внепроцессных зависимостей и моков, я кратко напомню типы зависимостей (за подробностями обращайтесь к главе 2):

- *совместная (shared)* зависимость — зависимость, совместно используемая тестами (не рабочим кодом);
- *внепроцессная (out-of-process)* зависимость — зависимость, находящаяся в процессе, отличном от процесса, в котором выполняется программа (например, база данных, шина сообщений или SMTP-сервис);
- *приватная зависимость* — любая зависимость, не являющаяся совместной.

Классическая школа рекомендует обходиться без совместных зависимостей в тестах, потому что через них тесты могут влиять на результаты друг друга и таким образом помешать параллельному выполнению этих тестов. Возможность выполнения тестов параллельно, последовательно или в произвольном порядке называется *изоляцией тестов*.

Если совместная зависимость не является внепроцессной, можно легко избежать ее переиспользования в тестах, создавая новый ее экземпляр при каждом запуске теста. Если же совместная зависимость является внепроцессной, тестирование усложняется. Невозможно создавать новый экземпляр базы данных или шины сообщений перед каждым выполнением теста; это привело бы к слишком сильному замедлению тестов. Обычно в таких ситуациях эти зависимости заменяются тестовыми заглушками — моками и стабами.

Но не все внепроцессные зависимости должны заменяться моками. *Если внепроцессная зависимость доступна только через ваше приложение, взаимодействия с ней не являются частью наблюдаемого поведения вашей системы.* Внепроцессная зависимость, которую нельзя увидеть извне, по сути, является частью вашего приложения (рис. 5.14).

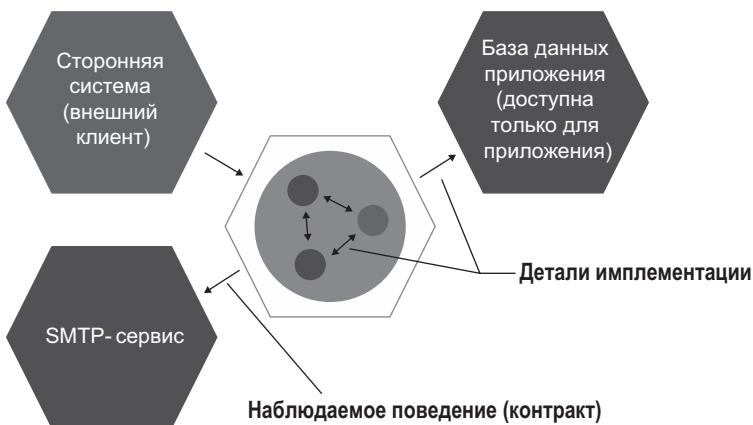


Рис. 5.14. Взаимодействия с внепроцессной зависимостью, которую нельзя увидеть извне, являются деталями имплементации. Они не обязаны оставаться без изменений после рефакторинга, а следовательно, не должны проверяться с использованием моков

Требование о сохранении схемы взаимодействий между вашим приложением и внешними системами происходит от необходимости сохранения обратной совместимости. Вам необходимо поддерживать формат, через который ваше приложение общается с внешними системами. Это объясняется тем, что вы не можете изменять эти внешние системы одновременно со своим приложением: у них может быть другой график развертывания или же они могут не находиться под вашим контролем.

Но когда ваше приложение действует как посредник для внешней системы и ни один клиент не может обратиться к этой системе напрямую, обратная совместимость перестает быть обязательным требованием. Теперь вы можете развертывать свое приложение вместе с внешней системой, и это не будет влиять на клиентов. Формат взаимодействия с такой системой становится деталью имплементации.

Хороший пример — база данных приложения, то есть база данных, которая используется только вашим приложением и к которой не имеют доступа внешние системы. Вы можете изменить формат взаимодействия между вашей системой и базой данных приложения так, как считаете нужным, при условии что это не нарушит существующей функциональности. Так как база данных полностью скрыта от клиентов вашего приложения, ее можно даже полностью заменить другой базой данных — никто этого не заметит.

Использование моков для внепроцессных зависимостей, находящихся под вашим полным контролем, также делает тесты хрупкими. Тесты не должны падать каждый раз, когда вы разбиваете таблицу на две или изменяете тип одного из параметров хранимой процедуры. База данных и приложение должны рассматриваться как единая система.

Здесь возникает очевидная проблема. Как тестировать работу с такими зависимостями без ущерба для скорости обратной связи, третьего атрибута хорошего юнит-теста? Эта тема подробно рассматривается в следующих двух главах.

5.4.2. Использование моков для проверки поведения

Часто говорят, что моки проверяют поведение. В подавляющем большинстве случаев они этого не делают. Способ взаимодействия каждого отдельного класса с соседними классами для достижения некоторой цели не имеет никакого отношения к наблюдаемому поведению, это деталь имплементации.

Проверка взаимодействий между классами сродни попыткам определить поведение человека посредством измерения сигналов, передаваемых между нейронами в его мозгу. Это слишком мелкий уровень детализации. Имеет значение только поведение, которое можно связать с целями клиента. Клиента не интересует, какие нейроны в вашем мозгу активизируются, когда он обращается к вам за помощью. Важна лишь сама помощь, которую вы оказываете, — ее качество и профессионализм. Моки имеют отношение к поведению только тогда, когда они проверяют взаимодействия, выходящие за границу приложения, и только когда побочные результаты таких взаимодействий видны внешнему миру.

Итоги

- Тестовая заглушка — общий термин, описывающий любые разновидности фиктивных зависимостей в тестах. Существуют пять типов тестовых заглушек (пустышки, стабы, шпионы, моки и фейки), которые можно разбить на две категории: моки и стабы. Шпионы функционально эквивалентны мокам; пустышки и фейки делают то же, что и стабы.
- Моки помогают эмулировать и проверять выходные взаимодействия — вызовы, совершаемые тестируемой системой к ее зависимостям для изменения их состояния. Стабы помогают эмулировать входные взаимодействия — вызовы, совершаемые тестируемой системой к ее зависимостям для получения входных данных.
- Мок-инструмент — класс мок-библиотеки, который может использоваться для создания мока — тестовой заглушки или стаба.
- Проверка взаимодействий со стабами делает тесты хрупкими. Такое взаимодействие не имеет отношения к конечному результату; это всего лишь промежуточный шаг на пути к такому результату, деталь имплементации.
- Принцип CQS (command query separation) утверждает, что каждый метод должен быть либо командой, либо запросом, но не и тем и другим одновременно. Тестовые заглушки, заменяющие команды, становятся моками. Тестовые заглушки, заменяющие запросы, становятся стабами.

- Весь рабочий код может быть классифицирован по двум измерениям: на публичные и приватные API, а также на наблюдаемое поведение и детали имплементации. Публичностью кода управляют модификаторы доступа (например, ключевые слова `public`, `private` и `internal`). Код является частью наблюдаемого поведения системы, если он удовлетворяет одному из следующих требований (весь прочий код относится к деталям имплементации):
 - он предоставляет операцию, которая помогает клиенту достичь одну из его целей. Операция — метод, который выполняет вычисление и/или создает побочный эффект;
 - он предоставляет доступ к состоянию системы, которое помогает клиенту достичь одну из его целей. Состояние — текущее состояние системы.
- В хорошо спроектированном коде наблюдаемое поведение совпадает с публичным API, тогда как все детали имплементации скрываются за приватным API. Утечка деталей имплементации происходит тогда, когда публичный API системы выходит за рамки наблюдаемого поведения и начинает раскрывать детали имплементации.
- Инкапсуляцией называется защита вашего кода от нарушений инвариантов. Раскрытие деталей имплементации часто приводит к нарушению инкапсуляции, потому что клиенты могут использовать детали имплементации, чтобы обойти инварианты кода.
- Гексагональная архитектура состоит из набора взаимодействующих приложений, изображаемых в виде гексагонов (шестиугольников). Каждый гексагон состоит из двух слоев: предметной области и сервисов приложений.
- Гексагональная архитектура подчеркивает три важных принципа:
 - разделение обязанностей между слоем предметной области и слоем сервисов приложения. Слой предметной области должен отвечать только за бизнес-логику, тогда как слой сервисов приложения должен координировать работу между слоем предметной области и внешними приложениями;
 - взаимодействия внутри приложения должны быть односторонними — от слоя сервисов приложения к слою предметной области. Доменные классы должны зависеть только друг от друга, они не должны зависеть от классов слоя сервисов приложения;
 - внешние приложения связываются с вашим приложением через общий интерфейс, поддерживаемый слоем сервисов приложения. Никто не может обращаться напрямую к слою предметной области.
- Каждый слой приложения имеет наблюдаемое поведение и содержит собственный набор деталей имплементации.
- В приложениях встречаются взаимодействия двух видов: внутрисистемные и межсистемные. Внутрисистемные взаимодействия связывают классы вашего

приложения. Межсистемные взаимодействия — это когда ваше приложение общается с другими приложениями.

- Внутрисистемные взаимодействия являются деталями имплементации. Межсистемные взаимодействия являются частью наблюдаемого поведения — за исключением внешних систем, доступных только через ваше приложение. Взаимодействия с такими системами также являются деталями имплементации, потому что порождаемые ими побочные эффекты не видны извне.
- Использование моков для проверки внутрисистемных взаимодействий делает тесты хрупкими. Моки должны использоваться только для межсистемных взаимодействий (взаимодействий, выходящих за границу приложения) и только когда эффекты таких взаимодействий видны внешнему миру.

Стили юнит-тестирования

В этой главе:

- ✓ Сравнение стилей юнит-тестирования.
- ✓ Отношение между функциональными и гексагональными архитектурами.
- ✓ Переход на тестирование выходных данных.

В главе 4 были представлены четыре атрибута хорошего юнит-теста: защита от багов, устойчивость к рефакторингу, быстрая обратная связь и простота поддержки. Эти атрибуты образуют систему координат, которая может использоваться для анализа тестов и методов юнит-тестирования. Один из таких методов — использование моков — был проанализирован в главе 5.

В этой главе я применю ту же систему координат к теме стилей юнит-тестирования. Существуют три таких стиля: проверка выходных данных, проверка состояния и проверка взаимодействий. Из этих трех вариантов проверка выходных данных дает тесты наивысшего качества. На втором месте стоит проверка состояния, а проверка взаимодействий должна применяться только в редких случаях.

К сожалению, проверка выходных данных не может использоваться повсеместно. Он применим только к коду, написанному в функциональном стиле. Но существуют методы, которые помогут вам преобразовать некоторые из ваших тестов к стилю проверки выходных данных. Для этого необходимо использовать принципы функционального программирования для реструктуризации рабочего кода, чтобы сделать этот код совместимым с функциональной архитектурой.

Следует заметить, что эта глава дает только самые основы функционального программирования. Тем не менее я надеюсь, что к концу этой главы вам станет понятно, как функциональное программирование связано с тестированием выходных данных. Вы также научитесь писать большую часть своих тестов в этом стиле, а также больше

узнаете об ограничениях функционального программирования и функциональной архитектуры.

6.1. Три стиля юнит-тестирования

Как я сказал во введении, существуют три стиля юнит-тестирования:

- проверка выходных данных;
- проверка состояния;
- проверка взаимодействий.

В одном тесте можно применить один, два или даже все три стиля. В этом разделе я расскажу об этих стилях (с примерами). В следующем разделе они сравниваются друг с другом.

6.1.1. Проверка выходных данных

Первый стиль юнит-тестирования — проверка выходных данных, при которой в тестируемую систему (SUT) подаются входные данные, после чего проверяется полученный результат (рис. 6.1). Этот стиль юнит-тестирования применим только к коду, который не изменяет глобального или внутреннего состояния, поэтому единственным компонентом, который нуждается в проверке, становится его возвращаемое значение.

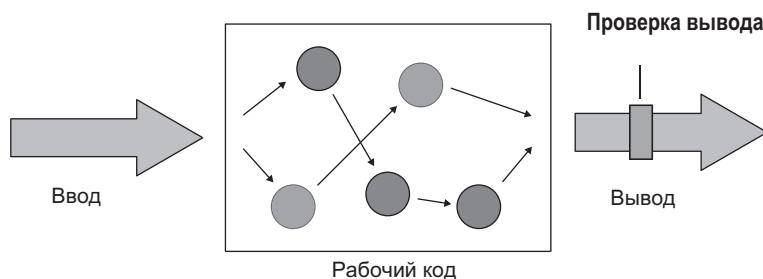


Рис. 6.1. При тестировании выходных данных тесты проверяют результат, генерируемый системой. Такой стиль тестирования предполагает отсутствие побочных эффектов, а единственным результатом работы тестируемой системы является возвращаемое значение

В листинге 6.1 приведен пример такого кода и покрывающего его теста. Класс `PriceEngine` получает массив продуктов и вычисляет скидку.

`PriceEngine` умножает количество продуктов на 1% и ограничивает максимальную скидку 20 процентами. Больше этот класс не делает ничего. Он не добавляет продукты во внутреннюю коллекцию и не сохраняет их в базе данных. Единственным результатом метода `CalculateDiscount()` является возвращаемая им скидка — выходное значение (рис. 6.2).

Листинг 6.1. Тестирование выходных данных

```
public class PriceEngine
{
    public decimal CalculateDiscount(params Product[] products)
    {
        decimal discount = products.Length * 0.01m;
        return Math.Min(discount, 0.2m);
    }
}

[Fact]
public void Discount_of_two_products()
{
    var product1 = new Product("Hand wash");
    var product2 = new Product("Shampoo");
    var sut = new PriceEngine();

    decimal discount = sut.CalculateDiscount(product1, product2);

    Assert.Equal(0.02m, discount);
}
```

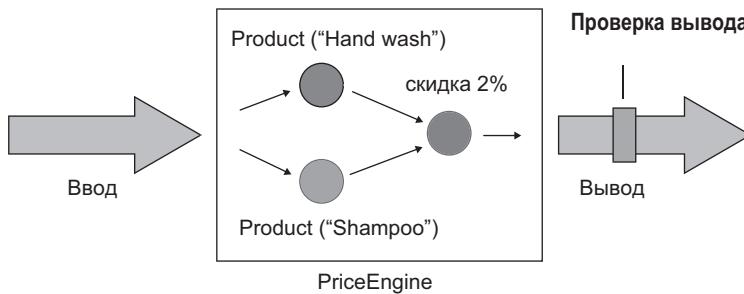


Рис. 6.2. Представление `PriceEngine` в системе записи «ввод-вывод». Метод `CalculateDiscount()` получает массив продуктов и вычисляет размер скидки

Проверка выходных данных также называется *функциональным* стилем юнит-тестирования. Название происходит из функционального программирования — методологии программирования, отдающей предпочтение коду без побочных эффектов. Функциональное программирование и функциональная архитектура более подробно рассматриваются далее в этой главе.

6.1.2. Проверка состояния

Проверка состояния — стиль юнит-тестирования, при котором тест проверяет состояние системы после завершения операции (рис. 6.3). Термин «состояние» в этом стиле тестирования может означать состояние самой тестируемой системы, одного из ее коллегораторов или внепроцессной зависимости — например, базы данных или файловой системы.

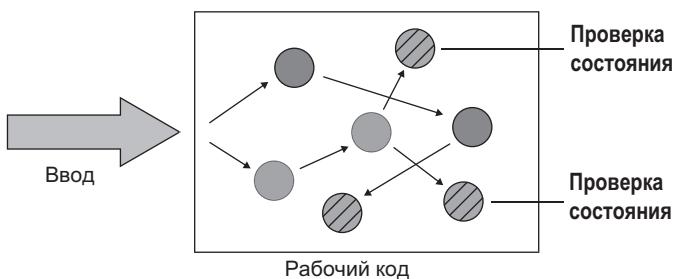


Рис. 6.3. При тестировании состояния тесты проверяют итоговое состояние системы после завершения операции. Заштрихованные круги представляют это итоговое состояние

В листинге 6.2 приведен пример тестирования состояния. Класс Order позволяет клиенту добавить новый товар.

Листинг 6.2. Тестирование состояния

```
public class Order
{
    private readonly List<Product> _products = new List<Product>();
    public IReadOnlyList<Product> Products => _products.ToList();

    public void AddProduct(Product product)
    {
        _products.Add(product);
    }
}

[Fact]
public void Adding_a_product_to_an_order()
{
    var product = new Product("Hand wash");
    var sut = new Order();

    sut.AddProduct(product);

    Assert.Equal(1, sut.Products.Count);
    Assert.Equal(product, sut.Products[0]);
}
```

Тест проверяет коллекцию Products после завершения операции добавления. В отличие от примера с проверкой выходных данных из листинга 6.1, результатом AddProduct() является изменение состояния заказа.

6.1.3. Проверка взаимодействий

Наконец, третьим стилем юнит-тестирования является проверка взаимодействий. Этот стиль использует моки для проверки взаимодействий между тестируемой системой и ее коллегами (рис. 6.4).

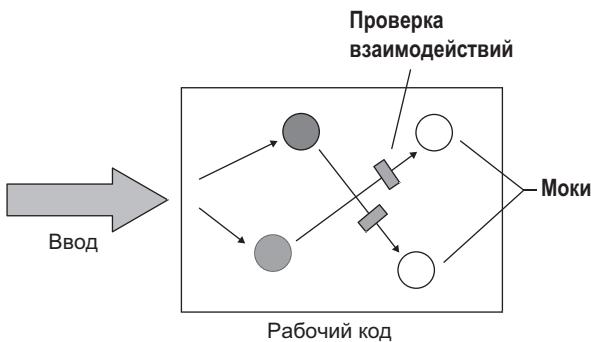


Рис. 6.4. При проверке взаимодействий тесты заменяют коллегов тестируемой системы моками для проверки того, что тестируемая система правильно вызывает этих коллег

В листинге 6.3 приведен пример тестирования взаимодействий.

Листинг 6.3. Тестирование взаимодействий

```
[Fact]
public void Sending_a_greetings_email()
{
    var emailGatewayMock = new Mock<IEmailGateway>();
    var sut = new Controller(emailGatewayMock.Object);

    sut.GreetUser("user@email.com");

    emailGatewayMock.Verify(
        x => x.SendGreetingsEmail("user@email.com"),
        Times.Once);
}
```

СТИЛИ И ШКОЛЫ ЮНИТ-ТЕСТИРОВАНИЯ

Классическая школа юнит-тестирования предпочитает проверку состояния, а не проверку взаимодействий. Лондонская школа делает противоположный выбор. Обе школы используют проверку выходных данных.

6.2. Сравнение трех стилей юнит-тестирования

В этих трех стилях юнит-тестирования нет ничего нового. Собственно, все эти стили уже были продемонстрированы ранее в книге. Интересно будет сравнить их друг с другом по четырем атрибутам хорошего юнит-теста. Еще раз приведу эти атрибуты (подробности см. в главе 4):

- защита от багов;
- устойчивость к рефакторингу;

- быстрая обратная связь;
- простота поддержки.

Рассмотрим все четыре атрибута по отдельности.

6.2.1. Сравнение стилей по метрикам защиты от багов и быстроте обратной связи

Сравним три стиля по атрибутам защиты от багов и быстроте обратной связи, так как эти атрибуты являются самыми простыми в этом конкретном сравнении. Метрика защиты от багов не зависит от конкретного стиля тестирования. Эта метрика является производной от следующих трех характеристик:

- объем кода, выполняемого в ходе теста;
- сложность этого кода;
- важность этого кода с точки зрения бизнес-логики.

Вы можете писать тесты, которые выполняют произвольный объем кода, никакой конкретный стиль не имеет преимуществ в этой области. То же относится к сложности кода и важности предметной области. Единственным исключением является проверка взаимодействий: злоупотребления могут привести к появлению поверхностных тестов, которые проверяют тонкий срез кода и заменяют все остальное моками. Но эта поверхность не является определяющей особенностью тестирования взаимодействий, это скорее крайний случай злоупотребления этим стилем тестирования.

Между стилями тестирования и быстротой обратной связи тоже корреляции нет. При условии что ваши тесты не притрагиваются к внепроцессным зависимостям и, следовательно, остаются в области юнит-тестирования (а не интеграционного тестирования), все стили приводят к тестам с приблизительно одинаковой скоростью выполнения. Проверка взаимодействий может показывать чуть худшие результаты, потому что моки обычно создают дополнительную задержку во время выполнения. Впрочем, различия обычно пренебрежимо малы, если только количество тестов не измеряется десятками тысяч.

6.2.2. Сравнение стилей по метрике устойчивости к рефакторингу

Что касается метрики устойчивости к рефакторингу, ситуация выглядит иначе. Устойчивость к рефакторингу показывает количество ложных срабатываний (ложных сигналов тревоги), выдаваемых тестами при рефакторинге рабочего кода. Ложные срабатывания, в свою очередь, являются результатом привязки тестов к деталям имплементации вместо наблюдаемого поведения.

Проверка выходных данных обеспечивает наилучшую защиту от ложных срабатываний, потому что получаемые тесты завязываются только на тестируемый метод. Они могут быть завязаны на детали имплементации только в одном случае: если сам тестируемый метод является деталью имплементации.

Проверка состояния обычно в большей степени подвержена ложным срабатываниям. Кроме тестируемого метода, такие тесты также работают с состоянием класса. С вероятностной точки зрения, чем больше связей между тестом и проверяемым им кодом, тем больше вероятность того, что этот тест окажется завязанным на детали имплементации, раскрытие которых в результате неверно выстроенного API. Тесты, проверяющие состояние, связываются с большей поверхностью API — следовательно, и вероятность завязывания их на детали имплементации тоже выше.

Проверка взаимодействий в наибольшей степени подвержена ложным срабатываниям. Как вы, вероятно, помните из главы 5, многие тесты, проверяющие взаимодействия с тестовыми заглушками, в конечном итоге оказываются хрупкими. Это всегда происходит в тестах, проверяющих взаимодействия со стабами, — никогда не проверяйте такие взаимодействия. Моки хороши только тогда, когда они проверяют взаимодействия, выходящие за границу приложения, и только когда результат этих взаимодействий виден внешнему миру. Как видите, проверка взаимодействий требует особой осмотрительности для сохранения устойчивости к рефакторингу.

И все же хрупкость, как и поверхность, не является определяющей особенностью стиля на основании проверок взаимодействий. Количество ложных срабатываний можно свести к минимуму, поддерживая правильную инкапсуляцию и связывая тесты только с наблюдаемым поведением. Однако следует признать, что объем усилий, требуемых для поддержания устойчивости к рефакторингу, варьируется в зависимости от стиля юнит-тестирования.

6.2.3. Сравнение стилей по метрике простоты поддержки

Наконец, метрика простоты поддержки сильно зависит от стиля юнит-тестирования, но, в отличие от устойчивости к рефакторингу, вы мало что можете с этим сделать. Напомню, что простота поддержки оценивает затраты на сопровождение юнит-тестов и определяется следующими двумя характеристиками:

- насколько трудно понять тест (зависит от размера теста);
- насколько трудно запустить этот тест (зависит от количества внепроцессных зависимостей, с которыми работает тест).

Большие тесты сложнее поддерживать, потому что их труднее понимать и изменять. Аналогичным образом тест, который работает напрямую с одной или несколькими внепроцессными зависимостями (например, базой данных), создает больше проблем с сопровождением, потому что вам приходится тратить время на поддержание этих

внепроцессных зависимостей в работоспособном состоянии: перезапускать сервер базы данных, решать проблемы с сетевым подключением и т. д.

Сопровождаемость тестов, проверяющих выходные данные

По сравнению с двумя другими стилями проверка выходных данных создает меньше всего проблем с поддержкой. Тесты почти всегда получаются короткими и лаконичными, и, как следствие, более простыми в сопровождении. Преимущество стиля на основании проверки выходных данных объясняется тем фактом, что этот стиль сводится всего к двум операциям: передаче входных параметров методу и проверке его выходного значения, для чего зачастую достаточно пары строк кода.

Так как проверяемый код при тестировании выходных данных не изменяет глобального или внутреннего состояния, такие тесты не работают с внепроцессными зависимостями. А следовательно, тестирование выходных данных оптимально с точки зрения обеих характеристик сопровождаемости.

Сопровождаемость тестов, проверяющих состояние

Тесты, проверяющие состояние, обычно обладают худшей сопровождаемостью, чем тесты, проверяющие выходные данные. Дело в том, что проверка состояния часто занимает больше места, чем проверка выходных данных. Приведу еще один пример тестирования состояния.

Листинг 6.4. Проверка состояния, занимающая много места

```
[Fact]
public void Adding_a_comment_to_an_article()
{
    var sut = new Article();
    var text = "Comment text";
    var author = "John Doe";
    var now = new DateTime(2019, 4, 1);

    sut.AddComment(text, author, now);

    Assert.Equal(1, sut.Comments.Count);
    Assert.Equal(text, sut.Comments[0].Text);
    Assert.Equal(author, sut.Comments[0].Author);
    Assert.Equal(now, sut.Comments[0].DateCreated);
}
```

Проверяет
состояние
статьи

Этот тест добавляет комментарий к статье, а затем проверяет, появился ли он в списке комментариев. Хотя этот тест предельно упрощен и содержит всего один комментарий, проверки в нем все равно занимают четыре строки. Тестам, проверяющим состояние, часто приходится проверять намного больше данных, вследствие чего они могут значительно увеличиваться в размерах.

Проблему можно частично решить введением вспомогательных методов, которые скрывают большую часть кода и таким образом сокращают тест (листинг 6.5), но написание и сопровождение таких методов требует значительных усилий. Эти усилия оправданы только в том случае, если методы будут переиспользоваться в разных тестах, что встречается довольно редко. Вспомогательные методы будут более подробно рассмотрены в части III.

Листинг 6.5. Использование вспомогательных методов в тестовых проверках

```
[Fact]
public void Adding_a_comment_to_an_article()
{
    var sut = new Article();
    var text = "Comment text";
    var author = "John Doe";
    var now = new DateTime(2019, 4, 1);

    sut.AddComment(text, author, now);

    sut.ShouldContainNumberOfComments(1)      | Вспомогательные
        .WithComment(text, author, now);      | методы
}
```

Другой способ сокращения размера тестов, проверяющих состояние, — добавить методы проверки равенства в проверяемый класс. В листинге 6.6 это класс `Comment`. Его можно преобразовать в объект-значение (класс, экземпляры которого сравниваются по значению, а не по ссылке), как показано ниже. Это также приведет к упрощению теста, особенно в сочетании с библиотекой Fluent Assertions.

Листинг 6.6. Сравнение Comment по значению

```
[Fact]
public void Adding_a_comment_to_an_article()
{
    var sut = new Article();
    var comment = new Comment(
        "Comment text",
        "John Doe",
        new DateTime(2019, 4, 1));

    sut.AddComment(comment.Text, comment.Author, comment.DateCreated);

    sut.Comments.Should().BeEquivalentTo(comment);
}
```

Тест использует тот факт, что комментарии могут сравниваться как целые значения, без необходимости проверки отдельных свойств. Также в teste используется метод `BeEquivalentTo` из Fluent Assertions, который может сравнивать целые коллекции, избавляя вас от необходимости проверять размер коллекции.

Это весьма эффективный прием, но он работает только в том случае, если класс представляет собой значение и может быть преобразован в объект-значение (value object). В противном случае он приводит к загрязнению кода (рабочая кодовая база загрязняется кодом, единственной целью которого является проведение или, как в данном случае, упрощение юнит-тестирования). Загрязнение кода рассматривается вместе с другими антипаттернами юнит-тестирования в главе 11.

Как видите, эти два приема — использование вспомогательных методов и преобразование классов в объекты-значения — могут применяться не всегда, а только в отдельных случаях. Причем даже если они применимы, тесты, проверяющие состояние, все равно занимают больше места, чем тесты, проверяющие выходные данные, что затрудняет сопровождение.

Сопровождаемость тестов, проверяющих взаимодействия

По метрике сопровождаемости тесты, проверяющие взаимодействия, уступают тестам, проверяющим выходные данные, и тестам, проверяющим состояние. Тестирование взаимодействий требует настройки тестовых заглушек и проверки взаимодействий с ними, а все это занимает много места. Тесты становятся еще больше и создают дополнительные трудности с сопровождением при наличии цепочек моков (моки или стабы, возвращающие другие моки, которые также возвращают моки, и т. д. на несколько уровней в глубину).

6.2.4. Сравнение стилей: результаты

Давайте теперь сравним стили юнит-тестирования по атрибутам хороших юнит-тестов. Сводка результатов представлена в таблице 6.1. Как обсуждалось в разделе 6.2.1, все три стиля показывают одинаковые результаты по метрикам защиты от багов и быстроте обратной связи, поэтому я исключил эти метрики из сравнения.

Таблица 6.1. Три стиля юнит-тестирования: сравнение

	Проверка выходных данных	Проверка состояния	Проверка взаимодействий
Особое внимание для сохранения устойчивости к рефакторингу	Низкое	Среднее	Среднее
Затраты на сопровождение	Низкие	Средние	Высокие

Тестирование выходных данных показывает наилучшие результаты. Этот стиль приводит к тестам, которые редко завязываются на детали имплементации и, как следствие, не требуют особого внимания для сохранения устойчивости к рефакторингу. Такие тесты также создают меньше всего проблем с сопровождением благодаря своей компактности и отсутствию внепроцессных зависимостей.

Тесты, проверяющие состояние и взаимодействия, уступают по обеим метрикам. Они с большей вероятностью окажутся завязаны на детали имплементации, а также становятся причиной повышенных затрат на сопровождение из-за увеличения их размера.

Всегда отдавайте предпочтение тестированию выходных данных. К сожалению, это проще сказать, чем сделать. Этот стиль юнит-тестирования применим только к коду, написанному в функциональном стиле, а такой код редко встречается в большинстве языков объектно-ориентированного программирования. Тем не менее существуют приемы, которыми вы можете воспользоваться, для того чтобы преобразовать ваши тесты так, чтобы они проверяли выходные данные вместо состояния или взаимодействий.

В оставшейся части этой главы показано, как выполняется переход от тестирования состояния и взаимодействий к тестированию выходных данных. Для такого преобразования необходимо переработать ваш код в функциональный стиль.

6.3. Функциональная архитектура

Прежде чем я смогу показать, как выполняется такое преобразование, понадобится небольшое вступление. В этом разделе вы узнаете, что такое функциональное программирование и функциональная архитектура и как последняя связана с гексагональной архитектурой. В разделе 6.4 такое преобразование продемонстрировано на примере.

Стоит отметить, что этот раздел не является глубоким анализом функционального программирования, а только объясняет его базовые принципы. Этих базовых принципов должно быть достаточно, для того чтобы понять связь между функциональным программированием и тестированием выходных данных. За более глубоким изложением темы функционального программирования обращайтесь на сайт Скотта Влашина (Scott Wlaschin) и к его книгам (<https://fsharpforfunandprofit.com/books>).

6.3.1. Что такое функциональное программирование?

Как упоминалось в разделе 6.1.1, стиль юнит-тестирования на основании проверки выходных данных также называется *функциональным*. Это объясняется тем, что тестируемый рабочий код должен быть написан в функциональном стиле с использованием функционального программирования. Что же это такое – функциональное программирование?

Функциональным программированием называется программирование, основанное на использовании математических функций. *Математическая функция* (также называемая чистой (pure) функцией) – это функция (или метод), не имеющая скрытых входов или выходов. Все входы или выходы математической функции должны быть

явно выражены в сигнатуре ее метода, которая состоит из имени, аргументов и возвращаемого типа. Математическая функция всегда выдает один и тот же результат для заданного параметра, сколько бы раз эта функция ни вызывалась.

Для примера возьмем метод `CalculateDiscount()` из листинга 6.1:

```
public decimal CalculateDiscount(Product[] products)
{
    decimal discount = products.Length * 0.01m;
    return Math.Min(discount, 0.2m);
}
```

Этот метод имеет одно входное значение (массив объектов класса `Product`) и одно выходное значение (скидка в формате `decimal`). Оба значения явно выражены в сигнатуре метода; метод не имеет скрытых входных или выходных значений. Эти обстоятельства делают `CalculateDiscount()` математической функцией (рис. 6.5).



Рис. 6.5. Метод `CalculateDiscount()` имеет одно входное значение (массив продуктов) и одно выходное значение (скидка). И входное, и выходное значение явно выражены в сигнатуре метода, что делает `CalculateDiscount()` математической функцией

Методы, не имеющие скрытых входных или выходных данных, называются математическими функциями, потому что такие методы соответствуют математическому определению функции.

ОПРЕДЕЛЕНИЕ

В математике *функцией* называется такое отображение между двумя множествами, при котором каждому элементу первого множества соответствует ровно один элемент второго множества.

На рис. 6.6 показано, как для каждого входного значения x функция $f(x) = x + 1$ находит соответствующее значение y . На рис. 6.7 представлен метод `CalculateDiscount()` с использованием тех же обозначений, что и на рис. 6.6.

Явно выраженные входные и выходные значения сильно упрощают тестирование математических функций, потому что тесты получаются короткими, простыми и легкими для понимания и сопровождения. Математические функции — единственная разновидность методов, к которым может применяться тестирование на основании проверки выходных данных. Такие тесты обладают наилучшей сопровождаемостью и наименьшей вероятностью выдать ложные срабатывания.

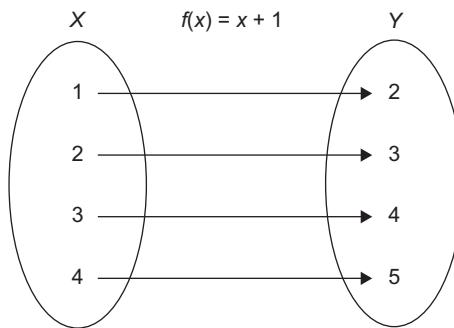


Рис. 6.6. Типичный пример математической функции — $f(x) = x + 1$. Для каждого входного значения x из множества X функция находит соответствующее число y из множества Y

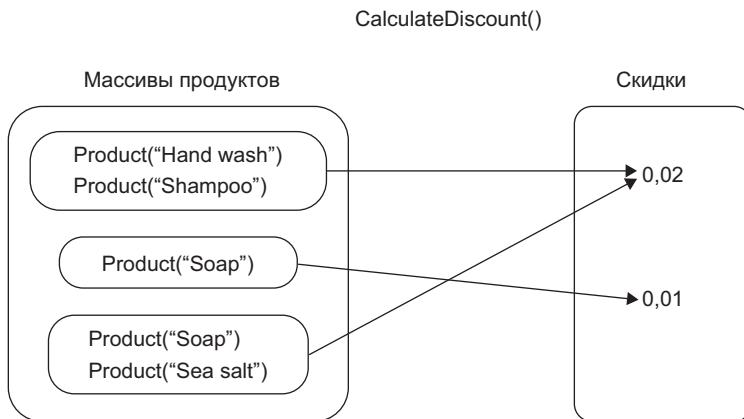


Рис. 6.7. Метод `CalculateDiscount()`, представленный в тех же обозначениях, что и функция $f(x) = x + 1$. Для каждого входного массива продуктов метод находит соответствующую скидку как выходное значение

С другой стороны, скрытые входы и выходы усложняют тестирование кода (и его понимание). Некоторые разновидности таких скрытых входов и выходов:

- *Побочные эффекты (side effects)* — результаты, которые не показываются в сигнатуре метода (а следовательно, являются скрытыми). Операция создает побочный эффект при изменении состояния экземпляра класса, обновлении файла на диске и т. д.
- *Исключения* — когда метод выдает исключение, он создает путь в последовательности выполнения программы, который обходит контракт, установленный сигнатурой метода. Выданное исключение может быть перехвачено в любом месте стека вызовов, создавая тем самым дополнительный результат, не показанный в сигнатуре метода.

- *Ссылка на внутреннее или внешнее состояние* — например, метод может получить текущую дату и время при помощи статического свойства `DateTime.Now`. Метод может также запросить информацию из базы данных или обратиться к приватному изменяемому полю. Все это — входные данные, не включенные в сигнатуру метода, а следовательно, они тоже являются скрытыми.

Существует хороший способ определения того, является ли метод математической функцией: попробуйте заменить вызов метода его возвращаемым значением и посмотрите, приведет ли это к изменению поведения программы. Возможность замены вызова метода соответствующим значением называется *ссыпочной прозрачностью* (*referential transparency*). Для примера рассмотрим следующий метод:

```
public int Increment(int x)
{
    return x + 1;
}
```

Этот метод является математической функцией. Следующие две команды эквивалентны:

```
int y = Increment(4);
int y = 5;
```

С другой стороны, следующий метод математической функцией не является. Его невозможно заменить возвращаемым значением, потому что возвращаемое значение не представляет все результаты выполнения метода. В данном случае скрытым результатом является изменение поля `x` (побочный эффект, *side effect*):

```
int x = 0;
public int Increment()
{
    x++;
    return x;
}
```

Побочные эффекты составляют наиболее многочисленную категорию скрытых результатов. В листинге 6.7 показан метод `AddComment`, который на первый взгляд похож на математическую функцию, но на самом деле ею не является. На рис. 6.8 этот метод представлен в графическом виде.

Листинг 6.7. Изменение внутреннего состояния

```
public Comment AddComment(string text)
{
    var comment = new Comment(text);
    _comments.Add(comment); ← Побочный эффект (side effect)
    return comment;
}
```

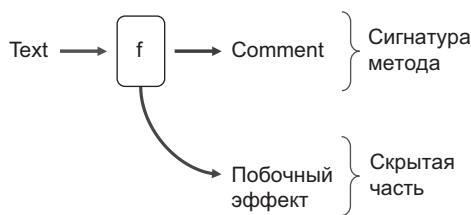


Рис. 6.8. Метод AddComment (представленный как *f*) получает входное значение *Text* и возвращает значение *Comment*. Оба значения представлены в сигнатуре метода. Побочным эффектом является дополнительный скрытый результат

6.3.2. Что такое функциональная архитектура?

Конечно, создать приложение, не имеющее вообще никаких побочных эффектов, невозможно. Такое приложение будет бесполезным. В конце концов, все приложения создаются именно ради побочных эффектов: обновления информации пользователя, включения новой позиции заказа в покупательскую корзину и т. д.

Целью функционального программирования является не полное исключение побочных эффектов, а отделение кода, работающего с бизнес-логикой, от кода, создающего побочные эффекты. Эти две обязанности достаточно сложны сами по себе, а их смешение многократно увеличивает сложность и вредит сопровождаемости кода в долгосрочной перспективе. На помощь приходит функциональная архитектура. Она отделяет бизнес-логику от побочных эффектов, выводя побочные эффекты за границу бизнес-операций.

ОПРЕДЕЛЕНИЕ

Функциональная архитектура максимизирует объем кода, написанного в чисто функциональном (*неизменяемом*) стиле, и сводит к минимуму объем кода, работающий с побочными эффектами. Под «*неизменяемостью*» имеется в виду, что после создания объекта его состояние изменяться не может — в отличие от *изменяемых* объектов, которые могут изменяться после создания.

Разделение бизнес-логики и побочных эффектов осуществляется посредством изоляции двух типов кода:

- код, принимающий решение. Этот код не требует побочных эффектов, а следовательно, может записываться с использованием математических функций;
- код, действующий по результатам этого решения. Этот код преобразует все решения, принятые математическими функциями, в видимые артефакты — например, изменения в базе данных или сообщения, отправленные по шине.

Код, принимающий решения, часто называется *функциональным ядром* (*functional core*) (также известен как *неизменяемое ядро*, *immutable core*). Код, действующий по результатам этих решений, называется *изменяемой оболочкой* (*mutable shell*) (рис. 6.9).



Рис. 6.9. В функциональной архитектуре функциональное ядро реализуется с использованием математических функций и принимает все решения в приложении. Изменяемая оболочка предоставляет функциональному ядру входные данные и интерпретирует его решения, применяя побочные эффекты к внепроцессным зависимостям (например, базе данных)

ИНКАПСУЛЯЦИЯ И НЕИЗМЕНЯЕМОСТЬ

Как и инкапсуляция, функциональная архитектура вообще и неизменяемость в частности служат той же цели, что и юнит-тестирование — обеспечению стабильного роста программного проекта. Более того, между концепциями инкапсуляции и неизменяемости существует глубокая связь.

Как говорилось в главе 5, *инкапсуляцией* называется защита вашего кода от нарушений логической целостности. Инкапсуляция защищает внутреннее устройство класса от повреждения за счет:

- ✓ сокращения площади API, допускающей модификацию данных;
- ✓ внимательной проверки оставшихся частей API.

Неизменяемость подходит к проблеме сохранения инвариантов с другого угла. С неизменяемыми классами не нужно беспокоиться о возможном повреждении состояния, потому что невозможно повредить то, что изменяться вообще не может. Как следствие, в функциональном программировании отпадает необходимость в инкапсуляции. Достаточно проверить состояние класса всего один раз при создании его экземпляра. После этого экземпляр можно свободно передавать в коде. Если все данные неизменяемы, то проблемы, связанные с недостаточной инкапсуляцией, попросту исчезают.

У Майкла Фезерса (Michael Feathers) есть замечательное высказывание по этому поводу: «Объектно-ориентированное программирование делает код более понятным за счет инкапсуляции подвижных частей. Функциональное программирование делает код более понятным за счет минимизации количества подвижных частей».

Функциональное ядро и изменяемая оболочка взаимодействуют следующим образом:

- изменяемая оболочка собирает все входные данные;
- функциональное ядро генерирует решения;
- оболочка преобразует решения в побочные эффекты.

Чтобы поддерживать правильное разделение между этими двумя слоями, необходимо проследить за тем, чтобы классы, представляющие решения, содержали достаточно информации, чтобы изменяемая оболочка могла действовать без принятия дополнительных решений. Иными словами, изменяемая оболочка должна быть настолько «глупой», насколько возможно. Такая конфигурация позволяет вам покрыть функциональное ядро тестами, проверяющими выходные данные, тогда как на долю изменяемой оболочки остается намного меньшее количество интеграционных тестов.

6.3.3. Сравнение функциональных и гексагональных архитектур

Между функциональными и гексагональными архитектурами существует сходство. Обе архитектуры строятся на базе идеи разделения обязанностей. Различаются подробности такого разделения.

Как говорилось в главе 5, в гексагональной архитектуре различается слой предметной области и слой сервисов приложения (рис. 6.10). Слой предметной области отвечает за бизнес-логику, тогда как слой сервисов приложения отвечает за взаимодействие



Рис. 6.10. Гексагональная архитектура представляет собой набор взаимодействующих приложений, изображенных в виде гексагонов (шестиугольников). Приложение состоит из слоя предметной области и слоя сервисов приложения, которые соответствуют функциональному ядру и изменяемой оболочке в функциональной архитектуре

с внешними приложениями (например, базой данных или SMTP-сервисом). Такая архитектура очень похожа на функциональную архитектуру с ее разделением решений и действий.

Другое сходство — однонаправленный характер зависимостей. В гексагональной архитектуре классы, находящиеся внутри слоя предметной области, должны зависеть только друг от друга; они не должны зависеть от классов слоя сервисов приложения. Аналогичным образом неизменяемое ядро в функциональной архитектуре не должно зависеть от изменяемой оболочки. Оно самодостаточно и может работать в изоляции от внешних слоев. Именно это обстоятельство делает функциональную архитектуру такой удобной в тестировании: вы можете полностью отделить неизменяемое ядро от изменяемой оболочки и имитировать входные данные, предоставляемые оболочкой, простыми значениями.

Различия между архитектурами проявляются в отношении к побочным эффектам. Функциональная архитектура перемещает все побочные эффекты из неизменяемого ядра к границам бизнес-операций. Этими границами занимается изменяемая оболочка. С другой стороны, гексагональная архитектура не выступает против побочных эффектов, производимых слоем предметной области, при условии что они ограничиваются только этим слоем предметной области. Все модификации в гексагональной архитектуре должны содержаться в слое предметной области, не пересекая границы этого слоя. Например, экземпляр класса предметной области не может ничего сохранять в базе данных напрямую, но может изменять собственное состояние. Сервис приложения берет это изменение и применяет его к базе данных.

ПРИМЕЧАНИЕ

Функциональная архитектура является подмножеством гексагональной архитектуры. Функциональную архитектуру можно рассматривать как гексагональную архитектуру,озведенную в абсолют.

6.4. Переход на функциональную архитектуру и тестирование выходных данных

В этом разделе мы возьмем приложение-пример и отрефакторим его в направлении функциональной архитектуры. Будут представлены две стадии рефакторинга:

- переход от внепроцессной зависимости к использованию мока;
- переход от мока к функциональной архитектуре.

Переход повлияет и на код тестов. Мы выполним рефакторинг тестов от проверки состояния и взаимодействий к проверке выходных данных. Прежде чем браться за рефакторинг, кратко рассмотрим проект и покрывающие его тесты.

6.4.1. Система аудита

В нашем примере рассматривается система аудита, которая отслеживает всех посетителей в организации. Для хранения данных используются текстовые файлы со структурой, показанной на рис. 6.11. Система добавляет имя посетителя и время посещения в конец текущего файла. Когда будет достигнуто максимальное количество записей в файле, создается новый файл с увеличенным индексом.

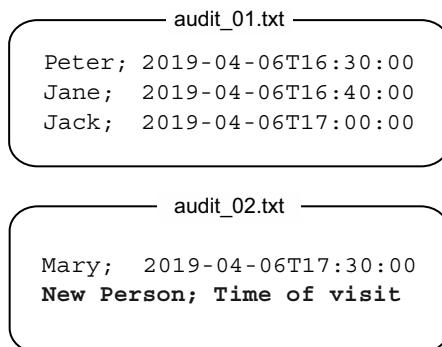


Рис. 6.11. Система аудита хранит информацию о посетителях в текстовых файлах конкретного формата.
При достижении максимального количества записей на файл система создает новый файл

В листинге 6.8 приведен код исходной версии системы.

Листинг 6.8. Исходная версия системы аудита

```
public class AuditManager
{
    private readonly int _maxEntriesPerFile;
    private readonly string _directoryName;

    public AuditManager(int maxEntriesPerFile, string directoryName)
    {
        _maxEntriesPerFile = maxEntriesPerFile;
        _directoryName = directoryName;
    }

    public void AddRecord(string visitorName, DateTime timeOfVisit)
    {
        string[] filePaths = Directory.GetFiles(_directoryName);
        (int index, string path)[] sorted = SortByIndex(filePaths);

        string newRecord = visitorName + ';' + timeOfVisit;

        if (sorted.Length == 0)
        {
            string newFile = Path.Combine(_directoryName, "audit_1.txt");
            File.WriteAllText(newFile, newRecord);
        }
        else
        {
            string currentFile = Path.Combine(_directoryName, "audit_" + index + ".txt");
            File.AppendAllText(currentFile, newRecord);
        }
    }
}
```

```
        return;
    }

    (int currentFileIndex, string currentFilePath) = sorted.Last();
    List<string> lines = File.ReadAllLines(currentFilePath).ToList();

    if (lines.Count < _maxEntriesPerFile)
    {
        lines.Add(newRecord);
        string newContent = string.Join("\r\n", lines);
        File.WriteAllText(currentFilePath, newContent);
    }
    else
    {
        int newIndex = currentFileIndex + 1;
        string newName = $"audit_{newIndex}.txt";
        string newFile = Path.Combine(_directoryName, newName);
        File.WriteAllText(newFile, newRecord);
    }
}
```

Код может показаться слишком длинным, но он довольно прост. `AuditManager` — главный класс приложения. Его конструктор получает максимальное количество записей в файле и путь к рабочей директории. Единственный публичный метод класса `AddRecord` выполняет всю работу системы аудита:

- получает полный список файлов из рабочей директории;
 - сортирует их по индексу (все имена файлов строятся по одной схеме: `audit_{индекс}.txt` [например, `audit_1.txt`]);
 - если ни один файл еще не существует, создает первый файл с одной записью;
 - если файлы уже существуют, читает последний файл и либо добавляет к нему новую запись, либо создает новый файл (в зависимости от того, достигло ли количество записей в файле порогового значения).

В таком виде класс `AuditManager` тестировать сложно, потому что он тесно связан с файловой системой. Перед тестированием необходимо разместить файлы в правильном каталоге, а после завершения тестов необходимо считать эти файлы, проверить их содержимое и удалить их (рис. 6.12).

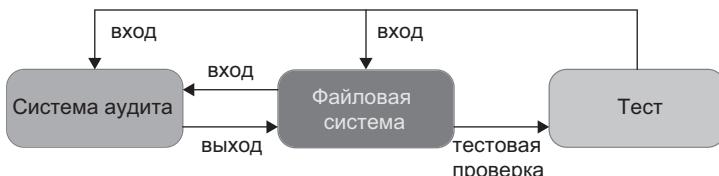


Рис. 6.12. Тесты, покрывающие исходную версию системы аудита, должны работать напрямую с файловой системой

Организовать параллельное выполнение таких тестов не получится — по крайней мере, без дополнительных усилий, которые значительно повысят затраты на сопровождение. Узким местом здесь является файловая система: это совместная зависимость, через которую тесты могут вмешаться в логику выполнения друг друга.

Файловая система также замедляет выполнение тестов. Страдает и сопровождаемость, потому что вам придется следить за тем, чтобы рабочий каталог существовал и был доступен для тестов — как на вашей локальной машине, так и на сервере сборки. Краткая сводка по всем четырем атрибутам хороших тестов приведена в таблице 6.2.

Таблица 6.2. Исходная версия системы аудита показывает плохие результаты по двум из четырех атрибутов хорошего теста

	Исходная версия
Защита от багов	Хорошо
Устойчивость к рефакторингу	Хорошо
Быстрая обратная связь	Плохо
Простота поддержки	Плохо

Кстати говоря, тесты, напрямую работающие с файловой системой, не подходят под определение юнит-теста. Они не соответствуют второму и третьему атрибутам юнит-теста и таким образом относятся к категории интеграционных тестов (за подробностями обращайтесь к главе 2):

- тест проверяет одну единицу поведения;
- делает это быстро
- и в изоляции от других тестов.

6.4.2. Использование моков для отделения тестов от файловой системы

Типичное решение проблемы сильной связности тестов — создание мока для файловой системы. Все операции с файлами выделяются в отдельный тип (`IFileSystem`), который внедряется в `AuditManager` через конструктор. Затем тесты заменяют моком этот тип и перехватывают обращения записи от системы аудита к файлам (рис. 6.13).

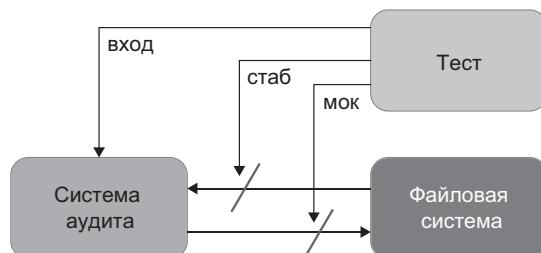


Рис. 6.13. Тесты могут заменить файловую систему моком и перехватить обращения записи, совершенные системой аудита к файлам

В листинге 6.9 показано, как файловая система внедряется в AuditManager.

Листинг 6.9. Явное внедрение файловой системы через конструктор

```
public class AuditManager
{
    private readonly int _maxEntriesPerFile;
    private readonly string _directoryName;
    private readonly IFileSystem _fileSystem;

    public AuditManager(
        int maxEntriesPerFile,
        string directoryName,
        IFileSystem fileSystem)
    {
        _maxEntriesPerFile = maxEntriesPerFile;
        _directoryName = directoryName;
        _fileSystem = fileSystem;
    }
}
```



Новый интерфейс представляет файловую систему

Затем идет метод AddRecord.

Листинг 6.10. Использование интерфейса IFileSystem

```
public void AddRecord(string visitorName, DateTime timeOfVisit)
{
    string[] filePaths = _fileSystem
        .GetFiles(_directoryName);
    (int index, string path)[] sorted = SortByIndex(filePaths);

    string newRecord = visitorName + ';' + timeOfVisit;

    if (sorted.Length == 0)
    {
        string newFile = Path.Combine(_directoryName, "audit_1.txt");
        _fileSystem.WriteAllText(
            newFile, newRecord);
        return;
    }

    (int currentFileIndex, string currentFilePath) = sorted.Last();
    List<string> lines = _fileSystem
        .ReadAllLines(currentFilePath);

    if (lines.Count < _maxEntriesPerFile)
    {
        lines.Add(newRecord);
        string newContent = string.Join("\r\n", lines);
        _fileSystem.WriteAllText(
            currentFilePath, newContent);
    }
    else
    {
```



Использование нового интерфейса

Использование нового интерфейса

Использование нового интерфейса

```
    int newIndex = currentFileIndex + 1;
    string newName = $"audit_{newIndex}.txt";
    string newFile = Path.Combine(_directoryName, newName);
    _fileSystem.WriteAllText(
        newFile, newRecord);
}
}
```

В листинге 6.10 `IFileSystem` — новый интерфейс, инкапсулирующий работу с файловой системой:

```
public interface IFileSystem
{
    string[] GetFiles(string directoryName);
    void WriteAllText(string filePath, string content);
    List<string> ReadAllLines(string filePath);
}
```

Теперь, когда класс `AuditManager` отделен от файловой системы, совместная зависимость исчезла, и тесты могут выполняться независимо друг от друга. Один из таких тестов приведен в листинге 6.11.

Листинг 6.11. Проверка поведения системы аудита с использованием мока

```
[Fact]
public void A_new_file_is_created_when_the_current_file_overflows()
{
    var fileSystemMock = new Mock<IFileSystem>();
    fileSystemMock
        .Setup(x => x.GetFiles("audits"))
        .Returns(new string[]
    {
        @"audits\audit_1.txt",
        @"audits\audit_2.txt"
    });
    fileSystemMock
        .Setup(x => x.ReadAllLines(@"audits\audit_2.txt"))
        .Returns(new List<string>
    {
        "Peter; 2019-04-06T16:30:00",
        "Jane; 2019-04-06T16:40:00",
        "Jack; 2019-04-06T17:00:00"
    });
    var sut = new AuditManager(3, "audits", fileSystemMock.Object);

    sut.AddRecord("Alice", DateTime.Parse("2019-04-06T18:00:00"));

    fileSystemMock.Verify(x => x.WriteAllText(
        @"audits\audit_3.txt",
        "Alice;2019-04-06T18:00:00"));
}
```

Этот тест проверяет, что когда количество записей в текущем файле достигает лимита (3 в данном случае), создается новый файл с одной записью. Следует отметить, что в данном конкретном случае использование мока оправданно. Приложение создает файлы, видимые для конечных пользователей (предполагается, что пользователи могут прочитать файлы, используя другую программу, будь то специализированная программа или обычный потепад.exe). Следовательно, взаимодействия с файловой системой и результаты этих взаимодействий (то есть изменения в файлах) являются частью наблюдаемого поведения приложения. Как вы, возможно, помните из главы 5, это единственный оправданный сценарий использования для моков.

Эта альтернативная реализация лучше исходной. Так как тесты больше не обращаются к файловой системе, они работают быстрее. А поскольку вам не нужно следить за файловой системой, чтобы обеспечить правильную работу тестов, затраты на сопровождение тоже сокращаются. Защита от багов и устойчивость к рефакторингу также не пострадали. В таблице 6.3 представлены различия между двумя версиями.

Таблица 6.3. Сравнение версии с моком с исходной версией системы аудита

	Исходная версия	Версия с моками
Защита от багов	Хорошо	Хорошо
Устойчивость к рефакторингу	Хорошо	Хорошо
Быстрая обратная связь	Плохо	Хорошо
Простота поддержки	Плохо	Средне

Впрочем, это все еще не лучший результат. Тест в листинге 6.11 содержит сложную подготовку, что не идеально в отношении затрат на сопровождение. Библиотеки мокирования стараются вам помочь, но полученные тесты читаются не так хорошо, как тесты, полагающиеся на простую проверку ввода и вывода.

6.4.3. Рефакторинг для перехода на функциональную архитектуру

Вместо того чтобы скрывать побочные эффекты за интерфейсом и внедрять этот интерфейс в AuditManager, можно полностью вынести эти побочные эффекты из класса. В этом случае AuditManager будет отвечать только за принятие решения относительно того, что делать с файлами. Новый класс Persister действует на основании этого решения и применяет обновления к файловой системе (рис. 6.14).

Persister в этом сценарии действует как изменяемая оболочка, тогда как AuditManager становится функциональным (неизменяемым) ядром. В листинге 6.12 показан класс AuditManager после рефакторинга.

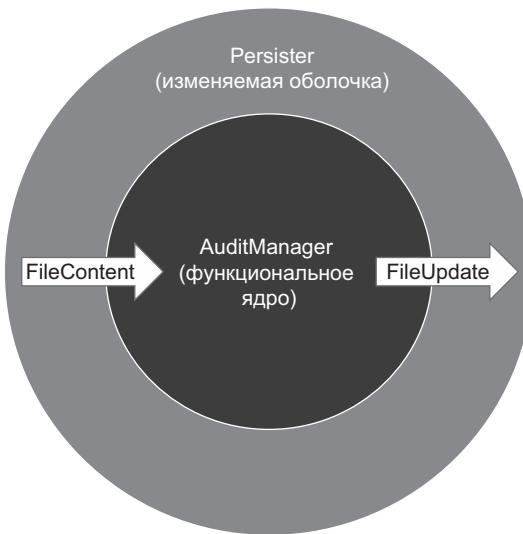


Рис. 6.14. Классы Persister и AuditManager образуют функциональную архитектуру.
Persister получает файлы и их содержимое из рабочего каталога, передает их AuditManager,
а затем преобразует возвращаемое значение в изменение в файловой системе

Листинг 6.12. Класс AuditManager после рефакторинга

```
public class AuditManager
{
    private readonly int _maxEntriesPerFile;

    public AuditManager(int maxEntriesPerFile)
    {
        _maxEntriesPerFile = maxEntriesPerFile;
    }

    public FileUpdate AddRecord(
        FileContent[] files,
        string visitorName,
        DateTime timeOfVisit)
    {
        (int index, FileContent file)[] sorted = SortByIndex(files);

        string newRecord = visitorName + ';' + timeOfVisit;

        if (sorted.Length == 0)
        {
            return new FileUpdate(
                "audit_1.txt", newRecord); | Возвращает инструкцию
                                         | на обновление
        }

        (int currentIndex, FileContent currentFile) = sorted.Last();
```

```

List<string> lines = currentFile.Lines.ToList();

if (lines.Count < _maxEntriesPerFile)
{
    lines.Add(newRecord);
    string newContent = string.Join("\r\n", lines);
    return new FileUpdate(
        currentFile.FileName, newContent);
}
else
{
    int newIndex = currentFileIndex + 1;
    string newName = $"audit_{newIndex}.txt";
    return new FileUpdate(
        newName, newRecord);
}
}
}

```

Возвращает инструкцию на обновление

Вместо пути к рабочему каталогу `AuditManager` теперь получает массив `FileContent`. Этот класс включает все, что необходимо знать `AuditManager` о файловой системе для принятия решения:

```

public class FileContent
{
    public readonly string FileName;
    public readonly string[] Lines;

    public FileContent(string fileName, string[] lines)
    {
        FileName = fileName;
        Lines = lines;
    }
}

```

Вместо того чтобы изменять файлы в рабочем каталоге, класс `AuditManager` теперь возвращает команду создания побочного эффекта, которую он хочет выполнить:

```

public class FileUpdate
{
    public readonly string FileName;
    public readonly string NewContent;

    public FileUpdate(string fileName, string newContent)
    {
        FileName = fileName;
        NewContent = newContent;
    }
}

```

В листинге 6.13 приведен класс `Persister`.

Листинг 6.13. Изменяемая оболочка, действующая по решению AuditManager

```
public class Persister
{
    public FileContent[] ReadDirectory(string directoryName)
    {
        return Directory
            .GetFiles(directoryName)
            .Select(x => new FileContent(
                Path.GetFileName(x),
                File.ReadAllLines(x)))
            .ToArray();
    }

    public void ApplyUpdate(string directoryName, FileUpdate update)
    {
        string filePath = Path.Combine(directoryName, update.FileName);
        File.WriteAllText(filePath, update.NewContent);
    }
}
```

Обратите внимание, насколько прост этот класс. Все, что он делает, — читает данные из рабочего каталога и применяет обновления, получаемые от `AuditManager`, к рабочему каталогу. В нем нет ветвления (команд `if`), вся сложность находится в классе `AuditManager`. Эти два класса — пример разделения бизнес-логики и побочных эффектов.

Чтобы поддерживать такое разделение, необходимо сделать интерфейсы классов `FileContent` и `FileUpdate` как можно более близкими к встроенным командам взаимодействия с файлами фреймворка. Весь разбор и подготовка должны выполняться в функциональном ядре, чтобы код за пределами ядра оставался тривиальным. Например, если бы в .NET не было встроенного метода `File.ReadAllText()`, который возвращает содержимое файла в виде массива строк, а был бы только метод `File.ReadAllText()`, возвращающий одну строку, то свойство `Lines` в `FileContent` нужно было бы заменить строкой и выполнять весь разбор в `AuditManager`:

```
public class FileContent
{
    public readonly string FileName;
    public readonly string Text; // в предыдущей версии: string[] Lines;
}
```

Для объединения `AuditManager` с `Persister` понадобится еще один класс — сервис приложения, приведенный в листинге 6.14.

Листинг 6.14. Объединение функционального ядра с изменяемой оболочкой

```
public class ApplicationService
{
    private readonly string _directoryName;
    private readonly AuditManager _auditManager;
```

```

private readonly Persister _persister;

public ApplicationService(
    string directoryName, int maxEntriesPerFile)
{
    _directoryName = directoryName;
    _auditManager = new AuditManager(maxEntriesPerFile);
    _persister = new Persister();
}

public void AddRecord(string visitorName, DateTime timeOfVisit)
{
    FileContent[] files = _persister.ReadDirectory(_directoryName);
    FileUpdate update = _auditManager.AddRecord(
        files, visitorName, timeOfVisit);
    _persister.ApplyUpdate(_directoryName, update);
}
}

```

Наряду с объединением функционального ядра с изменяемой оболочкой сервис приложения также предоставляет точку входа в систему для внешних клиентов (рис. 6.15). В этой реализации поведение системы аудита проверяется легко. Все тесты сводятся к передаче гипотетического состояния рабочего каталога и проверке решения, принятого классом `AuditManager`.

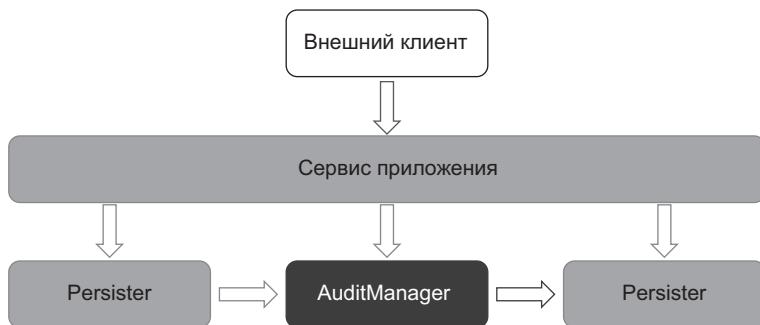


Рис. 6.15. ApplicationService объединяет функциональное ядро (`AuditManager`) с изменяемой оболочкой (`Persister`) и предоставляет точку входа для внешних клиентов. В терминах гексагональной архитектуры ApplicationService и Persister являются частью слоя сервисов приложения, тогда как AuditManager принадлежит модели предметной области

Листинг 6.15. Тест без моков

```

[Fact]
public void A_new_file_is_created_when_the_current_file_overflows()
{
    var sut = new AuditManager(3);
    var files = new FileContent[]
    {
        ...
    }
    ...
}

```

```

{
    new FileContent("audit_1.txt", new string[0]),
    new FileContent("audit_2.txt", new string[]
    {
        "Peter; 2019-04-06T16:30:00",
        "Jane; 2019-04-06T16:40:00",
        "Jack; 2019-04-06T17:00:00"
    })
};

FileUpdate update = sut.AddRecord(
    files, "Alice", DateTime.Parse("2019-04-06T18:00:00"));

Assert.Equal("audit_3.txt", update.FileName);
Assert.Equal("Alice;2019-04-06T18:00:00", update.NewContent);
}

```

Тест сохраняет улучшения версии с моками по сравнению с исходной версией (быстрая обратная связь), но также достигает дополнительных улучшений по метрике простоты поддержки. В сложной настройке моков больше нет необходимости — есть только простые входные и выходные значения, что существенно улучшает читаемость теста. В таблице 6.4 тест, проверяющий выходные данные, сравнивается с исходной версией и версией с моками.

Таблица 6.4. Сравнение теста с проверкой выходных данных с предыдущими двумя версиями

	Исходная версия	Версия с моками	Проверка выходных данных
Защита от багов	Хорошо	Хорошо	Хорошо
Устойчивость к рефакторингу	Хорошо	Хорошо	Хорошо
Быстрая обратная связь	Плохо	Хорошо	Хорошо
Простота поддержки	Плохо	Средне	Хорошо

Обратите внимание, что команды, генерируемые функциональным ядром, всегда представляют из себя значение или набор значений. Два экземпляра такого значения взаимозаменяемы при условии совпадения их содержимого. Вы можете воспользоваться этим фактом и еще больше улучшить читаемость теста, преобразуя `FileUpdate` в объект-значение. Чтобы сделать это в .NET, необходимо либо преобразовать класс в структуру, либо переопределить методы проверки равенства. Это позволит использовать сравнение по значению — в отличие от сравнения по ссылке, используемого по умолчанию для всех классов в C#. Сравнение по значению также позволяет объединить два тестовых утверждения из листинга 6.15 в одно:

```

Assert.Equal(
    new FileUpdate("audit_3.txt", "Alice;2019-04-06T18:00:00"),
    update);

```

Или с использованием Fluent Assertions:

```
update.Should().Be(
    new FileUpdate("audit_3.txt", "Alice;2019-04-06T18:00:00"));
```

6.4.4. Потенциальные будущие изменения

Давайте кратко рассмотрим другие изменения, которые могут быть реализованы в будущем в нашем проекте. Система аудита, которую я продемонстрировал, весьма проста — она содержит только три ветви:

- создание нового файла, если рабочий каталог пуст;
- присоединение новой записи к существующему файлу;
- создание нового файла при превышении лимита на количество записей в существующем файле.

Кроме того, задействован только один бизнес-сценарий: добавление новой записи в журнал аудита. Что, если появится другой сценарий (например, удаление всех упоминаний конкретного посетителя)? И что, если системе понадобится выполнять проверки (например, проверять максимальную длину имени посетителя)?

Удаление всех упоминаний конкретного посетителя может теоретически затронуть несколько файлов, поэтому новый метод должен возвращать несколько команд:

```
public FileUpdate[] DeleteAllMentions(
    FileContent[] files, string visitorName)
```

Более того, представители бизнес-стороны могут потребовать, чтобы пустые файлы не хранились в рабочем каталоге. Если удаленная запись была последней записью в файле аудита, этот файл тоже необходимо удалить. Чтобы реализовать это требование, можно переименовать `FileUpdate` в `FileAction` и ввести дополнительное поле перечисляемого типа `ActionType`, чтобы указать тип операции (обновление или удаление).

Функциональная архитектура также упрощает обработку ошибок и позволяет делать ее более явной. Ошибки теперь можно встроить в сигнатуру метода — как часть класса `FileUpdate` или как отдельный компонент:

```
public (FileUpdate update, Error error) AddRecord(
    FileContent[] files,
    string visitorName,
    DateTime timeOfVisit)
```

Сервисам приложения затем нужно будет проверить эту ошибку. Если она есть, то сервис не должен передавать команду обновления в `Persister`, а вместо этого будет показывать пользователю сообщение об ошибке.

6.5. Недостатки функциональной архитектуры

К сожалению, не во всех приложениях можно использовать функциональную архитектуру. И даже если ее можно реализовать, выигрыш от сопровождаемости часто перевешивается потерями быстродействия и возрастанием размера кодовой базы. В этом разделе будут рассмотрены компромиссы, присущие функциональной архитектуре.

6.5.1. Применимость функциональной архитектуры

Функциональная архитектура подошла для нашей системы аудита, потому что система могла получить все входные данные заранее, до принятия решения. Часто логика выполнения оказывается не столь прямолинейной. Иногда приходится запрашивать дополнительные данные от внепроцессной зависимости на основании промежуточного результата в ходе принятия решения.

Допустим, система аудита должна проверять уровень доступа посетителя в случае, если количество его посещений за последние 24 часа превышает некий порог. Также будем считать, что уровни доступа всех посетителей хранятся в базе данных. Вы не сможете просто передать в `AuditManager` экземпляр `IDatabase`:

```
public FileUpdate AddRecord(
    FileContent[] files, string visitorName,
    DateTime timeOfVisit, IDatabase database
)
```

Такой экземпляр создаст скрытые входные данные для метода `AddRecord()`. Следовательно, метод перестанет быть математической функцией (рис. 6.16), а потому возможность применения тестирования на основании проверки выходных данных будет потеряна.

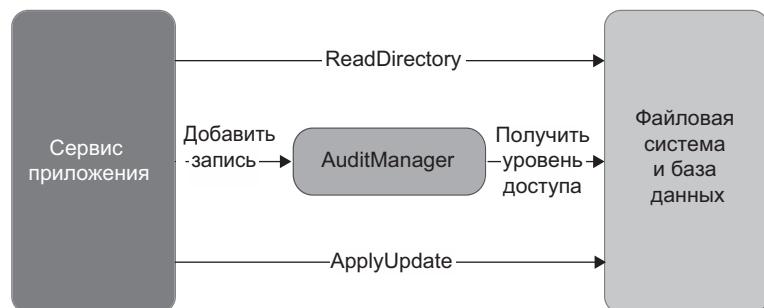


Рис. 6.16. Зависимость от базы данных создает скрытые входные данные для AuditManager. Класс перестает быть чисто функциональным, а все приложение уже не следует правилам функциональной архитектуры

В такой ситуации возможны два решения:

- получить уровень доступа посетителя в службе приложения заранее, вместе с содержимым каталога;
- включить в AuditManager новый метод — например, `IsAccessLevelCheckRequired()`. Сервис приложения будет вызывать этот метод до `AddRecord()`, и если он вернет `true`, сервис будет читать уровень доступа из базы данных и передавать его в `AddRecord()`.

Оба подхода имеют свои недостатки. Первый приводит к снижению быстродействия — он будет всегда обращаться к базе данных, даже в тех случаях, когда уровень доступа не нужен. Но этот подход полностью сохраняет разделение между бизнес-логикой и взаимодействиями с внешними системами: все принятие решений, как и прежде, находится в AuditManager. Второй подход приводит к частичной потере разделения ради улучшения быстродействия. Решение о том, нужно ли обращаться к базе данных, теперь выносится в сервис приложения из AuditManager.

КОЛЛАБОРАТОРЫ И ЗНАЧЕНИЯ

Возможно, вы заметили, что у метода `AddRecord()` из класса AuditManager существует зависимость, не включенная в его сигнатуру: поле `_maxEntriesPerFile`. AuditManager обращается к этому полю для принятия решения относительно того, нужно ли добавлять данные к существующему файлу или же создавать новый файл.

Хотя эта зависимость и не входит в список аргументов метода, она не является скрытой. Ее можно вычислить по сигнатуре конструктора класса. А поскольку поле `_maxEntriesPerFile` неизменяемо, оно остается неизменным между созданием экземпляра класса и вызовом `AddRecord()`. Иначе говоря, это поле является значением. С зависимостью IDatabase ситуация иная, потому что, в отличие от `_maxEntriesPerFile`, это коллaborатор, а не значение. Как говорилось в главе 2, коллаборатором называется зависимость, которая относится к одной из двух следующих категорий:

- ✓ она изменяемая (допускает изменение своего состояния);
- ✓ она является посредником для данных, еще не находящихся в памяти (согласованная зависимость).

Экземпляр IDatabase относится ко второй категории, а следовательно, является коллаборатором. Он требует дополнительного вызова внепроцессной зависимости, а значит, исключает применение тестирования на основании проверки выходных данных.

Следует заметить, что, в отличие от этих двух вариантов, зависимость модели предметной области (AuditManager) от базы данных не является хорошей опцией. Баланс между быстродействием и разделением обязанностей будет более подробно рассмотрен в двух следующих главах.

ПРИМЕЧАНИЕ

Класс из функционального ядра должен работать не с коллаборатором, а с результатом его работы, то есть со значением.

6.5.2. Недостатки по быстродействию

Основным аргументом против функциональной архитектуры обычно становятся последствия по быстродействию для системы в целом. Обратите внимание: страдает не быстродействие тестов. Тесты на основании выходных данных работают так же быстро, как и тесты с моками. Самой системе приходится делать больше вызовов к внепроцессным зависимостям, и она становится менее производительной. Исходная версия системы аудита не читала все файлы из рабочего каталога; не делала этого и версия с моками. Однако итоговой версии приходится делать это для того, чтобы соответствовать схеме «чтение — решение — действие».

Выбор между функциональной и более традиционной архитектурой означает компромисс между быстродействием и сопровождаемостью кода (как рабочего кода, так и кода тестов). В некоторых проектах, где последствия для быстродействия не столь заметны, лучше выбирать функциональную архитектуру для дополнительного выигрыша в простоте поддержки. В других случаях, возможно, придется принимать обратное решение. Единственно правильного решения не существует.

6.5.3. Увеличение размера кодовой базы

Все сказанное выше относится и к размеру кодовой базы. Функциональная архитектура требует четкого разделения между функциональным (неизменяемым) ядром и изменяемой оболочкой. На начальной стадии это требует написания дополнительного кода, хотя в конечном итоге приводит к сокращению сложности кода и улучшению сопровождаемости.

Впрочем, не все проекты обладают достаточно высокой степенью сложности, оправдывающей такие начальные вложения. Некоторые кодовые базы не настолько значительны с точки зрения бизнеса или вообще слишком просты. Нет смысла применять функциональную архитектуру в таких проектах, потому что начальные вложения в них не окупятся. Всегда применяйте функциональную архитектуру стратегически, учитывая сложность и важность вашей системы.

Наконец, не гонитесь за чистотой функционального подхода, если эта чистота дается слишком высокой ценой. В большинстве проектов вы не сможете сделать модель предметной области полностью неизменяемой, а следовательно, не сможете положиться исключительно на тесты, проверяющие выходные данные (по крайней мере, не в таких ООП-языках, как C# или Java). В большинстве случаев у вас будет сочетание стилей проверки выходных данных и состояния с небольшой примесью тестов, проверяющих взаимодействия, и это нормально. Цель этой главы заключается не в том, чтобы побудить вас перевести *все* ваши тесты к стилю проверки выходных данных, а в том, чтобы перевести настолько много их, насколько возможно в разумных пределах.

Итоги

- В стиле тестирования с проверкой выходных данных в тестируемую систему (SUT) подаются входные данные, после чего проверяются полученные результаты. Этот стиль юнит-тестирования предполагает отсутствие скрытых входных и выходных данных, поэтому единственным результатом работы тестируемой системы становится возвращаемое ею значение.
- Тестирование на основании состояния проверяет состояние системы после завершения операции.
- При тестировании взаимодействий тесты используют моки для проверки взаимодействий между тестируемой системой и ее колабораторами.
- Классическая школа юнит-тестирования отдает предпочтение проверкам состояния, а не проверкам взаимодействий. Лондонская школа поступает наоборот. Проверка выходных данных используется обеими школами.
- Тестирование выходных данных производит тесты наивысшего качества. Такие тесты редко завязываются на детали имплементации, а следовательно, устойчивы к рефакторингу. Они также компактны, а значит, обладают наилучшей сопровождаемостью.
- Тестирование состояния требует дополнительных мер для предотвращения хрупкости тестов: вы должны следить за тем, чтобы не раскрыть приватное состояние. Так как тесты, проверяющие состояние, обычно имеют больший размер, чем тесты, проверяющие выходные данные, они также создают больше проблем с сопровождением. Проблемы сопровождения иногда можно преодолеть (но не исключить) использованием вспомогательных методов и объектов-значений.
- Тестирование взаимодействий также требует дополнительных мер для предотвращения хрупкости тестов. Проверяйте только те взаимодействия, которые выходят за границу приложения и результаты работы которых видны внешнему миру. Тесты, проверяющие взаимодействия, уступают по сопровождаемости тестам, проверяющим выходные данные или состояние. Моки обычно занимают много места, что затрудняет чтение тестов.
- Функциональное программирование — программирование с использованием математических функций.
- Математическая функция — это функция (или метод), не имеющая никаких скрытых входных или выходных данных. Побочные эффекты и исключения являются скрытыми выходными данными. Ссылки на внутреннее или внешнее состояние являются скрытыми входными данными. Прямота и открытость математических функций делает их в высшей степени подходящими для юнит-тестирования.

- Цель функционального программирования — отделение бизнес-логики от побочных эффектов.
- Функциональная архитектура помогает достичь такого разделения выведением побочных эффектов на границы бизнес-операций. Такой подход максимизирует объем кода, написанного в чисто функциональном стиле, одновременно сводя к минимуму код, работающий с побочными эффектами.
- Функциональная архитектура делит весь код на две категории: функциональное ядро и изменяемая оболочка. Функциональное ядро принимает решения. Изменяемая оболочка поставляет входные данные функциональному ядру и преобразует решения, принимаемые ядром, в побочные эффекты.
- Различие между функциональной и гексагональной архитектурами проявляются в их отношении к побочным эффектам. Функциональная архитектура вытесняет все побочные эффекты за границы слоя предметной области. С другой стороны, гексагональная архитектура не возражает против побочных эффектов, производимых слоем предметной области, при условии что они ограничиваются только этим слоем предметной области. Функциональную архитектуру можно рассматривать как гексагональную архитектуру, возведенную в абсолют.
- Выбор между функциональной и более традиционной архитектурой означает компромисс между быстродействием и сопровождаемостью кода. Функциональная архитектура жертвує быстродействием ради улучшенной сопровождаемости.
- Применение функциональной архитектуры оправданно не для всех кодовых баз. Всегда применяйте функциональную архитектуру стратегически, учитывая сложность и важность вашей системы. Если кодовая база слишком проста или не настолько важна, начальные вложения, необходимые для функциональной архитектуры, не окупятся.

Рефакторинг для получения эффективных юнит-тестов

В этой главе:

- ✓ Четыре типа кода.
- ✓ Паттерн «Простой объект».
- ✓ Написание эффективных тестов.

В главе 1 были перечислены свойства хороших юнит-тестов:

- интегрированы в цикл разработки;
- ориентированы только на самые важные части вашего кода;
- дают максимальную защиту от ошибок при минимуме затрат на сопровождение.

Для достижения последнего атрибута вы должны уметь:

- распознавать эффективные тесты (и по аналогии — тесты с низкой эффективностью);
- писать эффективные тесты.

В главе 4 рассматривалась тема идентификации эффективных тестов по четырем атрибутам: защите от багов, устойчивости к рефакторингу, быстроте обратной связи и простоте поддержки. В главе 5 подробно рассматривался один из четырех атрибутов: устойчивость к рефакторингу.

Как упоминалось ранее, научиться *распознавать* эффективные тесты недостаточно — нужно также уметь писать такие тесты. Для их написания необходим не только навык распознавания эффективных тестов, но и хорошее владение методами разработки. Юнит-тесты тесно связаны с тестируемым ими кодом, и чтобы создать эффективные тесты, придется основательно поработать над кодом, который они покрывают.

Пример преобразования кода был приведен в главе 6, где мы преобразовали систему аудита к функциональной архитектуре, что позволило применить стиль тестирования выходных данных. В этой главе описанный подход будет обобщен для более широкого спектра применения, включая ситуации, в которых функциональная архитектура использоваться не может. Я приведу практические рекомендации по написанию эффективных тестов практически в любых программных проектах.

7.1. Определение кода для рефакторинга

Значительное улучшение качества тестов обычно возможно только при рефакторинге тестируемого кода. Другого пути нет — между тестовым и рабочим кодом существует тесная связь. В этом разделе представлена классификация кода на четыре типа для определения направления рефакторинга. Затем будет рассмотрен подробный пример.

7.1.1. Четыре типа кода

В этом разделе рассматриваются четыре типа кода, которые служат основанием для всего материала этой главы.

Весь рабочий код можно классифицировать по двум измерениям:

- сложность или важность для проекта;
- количество коллaborаторов.

Сложность кода определяется количеством принимаемых решений (точек ветвления) в коде. Чем больше это число, тем выше сложность.

КАК ВЫЧИСЛЯЕТСЯ ЦИКЛОМАТИЧЕСКАЯ СЛОЖНОСТЬ

В компьютерной теории существует специальный термин для описания сложности кода: цикломатическая сложность. Цикломатическая сложность обозначает количество возможных ветвей в заданной программе или методе. Метрика вычисляется по формуле
 $1 + \langle\text{количество точек ветвления}\rangle$

Таким образом, метод, не содержащий управляющих конструкций (например, команд if или условных циклов), обладает цикломатической сложностью $1 + 0 = 1$.

У этой метрики есть и другой смысл. Ее можно рассматривать как количество независимых путей от точки входа метода до точки выхода или как количество тестов, необходимых для достижения 100%-ного покрытия по метрике branch coverage (покрытие ветвей).

Количество точек ветвления подсчитывается как количество простейших предикатов в коде. Например, команда if условие1 AND условие2 THEN... эквивалентна команде if условие1 THEN if условие2 THEN... Следовательно, его сложность будет равна $1 + 2 = 3$.

Важность для проекта показывает, насколько значимым является код для предметной области (домена) вашего проекта. Обычно весь код в слое предметной области (в доменном слое) напрямую связан с целями конечного пользователя, а следовательно, характеризуется высокой важностью для проекта. С другой стороны, у вспомогательного кода такая связь отсутствует.

Тестируирование сложного кода и кода, обладающего высокой важностью для проекта, приносит больше всего пользы, потому что соответствующие тесты обладают высокой защитой от багов. Следует заметить, что доменный код не обязан быть сложным, а сложный код не обязан быть доменным, для того чтобы заслуживать покрытия тестами. Эти два фактора существуют независимо друг от друга. Например, метод для вычисления стоимости заказа может не содержать условных команд, а следовательно, иметь цикломатическую сложность 1. Тем не менее такой метод важно протестировать, потому что он представляет функциональность, критическую для бизнеса.

Второе измерение — количество коллaborаторов класса или метода. Как говорилось в главе 2, коллаборатором называется зависимость, которая является изменяемой и/или внепроцессной. Тестируирование кода с большим количеством коллaborаторов требует значительных затрат. Это обусловлено метрикой простоты поддержки, которая зависит от размера теста. Чтобы привести коллабораторы к необходимому состоянию, а затем проверить их состояние или взаимодействия с ними, придется написать код, занимающий немало места. И чем больше коллабораторов, тем больше становится тест.

Тип коллабораторов тоже важен. Внепроцессные коллабораторы никогда не должны использоваться в модели предметной области. Они привносят дополнительные затраты на сопровождение, обусловленные необходимостью поддержания моков в тестах. Вы должны быть крайне осмотрительны и использовать моки только для проверки взаимодействий, пересекающих границу приложения, чтобы поддерживать устойчивость к рефакторингу и избегать хрупких тестов (за подробностями обращайтесь к главе 5). Лучше делегировать все взаимодействия с внепроцессными зависимостями классам вне слоя предметной области. Тогда классы предметной области будут работать только с внутрипроцессными зависимостями.

Обратите внимание, что при подсчете количества коллабораторов учитываются как неявные, так и явные коллабораторы. Неважно, получает ли тестируемая система (SUT) коллаборатор как входной параметр или же неявно обращается к нему через статический метод; вам все равно придется настраивать этот коллаборатор в тестах. И наоборот, неизменяемые зависимости (значения или объекты-значения) не считаются. Такие зависимости легко настроить и проверить.

Комбинация этих двух измерений: сложность кода и его важность для проекта, с одной стороны, и с количеством коллабораторов — с другой, образует четыре типа кода, изображенных на рис. 7.1:

- *Модель предметной области и алгоритмы* (наверху слева): сложный код обычно также является и частью доменной модели, но все же не в 100 % случаев. Иногда попадаются сложные алгоритмы, не связанные напрямую с предметной областью проекта.
- *Тривиальный код* (внизу слева): примерами такого кода в C# служат конструкторы без параметров и однострочные свойства. Они имеют минимальное количество коллабораторов (или не имеют вообще) и обладают низкой сложностью и важностью для проекта.
- *Контроллеры* (внизу справа): код сам по себе не выполняет никакой сложной или важной работы, но координирует работу других компонентов (например, классов предметной области и внешних приложений).
- *Переусложненный код* (наверху справа): такой код показывает высокие результаты по обеим метрикам: у него много коллaborаторов и он сложен и/или важен. Примером служат «толстые» контроллеры (контроллеры, которые никому не делегируют сложную работу и делают все сами).

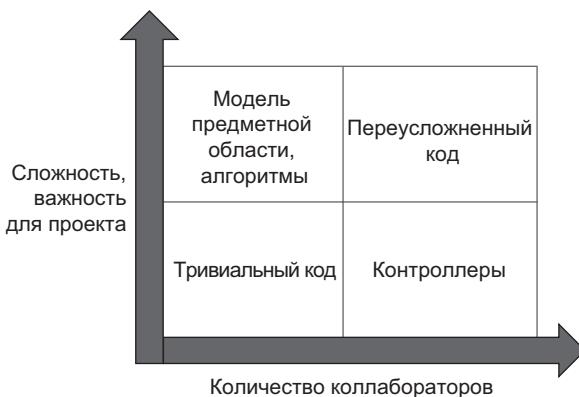


Рис. 7.1. Четыре типа кода, классифицируемых по сложности кода или важности для проекта (вертикальная ось) и количеству колабораторов (горизонтальная ось)

Юнит-тестирование левой верхней четверти (модель предметной области и алгоритмы) обеспечивает тестам наибольшую эффективность. Полученные юнит-тесты дают хорошую защиту от багов, потому что тестируемый код реализует сложную или важную логику. В то же время такие тесты просты в сопровождении, потому что код имеет небольшое количество колабораторов (в идеале ни одного).

Тривиальный код в тестировании вообще не нуждается; ценность таких тестов близка к нулю. Что касается контроллеров, их следует тестировать кратко, как часть интеграционного тестирования (эта тема рассматривается в части III).

Самая проблематичная разновидность кода — переусложненный код. Он создает большие сложности с юнит-тестированием, но оставлять его без тестового покрытия слишком рискованно. Такой код — одна из главных причин, по которым у многих разработчиков возникают проблемы с юнит-тестированием. Вся эта глава посвящена тому, как можно обойти эту дилемму. Общая идея заключается в том, чтобы разбить переусложненный код на две части: алгоритмы и контроллеры (рис. 7.2), хотя реализовать такое разбиение на практике может быть нелегко.

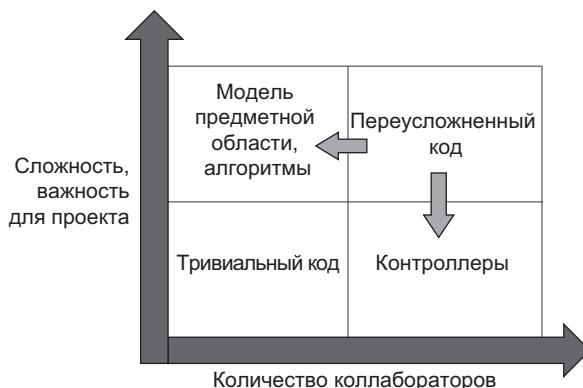


Рис. 7.2. Отрефакторите переусложненный код, разбив его на алгоритмы и контроллеры.
В идеале в правой верхней четверти кода вообще не должно быть

СОВЕТ

Чем важнее или сложнее код, тем меньше у него должно быть коллабораторов.

Избавление от переусложненного кода и юнит-тестирование только модели предметной области и алгоритмов — путь к эффективным и простым в сопровождении тестам. При таком подходе у вас не будет 100%-ного тестового покрытия, но это и не нужно — 100%-ное покрытие не должно быть вашей целью. Вы должны стремиться к тестам, из которых каждый приносит значительную пользу проекту. Все остальные тесты следует отрефакторить или удалить.

ПРИМЕЧАНИЕ

Помните: лучше вообще не писать тест, чем написать плохой тест.

Конечно, избавиться от переусложненного кода не так просто. Тем не менее существуют приемы, которые помогут вам в этом. Сначала я объясню теорию, лежащую в основе этих приемов, а затем продемонстрирую их применение на примере.

7.1.2. Использование паттерна «Простой объект» для разделения переусложненного кода

Для разделения переусложненного кода следует воспользоваться паттерном проектирования «Простой объект» (Humble Object).

Этот паттерн ввел Джерард Месарош (Gerard Meszaros) в своей книге «xUnit Test Patterns: Refactoring Test Code» (Addison-Wesley, 2007) как один из способов борьбы со связыванием кода, но он имеет гораздо более широкое применение.

Часто оказывается, что тестирование кода осложняется его привязкой к зависимостям фреймворка (рис. 7.3). Примеры — асинхронное или многопоточное выполнение, пользовательские интерфейсы, взаимодействие с внепроцессными зависимостями и т. д.

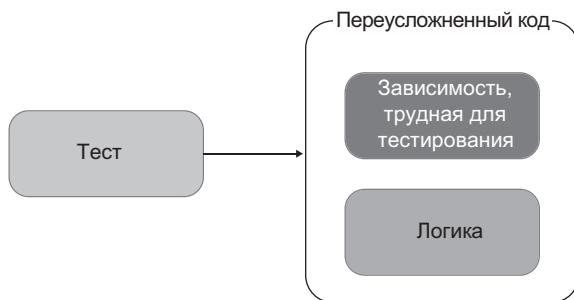


Рис. 7.3. При тестировании кода, связанного со сложной зависимостью, возникают проблемы. Тесты должны напрямую работать с такой зависимостью, что увеличивает затраты на их сопровождение

Чтобы протестировать такой код, необходимо выделить из него тестируемую часть. В результате код становится простой оберткой над тестируемой частью: он «склеивает» трудную для тестирования зависимость с выделенным компонентом, но сам по себе почти не содержит логики и поэтому не нуждается в тестировании (рис. 7.4).

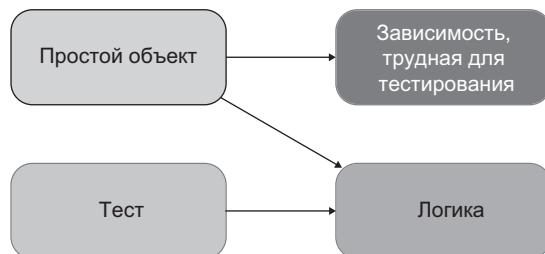


Рис. 7.4. Паттерн «Простой объект» (Humble Object) выделяет логику из переусложненного кода, в результате чего код становится настолько тривиальным, что уже не нуждается в тестировании. Выделенная логика перемещается в другой класс, отделенный от зависимости, трудной для тестирования

Если такой подход кажется вам знакомым, то это потому что вы уже видели его в книге. Как гексагональная, так и функциональная архитектура реализует именно этот паттерн. Как вы помните по предыдущим главам, гексагональная архитектура требует разделения бизнес-логики и взаимодействий с внепроцессными зависимостями. За эти области отвечают слои предметной области и сервисов приложения соответственно.

Функциональная архитектура идет еще дальше: она отделяет бизнес-логику от взаимодействий со всеми колабораторами, не только внепроцессными. Именно этот факт делает функциональную архитектуру настолько удобной для тестирования: ее функциональное ядро вообще не имеет колабораторов. Все зависимости в функциональном ядре неизменяемы, в результате чего оно смещается очень близко к вертикальной оси на диаграмме разновидностей кода (рис. 7.5).

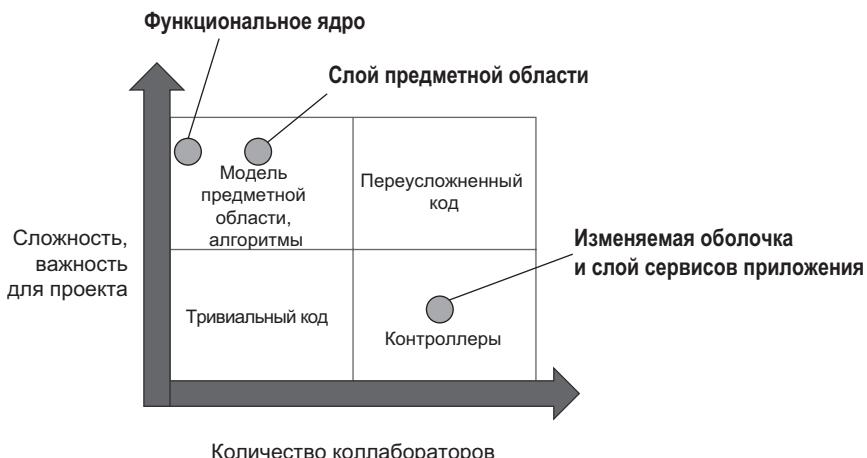


Рис. 7.5. Функциональное ядро в функциональной архитектуре и слой предметной области в гексагональной архитектуре находятся в левой верхней четверти: они имеют небольшое количество колабораторов, высокую сложность или важность для проекта. Функциональное ядро ближе к вертикальной оси, потому что у него вообще нет колабораторов. Изменяемая оболочка (функциональная архитектура) и слой сервисов приложения (гексагональная архитектура) относятся к контроллерам

Также паттерн «Простой объект» можно рассматривать как средство соблюдения принципа единственной ответственности (SRP, Single Responsibility principle), который гласит, что каждый класс должен иметь только одну ответственность¹. Одной из таких обязанностей всегда является бизнес-логика; паттерн может применяться для отделения этой логики практически от чего угодно.

В данной ситуации нас интересует отделение бизнес-логики от координации. Эти две обязанности можно рассматривать в контексте *глубины* и *ширины* кода. Ваш код

¹ См. *Agile Principles, Patterns, and Practices in C#*, Роберт Мартин (Robert C. Martin) и Майка Мартин (Micah Martin), Prentice Hall, 2006.

может быть либо глубоким (сложным или важным), либо широким (работающим со многими коллабораторами), но никогда не должен быть и тем и другим (рис. 7.6).

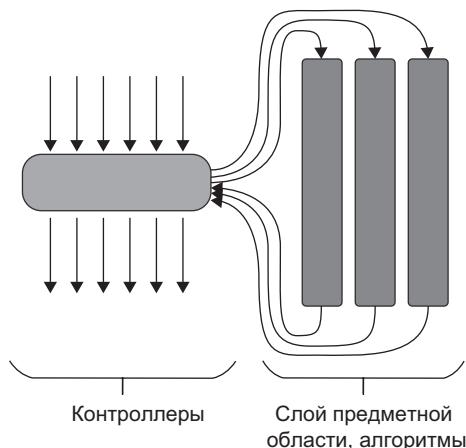


Рис. 7.6. Глубина и ширина кода — полезная метафора, применяемая при анализе отделения бизнес-логики от обязанностей координации. Контроллеры работают с множеством зависимостей (представленных стрелками на схеме), но сами по себе они не сложны (сложность представлена высотой прямоугольника). Классы предметной области обладают противоположными свойствами

Трудно переоценить важность этого разделения. Многие хорошо известные принципы и паттерны можно описать в форме паттерна «Простой объект»: они также отделяют сложный код от кода, который выполняет координацию.

Вы уже видели связь между этим паттерном и гексагональной и функциональной архитектурами. Другие примеры – паттерны MVP (Model-View-Presenter) и MVC (Model-View-Controller). Эти два паттерна помогают разделить бизнес-логику (модель, Model), аспекты UI (представление – View) и координацию между ними (презентер или контроллер – Presenter/Controller). Презентер и контроллер являются *простыми объектами*: они связывают представление с моделью.

Другим примером служит паттерн «Агрегат» из Domain-Driven Design¹. Одна из его целей заключается в сокращении связности между классами посредством их группировки в кластеры — *агрегаты*. Между классами в этих кластерах существует сильная связность, но сами кластеры слабо зависят друг от друга. Такая структура уменьшает общее количество взаимодействий в коде. В свою очередь, снижение связности приводит к упрощению тестирования.

Удобство тестирования — не единственная причина для поддержания разделения между бизнес-логикой и координацией. Такое разделение также помогает справиться

¹ См. *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Эрик Эванс (Eric Evans), Addison-Wesley, 2003.

со сложностью кода, что критично для роста проекта, особенно в долгосрочной перспективе.

7.2. Рефакторинг для получения эффективных юнит-тестов

В этом разделе будет рассмотрен подробный пример разбиения переусложненного кода на алгоритмы и контроллеры. Похожий пример приводился в предыдущей главе, когда речь шла о тестировании выходных данных и функциональной архитектуре. На этот раз этот метод будет обобщен для всех корпоративных приложений с использованием паттерна «Простой объект». Проект будет использоваться не только в этой главе, но и в последующих главах части III.

7.2.1. Знакомство с системой управления клиентами

В примере будет рассматриваться система управления клиентами (CRM, Customer Management System), управляющая регистрацией пользователей. Данные всех пользователей хранятся в базе данных. Система в настоящее время поддерживает только один бизнес-сценарий: изменение адреса электронной почты пользователя. В этой операции задействованы три бизнес-правила:

- если адрес электронной почты (имейл) принадлежит домену компании, то пользователь помечается как работник. В противном случае он помечается как клиент;
- система должна отслеживать количество работников. Если тип пользователя меняется с работника на клиента (или наоборот), то это число тоже должно изменяться;
- при изменении имейла система должна оповестить об этом внешние системы, отправив сообщение по шине сообщений.

Исходная реализация системы CRM приведена в листинге 7.1.

Листинг 7.1. Исходная реализация CRM

```
public class User
{
    public int UserId { get; private set; }
    public string Email { get; private set; }
    public UserType Type { get; private set; }

    public void ChangeEmail(int userId, string newEmail)
    {
        object[] data = Database.GetUserById(userId); ←
        UserId = userId;
        Email = (string)data[1];
        Type = (UserType)data[2];
    }
}
```

Читает текущий имейл и тип пользователя из базы данных

```

if (Email == newEmail)
    return;

object[] companyData = Database.GetCompany();           ← Читает из базы данных
                                                        имя домена организации
                                                        и количество работников

string companyDomainName = (string)companyData[0];
int numberOfEmployees = (int)companyData[1];

string emailDomain = newEmail.Split('@')[1];
bool isEmailCorporate = emailDomain == companyDomainName;
UserType newType = isEmailCorporate
    ? UserType.Employee
    : UserType.Customer;                                | Задает тип пользователя
                                                        в зависимости от имени
                                                        домена

if (Type != newType)
{
    int delta = newType == UserType.Employee ? 1 : -1;
    int newNumber = numberOfEmployees + delta;
    Database.SaveCompany(newNumber);                  ← Обновляет количество
                                                        работников в организации
                                                        при необходимости

}

Email = newEmail;
Type = newType;
Database.SaveUser(this);                            ← Сохраняет информацию
                                                        пользователя в базе данных
MessageBus.SendEmailChangedMessage(UserId, newEmail); ← Отправляет
                                                        уведомление
                                                        по шине сообщений

}

public enum UserType
{
    Customer = 1,
    Employee = 2
}

```

Класс `User` изменяет адрес электронной почты пользователя. Я опустил простые проверки (например, проверку корректности электронной почты и существования пользователя в базе данных) для краткости. Проанализируем эту реализацию с точки зрения диаграммы разновидностей кода.

Сложность кода не слишком высока. Метод `ChangeEmail` содержит только пару явных мест, где принимаются решения: считать пользователя работником или клиентом и обновление количества работников в компании. Несмотря на простоту, эти решения важны: они относятся к бизнес-логике приложения. Следовательно, класс обладает высокими значениями по шкале сложности и важности для проекта.

С другой стороны, класс `User` имеет четыре зависимости, две из которых являются явными, а две другие — неявными. Явные зависимости — аргументы `UserId`

и newEmail. Они являются значениями, а следовательно, не учитываются при подсчете коллабораторов класса. Неявные зависимости — Database и MessageBus — являются внепроцессными коллабораторами. Как упоминалось ранее, внепроцессные коллабораторы не должны использоваться в коде с высокой важностью для проекта. Таким образом, класс User обладает высоким показателем по оси количества коллабораторов, в результате чего класс попадает в категорию переусложненного кода (рис. 7.7).

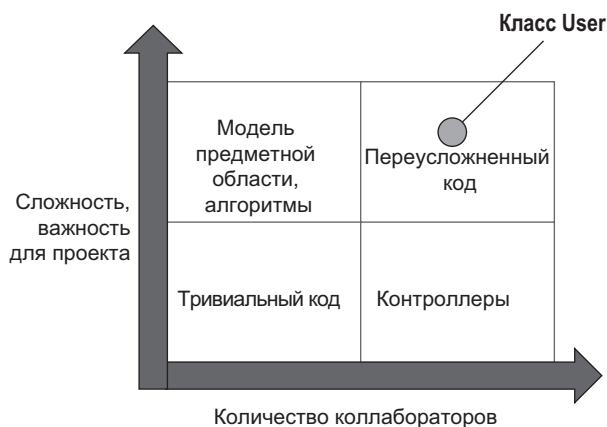


Рис. 7.7. Исходная реализация класса User обладает высокими значениями по обеим осям, а следовательно, попадает в категорию переусложненного кода

Такой подход, при котором класс читает и сохраняет себя в базе данных, называется паттерном «Активная запись» (Active Record). Он хорошо работает в простых или недолговечных проектах, но часто не масштабируется с ростом кодовой базы. Это объясняется именно отсутствием разделения между двумя обязанностями: бизнес-логикой и взаимодействием с внепроцессными зависимостями.

7.2.2. Версия 1: преобразование неявных зависимостей в явные

Обычный подход к улучшению тестируемости основан на преобразовании неявных зависимостей в явные. Для этого можно добавить интерфейсы для Database и MessageBus, внедрить эти интерфейсы в User, а затем заменить на моки в тестах. Такой подход помогает; именно это было сделано в предыдущей главе, когда была представлена реализация с моками для системы аудита. Тем не менее этого недостаточно.

С точки зрения диаграммы типов кода не имеет значения, обращается ли модель предметной области к внепроцессным зависимостям напрямую или через интерфейс. Такие зависимости остаются внепроцессными — они являются посредниками для данных, которые еще не находятся в памяти. Вам все равно придется использовать

моки для тестирования таких классов, что приводит к повышению затрат на сопровождение тестов. Более того, использование моков для базы данных сделает тесты хрупкими (эта тема будет рассмотрена в следующей главе).

Намного лучше делать так, чтобы модель предметной области вообще не зависела от внепроцессных коллaborаторов, неважно, напрямую или косвенно (через интерфейс). Гексагональная архитектура также выступает за этот подход — модель предметной области не должна отвечать за взаимодействия с внешними системами.

7.2.3. Версия 2: уровень сервисов приложения

Чтобы решить проблему прямого взаимодействия доменной модели с внешними системами, необходимо переместить эту обязанность в другой класс — *простой (humble) контроллер* (сервис приложения в терминах гексагональной архитектуры). Классы предметной области должны зависеть только от внутрипроцессных зависимостей (таких как другие классы предметной области) или простых значений. В листинге 7.2 показано, как может выглядеть первая версия этого сервиса приложения.

Листинг 7.2. Сервис приложения, версия 1

```
public class UserController
{
    private readonly Database _database = new Database();
    private readonly MessageBus _messageBus = new MessageBus();

    public void ChangeEmail(int userId, string newEmail)
    {
        object[] data = _database.GetUserById(userId);
        string email = (string)data[1];
        UserType type = (UserType)data[2];
        var user = new User(userId, email, type);

        object[] companyData = _database.GetCompany();
        string companyDomainName = (string)companyData[0];
        int numberOfEmployees = (int)companyData[1];

        int newNumberOfEmployees = user.ChangeEmail(
            newEmail, companyDomainName, numberOfEmployees);

        _database.SaveCompany(newNumberOfEmployees);
        _database.SaveUser(user);
        _messageBus.SendEmailChangedMessage(userId, newEmail);
    }
}
```

Для первой попытки получилось неплохо; сервис приложения помог вынести работу с внепроцессными зависимостями из класса `User`. Тем не менее у этой реализации имеется ряд недостатков:

- Экземпляры внепроцессных зависимостей (`Database` и `MessageBus`) создаются напрямую, а не внедряются. Это создаст проблемы для интеграционных тестов, которые будут написаны для этого класса.
- Контроллер создает экземпляр `User` по данным, полученным из базы данных. Это сложная логика, которая не должна относиться к слою сервисов приложений. Единственной обязанностью этого слоя должна быть координация, а не сложная или важная для проекта логика.
- Сказанное также относится к данным компании. Другая проблема заключается в том, что `User` теперь возвращает обновленное количество работников. Однако количество работников компании не имеет никакого отношения к конкретному пользователю. Эта обязанность должна находиться где-то в другом месте.
- Контроллер сохраняет измененные данные и отправляет уведомления по шине сообщений всегда, независимо от того, отличается ли новый адрес от предыдущего.

Класс `User` теперь довольно прост в тестировании, потому что ему уже не приходится взаимодействовать с внепроцессными зависимостями. У него теперь вообще нет коллегаторов, внепроцессных или внутрипроцессных. Новая версия метода `ChangeEmail` класса `User` выглядит так:

```
public int ChangeEmail(string newEmail,
    string companyDomainName, int numberOfWorkers)
{
    if (Email == newEmail)
        return numberOfWorkers;

    string emailDomain = newEmail.Split('@')[1];
    bool isEmailCorporate = emailDomain == companyDomainName;
    UserType newType = isEmailCorporate
        ? UserType.Employee
        : UserType.Customer;

    if (Type != newType)
    {
        int delta = newType == UserType.Employee?1:-1;
        int newNumber = numberOfWorkers + delta;
        numberOfWorkers = newNumber;
    }

    Email = newEmail;
    Type = newType;

    return numberOfWorkers;
}
```

На рис. 7.8 обозначено текущее состояние классов `User` и `UserController` на нашей диаграмме. Класс `User` переместился в четверть модели предметной области рядом

с вертикальной осью, потому что ему уже не приходится иметь дело с коллабораторами. С классом `UserController` дело обстоит сложнее. Хотя я поместил его в четверть контроллеров, он почти пересекает границу с переусложненным кодом, потому что содержащаяся в нем логика достаточно сложна.

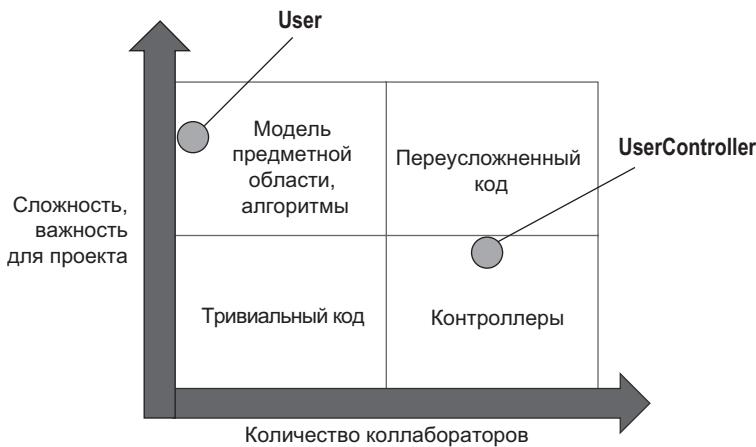


Рис. 7.8. Во второй версии класс `User` размещается в четверти модели предметной области рядом с вертикальной осью. Класс `UserController` почти пересекает границу с переусложненным кодом, потому что он содержит сложную логику

7.2.4. Версия 3: вынесение сложности из сервисов приложения

Чтобы класс `UserController` однозначно относился к четверти контроллеров, необходимо выделить из него логику создания доменных классов. Если вы используете ORM (Object Relational Mapping), библиотеку для отображения базы данных на модель предметной области, она будет хорошим местом для размещения такой логики. У каждой библиотеки ORM имеется специальная область, в которой вы указываете, как таблицы базы данных должны отображаться на классы предметной области (например, с использованием атрибутов в этих классах, XML-файлов или файлов с fluent (текущими) отображениями).

Если использование ORM-библиотеки по каким-то причинам невозможно, создайте в доменной модели фабрику, которая будет создавать экземпляры классов предметной области на основании данных, полученных из базы. Фабрика может представлять собой отдельный класс или, в более простых случаях, статический метод в существующих классах предметной области. Логика создания в нашем приложении не слишком сложна, но ее все же лучше держать отдельно от самих доменных классов, поэтому я поместил ее в отдельный класс `UserFactory`, как показано в листинге 7.3.

Листинг 7.3. Фабрика User

```
public class UserFactory
{
    public static User Create(object[] data)
    {
        Precondition.Requires(data.Length >= 3);

        int id = (int)data[0];
        string email = (string)data[1];
        UserType type = (UserType)data[2];

        return new User(id, email, type);
    }
}
```

Этот код полностью изолирован от всех коллег, а следовательно, легко тестируется. Обратите внимание на проверку, которую я включил в этот метод: требование о том, чтобы массив данных содержал не менее трех элементов. `Precondition` — вспомогательный класс, который выдает исключение в случае, если входной параметр ложен. Для чего он нужен? Для более компактного кода и инверсии проверяемого условия: положительные утверждения читаются проще, чем отрицательные. В нашем примере `data.Length >= 3` читается лучше, чем

```
if (data.Length < 3)
    throw new Exception();
```

Хотя логика создания получается относительно сложной, она не относится к логике предметной области, потому что не связана напрямую с целью клиента по изменению адреса электронной почты. Она относится к категории вспомогательного кода, о котором я упоминал в предыдущих главах.

ПОЧЕМУ ЛОГИКА СОЗДАНИЯ НАЗВАНА СЛОЖНОЙ?

Почему я назвал логику реконструкции сложной, хотя метод `UserFactory.Create()` содержит всего одно ветвление? Как упоминалось в главе 1, в библиотеках, используемых кодом, может быть много скрытых точек принятия решений, а следовательно, много возможностей для неправильного выполнения кода. Именно так дело обстоит с методом `UserFactory.Create()`.

Обращение к элементу массива по индексу (`data[0]`) приводит к внутреннему решению, принимаемому .NET Framework относительно того, к какому элементу данных следует обращаться. То же относится к преобразованию объекта в `int` или `string`. В своей внутренней реализации .NET Framework решает, нужно ли выдать исключение или разрешить дальнейшее преобразование. Из-за всех этих скрытых ветвей логика создания требует тестирования, несмотря на отсутствие в ней точек принятия решений.

7.2.5. Версия 4: новый класс Company

Еще раз обратите внимание на следующий код в контроллере:

```
object[] companyData = _database.GetCompany();
string companyDomainName = (string)companyData[0];
int numberOfEmployees = (int)companyData[1];

int newNumberOfEmployees = user.ChangeEmail(
    newEmail, companyDomainName, numberOfEmployees);
```

Неудобство возвращения обновленного количества работников из `User` — признак неправильного распределения обязанностей, что само по себе является признаком отсутствующей абстракции. Чтобы исправить ситуацию, следует добавить еще один класс предметной области `Company`, связывающий воедино данные и логику, относящуюся к компании, как показано в листинге 7.4.

Листинг 7.4. Новый класс слоя предметной области

```
public class Company
{
    public string DomainName { get; private set; }
    public int NumberOfEmployees { get; private set; }

    public void ChangeNumberOfEmployees(int delta)
    {
        Precondition.Requires(NumberOfEmployees + delta >= 0);

        NumberOfEmployees += delta;
    }

    public bool IsEmailCorporate(string email)
    {
        string emailDomain = email.Split('@')[1];
        return emailDomain == DomainName;
    }
}
```

Класс содержит два метода: `ChangeNumberOfEmployees()` и `IsEmailCorporate()`. Они помогают соблюдать принцип tell-don't-ask, упоминавшийся в главе 5. Этот принцип выступает за упаковку данных вместе с операциями над этими данными. Экземпляр `User` говорит компании изменить ее количество работников или определить, является ли конкретный адрес электронной почты корпоративным; он не запрашивает данные, чтобы сделать все самостоятельно.

Также появился новый класс `CompanyFactory`, который отвечает за реконструкцию объектов `Company` (по аналогии с `UserFactory`). В листинге 7.5 показано, как теперь выглядит контроллер.

Листинг 7.5. Контроллер после рефакторинга

```
public class UserController
{
    private readonly Database _database = new Database();
    private readonly MessageBus _messageBus = new MessageBus();

    public void ChangeEmail(int userId, string newEmail)
    {
        object[] userData = _database.GetUserById(userId);
        User user = UserFactory.Create(userData);

        object[] companyData = _database.GetCompany();
        Company company = CompanyFactory.Create(companyData);

        user.ChangeEmail(newEmail, company);

        _database.SaveCompany(company);
        _database.SaveUser(user);
        _messageBus.SendEmailChangedMessage(userId, newEmail);
    }
}
```

В листинге 7.6 приведен класс User.

Листинг 7.6. Класс User после рефакторинга

```
public class User
{
    public int UserId { get; private set; }
    public string Email { get; private set; }
    public UserType Type { get; private set; }

    public void ChangeEmail(string newEmail, Company company)
    {
        if (Email == newEmail)
            return;

        UserType newType = company.IsEmailCorporate(newEmail)
            ? UserType.Employee
            : UserType.Customer;

        if (Type != newType)
        {
            int delta = newType == UserType.Employee ? 1 : -1;
            company.ChangeNumberOfEmployees(delta);
        }

        Email = newEmail;
        Type = newType;
    }
}
```

Обратите внимание на то, что с удалением неуместно размещенной обязанности код User стал намного чище. Вместо того чтобы работать с данными компаний, он

получает экземпляр `Company` и делегирует ему две важные части работы: определение того, является ли адрес электронной почты корпоративным, и изменение количества работников в компании.

На рис. 7.9 показано место каждого класса на диаграмме. Фабрики и оба класса предметной области находятся в четверти модели предметной области и алгоритмов. Класс `User` переместился немного правее, потому что теперь у него есть один коллега `Company`, тогда как ранее не было ни одного. Это затрудняет тестирование класса `User`, но ненамного.

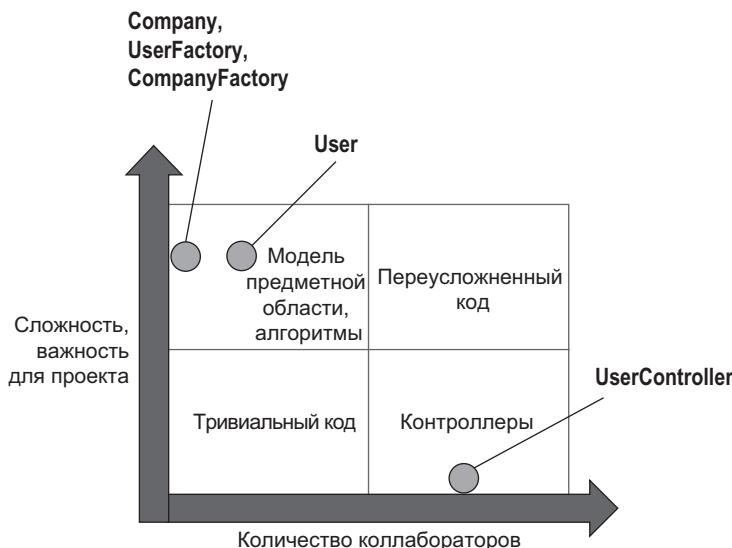


Рис. 7.9. Класс `User` сместился вправо, потому что теперь у него есть один коллега `Company`. Класс `UserController` теперь однозначно размещается в четверти контроллеров, а вся сложность переместила в фабрики

`UserController` теперь однозначно размещается в четверти контроллеров, потому что вся его сложность переместилась в фабрики. Теперь единственная обязанность этого класса — связывание всех общающихся компонентов.

Обратите внимание на сходство этой реализации с функциональной архитектурой из предыдущей главы. Ни функциональное ядро в системе аудита, ни уровень предметной области в этой CRM (классы `User` и `Company`) не взаимодействуют с внепроцессными зависимостями. В обоих реализациях слой сервисов приложения отвечает за такое взаимодействие: он получает данные от файловой системы или из базы данных, передает их алгоритмам или модели предметной области, а затем сохраняет результаты в хранилище данных.

Различия между двумя реализациями проявляются в их отношении к побочным эффектам. Функциональное ядро не создает никаких побочных эффектов. Доменная

модель CRM их создает, но все эти побочные эффекты остаются в этой модели в форме измененного адреса электронной почты и количества работников. Побочные эффекты выходят за границу доменной модели только тогда, когда контроллер сохраняет объекты `User` и `Company` в базе данных.

Тот факт, что все побочные эффекты содержатся в памяти до самого последнего момента, значительно упрощает тестирование. Вашим тестам не нужно ни анализировать внепроцессные зависимости, ни прибегать к тестированию взаимодействий. Вся проверка может осуществляться посредством тестирования выходных данных и состояния объектов в памяти.

7.3. Анализ оптимального покрытия юнит-тестов

Итак, рефакторинг на основе паттерна «Простой объект» (Humble Object) завершен. Теперь проанализируем, какие части проекта относятся к той или иной категории кода и как эти части должны тестироваться. В таблице 7.1 представлен весь код проекта, сгруппированный по их позиции на диаграмме разновидностей кода.

Таблица 7.1. Типы кода в проекте после рефакторинга с использованием паттерна «Простой объект»

	Мало коллабораторов	Много коллaborаторов
Высокая сложность или важность для проекта	<code>ChangeEmail(newEmail, company)</code> в <code>User</code> ; <code>ChangeNumberOfEmployees(delta)</code> и <code>IsEmailCorporate(email)</code> в <code>Company</code> ; <code>Create(data)</code> в <code>UserFactory</code> и <code>CompanyFactory</code>	
Низкая сложность и важность для проекта	Конструкторы в <code>User</code> и <code>Company</code>	<code>ChangeEmail(userId, newEmail)</code> в <code>UserController</code>

При полном разделении бизнес-логики и координации легко решить, какие части кода тестировать.

7.3.1. Тестирование слоя предметной области и вспомогательного кода

Тестирование методов в левой верхней четверти таблицы 7.1 обеспечивает наилучшее соотношение защиты от багов к затратам на сопровождение. Высокая сложность или важность для проекта кода гарантирует отличную защиту от багов, тогда как небольшое количество коллабораторов гарантирует минимальные затраты на сопровождение. Пример тестирования класса `User`:

```
[Fact]
public void Changing_email_from_non_corporate_to_corporate()
{
    var company = new Company("mycorp.com", 1);
    var sut = new User(1, "user@gmail.com", UserType.Customer);

    sut.ChangeEmail("new@mycorp.com", company);

    Assert.Equal(2, company.NumberOfEmployees);
    Assert.Equal("new@mycorp.com", sut.Email);
    Assert.Equal(UserType.Employee, sut.Type);
}
```

Для достижения полного покрытия понадобятся еще три таких теста:

```
public void Changing_email_from_corporate_to_non_corporate()
public void Changing_email_without_changing_user_type()
public void Changing_email_to_the_same_one()
```

Тесты для трех других классов будут еще короче, а для группировки тестовых сценариев можно воспользоваться параметризованными тестами:

```
[InlineData("mycorp.com", "email@mycorp.com", true)]
[InlineData("mycorp.com", "email@gmail.com", false)]
[Theory]
public void Differentiates_a_corporate_email_from_non_corporate(
    string domain, string email, bool expectedResult)
{
    var sut = new Company(domain, 0);

    bool isEmailCorporate = sut.IsEmailCorporate(email);

    Assert.Equal(expectedResult, isEmailCorporate);
}
```

7.3.2. Тестирование кода из трех других четвертей

Код с низкой сложностью и небольшим количеством коллабораторов (левая нижняя четверть в таблице 7.1) представлен конструкторами классов `User` и `Company`, например:

```
public User(int userId, string email, UserType type)
{
    UserId = userId;
    Email = email;
    Type = type;
}
```

Эти конструкторы тривиальны, и на них не стоит тратить время. Полученные тесты не обеспечат достаточной защиты от багов.

Рефакторинг исключил весь код с высокой сложностью и большим количеством коллабораторов (правая верхняя четверть в таблице 7.1), поэтому и здесь тестировать нечего. Что касается контроллеров (правая нижняя четверть в таблице 7.1), их тестирование будет рассмотрено в следующей главе.

7.3.3. Нужно ли тестировать предусловия?

Возьмем особую разновидность точек принятия решений — *предусловия (pre-conditions)* — и посмотрим, нужно ли их тестировать. Например, взгляните на следующий метод класса `Company`:

```
public void ChangeNumberOfEmployees(int delta)
{
    Precondition.Requires(NumberOfEmployees + delta >= 0);

    NumberOfEmployees += delta;
}
```

Метод имеет предусловие, которое гласит, что количество работников в компании никогда не должно быть отрицательным. Это предусловие — защитная мера, которая активизируется только в исключительных ситуациях. Такие исключительные ситуации обычно возникают в результате ошибок. Единственная возможная причина, из-за которой количество работников может стать отрицательным, — наличие ошибки в коде. Эта защитная мера обеспечивает быстрое выявление ошибки и предотвращает распространение ошибки и ее сохранение в базе данных, где справиться с ней будет намного сложнее. Нужно ли тестировать такие предусловия? Иначе говоря, будут ли такие тесты достаточно ценными для включения в проект?

Однозначных правил на этот счет нет, но в общем случае я рекомендую тестировать все предусловия, которые относятся к предметной области (домену) приложения. Требование неотрицательности количества работников является таким предусловием. Оно является частью инвариантов класса `Company`: условий, которые должны всегда соблюдаться. Но не тратьте время на тестирование предусловий, не относящихся к предметной области. Например, `UserFactory` содержит следующую проверку в своем методе `Create`:

```
public static User Create(object[] data)
{
    Precondition.Requires(data.Length >= 3);

    /* Извлечь идентификатор, адрес электронной почты и тип */
}
```

Это предусловие не имеет смысла для предметной области, и ее тестирование особой ценности не представляет.

7.4. Условная логика в контроллерах

Обработка условной логики одновременно с поддержанием слоя предметной области свободным от внепроцессных коллaborаторов часто создает проблемы и требует различных компромиссов. В этом разделе я покажу, в чем суть этих компромиссов и как решить, какие из них стоит выбрать для вашего проекта.

Разделение между бизнес-логикой и координацией лучше всего работает в том случае, когда бизнес-операция состоит из трех четко определенных фаз:

- чтение данных из базы;
- выполнение бизнес-логики;
- сохранение данных в базу (рис. 7.10).

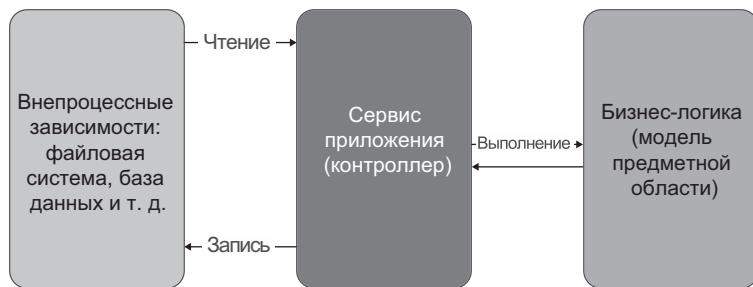


Рис. 7.10. Гексагональные и функциональные архитектуры лучше всего работают в тех ситуациях, когда все обращения к внепроцессным зависимостям могут быть вытеснены к границам бизнес-операций

Во многих ситуациях эти фазы не так четко отделены друг от друга. Как обсуждалось в главе 6, иногда вам приходится запрашивать дополнительные данные от внепроцессных зависимостей на основании промежуточного результата в процессе принятия решений (рис. 7.11). Запись во внепроцессную зависимость тоже часто зависит от этого результата.

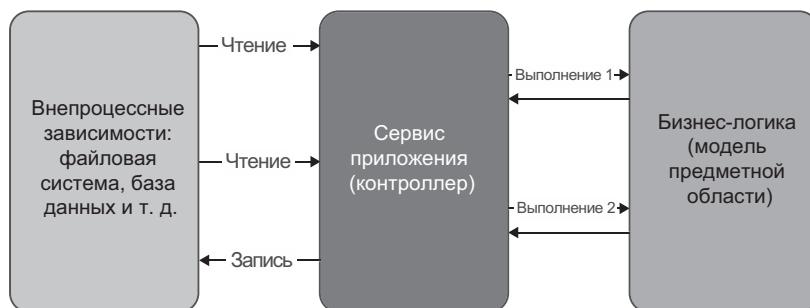


Рис. 7.11. Гексагональная архитектура не так хорошо работает, если вам требуется обращаться к внепроцессным зависимостям в середине бизнес-операции

Как обсуждалось в предыдущей главе, в такой ситуации возможны три варианта:

- *переместить все внешние операции чтения и записи к границам.* Такой подход сохраняет структуру «чтение — решение — действие», но приводит к потере быстродействия: контроллер будет обращаться к внепроцессным зависимостям, даже если в этом нет необходимости;
- *внедрить внепроцессные зависимости в модель предметной области* и дать бизнес-логике непосредственно решать, когда следует вызывать эти зависимости;
- *разбить процесс принятия решений на более мелкие шаги* и дать контроллеру действовать на основании каждого из этих шагов по отдельности.

Проблема заключается в том, чтобы найти баланс между следующими тремя атрибутами:

- *тестируемость модели предметной области*, что зависит от количества и типа колабораторов в классах предметной области;
- *простота контроллера*, зависящая от присутствия точек принятия решений (ветвлений) в контроллере;
- *быстродействие*, определяемое как количество обращений к внепроцессным зависимостям.

Каждый вариант обеспечивает только два из трех атрибутов (рис. 7.12):

- *перемещение всех внешних операций чтения и записи к границам бизнес-операции* сохраняет простоту контроллера и изоляцию модели предметной области от внепроцессных зависимостей (и ее хорошую тестируемость), но с потерями для быстродействия;
- *внедрение внепроцессных зависимостей в модель предметной области* сохраняет быстродействие и простоту контроллера, но с потерями для тестируемости модели предметной области;
- *разбиение процесса принятия решений на более мелкие шаги* помогает с быстродействием и тестируемостью модели предметной области, но с потерями для простоты контроллера. Для управления этими мелкими шагами вам придется включить в контроллер точки принятия решений.

В большинстве проектов быстродействие имеет важное значение, поэтому первый подход (перемещение внешних операций чтения и записи к границам бизнес-операции) исключается. Второй вариант (внедрение внепроцессных зависимостей в модель предметной области) переводит большую часть кода в переусложненную четверть диаграммы разновидностей кода. Именно для того, чтобы уйти от этой ситуации, мы проводили исходный рефакторинг CRM. Я рекомендую избегать этого подхода: такой код уже не обеспечивает разделения бизнес-логики и взаимодействий с внепроцессными зависимостями, а это значительно усложняет его тестирование и сопровождение.

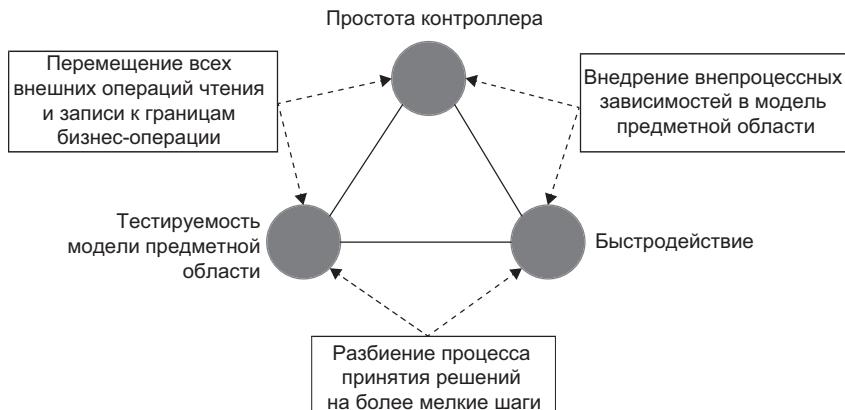


Рис. 7.12. Не существует единственного решения, которое обеспечивает все три атрибута: простоту контроллера, тестируемость модели предметной области и быстродействие. Придется выбрать два атрибута из трех

Остается третий вариант: разбиение процесса принятий решений на более мелкие шаги. При таком подходе вам придется сделать контроллеры более сложными, что приведет к их смещению к переусложненной четверти. Тем не менее проблему можно сгладить. Хотя вам обычно не удастся выделить из контроллеров всю сложность, как это делалось в примере, эту сложность можно сделать менее проблематичной.

7.4.1. Паттерн «CanExecute/Execute»

Первый способ преодоления сложности контроллеров основан на использовании паттерна «CanExecute/Execute», предотвращающего проникновение бизнес-логики из модели предметной области в контроллеры. Этот паттерн лучше всего пояснить на примере, поэтому мы расширим свой проект дополнительным требованием.

Предположим, что пользователь может изменить свой имейл только до того момента, когда он его подтвердит. Если пользователь попытается изменить имейл после подтверждения, должно появиться сообщение об ошибке. Чтобы реализовать это новое требование, мы добавим в класс `User` новое свойство.

Листинг 7.7. Класс User с новым свойством

```
public class User
{
    public int UserId { get; private set; }
    public string Email { get; private set; }
    public UserType Type { get; private set; }
    public bool IsEmailConfirmed | Новое свойство
        { get; private set; }

    /* Метод ChangeEmail(newEmail, company) */
}
```

Где должна размещаться такая проверка? Возможны два варианта. Во-первых, ее можно разместить в методе `ChangeEmail` класса `User`:

```
public string ChangeEmail(string newEmail, Company company)
{
    if (IsEmailConfirmed)
        return "Can't change a confirmed email";

    /* ... */
}
```

Тогда контроллер может либо вернуть ошибку, либо инициировать все необходимые побочные эффекты в зависимости от результата метода:

Листинг 7.8. Контроллер, все еще избавленный от принятия решений

```
public string ChangeEmail(int userId, string newEmail)
{
    object[] userData = _database.GetUserById(userId);
    User user = UserFactory.Create(userData);

    object[] companyData = _database.GetCompany();
    Company company = CompanyFactory.Create(companyData);

    string error = user.ChangeEmail(newEmail, company);
    if (error != null)
        return error;

    _database.SaveCompany(company);
    _database.SaveUser(user);
    _messageBus.SendEmailChangedMessage(userId, newEmail);

    return "OK";
}
```



Подготавливает данные

Принимает решение

Действует по результатам решения

Эта реализация избавляет контроллер от принятия решений, но делает это за счет потерь для быстродействия. Экземпляр `Company` читается из базы данных всегда, даже если имейл подтвержден и изменяться не может. Это пример выведения всех внешних операций чтения и записи к границам бизнес-операции.

ПРИМЕЧАНИЕ

Я не считаю новую команду `if`, анализирующую строку ошибки, увеличением сложности контроллера, потому что она принадлежит фазе действий; она не является частью процесса принятия решений. Все решения принимаются классом `User`, контроллер только действует на основании этих решений.

Второй вариант заключается в перемещении проверки `IsEmailConfirmed` из класса `User` в контроллер.

Листинг 7.9. Решение о том, можно ли изменить адрес электронной почты, принимается контроллером

```
public string ChangeEmail(int userId, string newEmail)
{
    object[] userData = _database.GetUserId(userId);
    User user = UserFactory.Create(userData);

    if (user.IsEmailConfirmed) | Принятие решения переместилось
        return "Can't change a confirmed email"; | из User в эту точку

    object[] companyData = _database.GetCompany();
    Company company = CompanyFactory.Create(companyData);

    user.ChangeEmail(newEmail, company);

    _database.SaveCompany(company);
    _database.SaveUser(user);
    _messageBus.SendEmailChangedMessage(userId, newEmail);

    return "OK";
}
```

С такой реализацией быстродействие остается в порядке: экземпляр `Company` читается из базы данных только в том случае, если имейл может быть изменен. Но теперь процесс принятия решений разбивается на две части:

- можно ли продолжить изменение адреса электронной почты (выполняется контроллером);
- что делать во время этого изменения (выполняется в `User`).

Теперь адрес электронной почты можно изменить без предварительной проверки `IsEmailConfirmed`, что идет в ущерб инкапсуляции модели предметной области. Такая фрагментация препятствует разделению между бизнес-логикой и координацией, а контроллер смещается ближе к переусложненной зоне.

Для предотвращения фрагментации можно включить в `User` новый метод `CanChangeEmail()` и сделать его успешное выполнение предусловием для изменения адреса. Измененная версия в листинге 7.10 построена на основе паттерна `CanExecute/Execute`.

Такое решение обладает двумя важными преимуществами.

- Контроллеру больше не нужно ничего знать о процессе изменения адреса электронной почты. Все, что ему нужно, — вызвать метод `CanChangeEmail()`, чтобы понять, возможно ли выполнить операцию. Этот метод также может содержать другие проверки, инкапсулированные от контроллера.
- Дополнительное предусловие в `ChangeEmail()` гарантирует, что адрес электронной почты не будет изменен без предварительной проверки.

Листинг 7.10. Изменение адреса электронной почты с использованием паттерна CanExecute/Execute

```
public string CanChangeEmail()
{
    if (IsEmailConfirmed)
        return "Can't change a confirmed email";

    return null;
}

public void ChangeEmail(string newEmail, Company company)
{
    Precondition.Requires(CanChangeEmail() == null);

    /* ... */
}
```

Паттерн помогает консолидировать все решения на уровне предметной области. Контроллер уже не сможет не проверить имейл, что фактически устраниет из контроллера новую точку принятия решений. Таким образом, хотя контроллер все еще содержит команду `if` с вызовом `CanChangeEmail()`, проверять эту команду `if` не нужно. Юнит-тестирования самого предусловия в классе `User` будет достаточно.

ПРИМЕЧАНИЕ

Для простоты я использую строку для обозначения ошибки. В реальном проекте вы, вероятно, будете использовать специальный класс `Result` для обозначения успеха или неудачи операции.

7.4.2. Использование доменных событий для отслеживания изменений доменной модели

Иногда бывает трудно определить, какие действия привели доменную модель в текущее состояние. Тем не менее знать эти шаги бывает полезно, потому что вам нужно информировать внешние системы о том, что именно произошло в вашем приложении. Размещение этих обязанностей в контроллерах приведет к их усложнению. Чтобы избежать этого, можно отслеживать важные изменения в доменной модели, а затем преобразовать их в вызовы внепроцессных зависимостей после завершения бизнес-операции. Доменные события помогут вам реализовать получение информации такого рода.

В нашей CRM-системе тоже существует необходимость в получении такой информации: она должна уведомлять внешние системы об изменении адреса электронной почты, отправляя сообщения по шине сообщений. В текущей реализации в функциональности уведомления присутствует дефект: она отправляет сообщения даже в том случае, если адрес электронной почты не изменился, как показано в листинге 7.11.

ОПРЕДЕЛЕНИЕ

Доменное событие описывает событие приложения, которое имеет смысл для эксперта в предметной области. Осмысленность с точки зрения предметной области — то, что отличает доменные события от обычных событий (например, нажатий на кнопки). Доменные события часто используются для оповещения внешних приложений о важных изменениях, произошедших в вашей системе.

Листинг 7.11. Уведомление отправляется даже в том случае, если адрес не изменился

```
// User
public void ChangeEmail(string newEmail, Company company)
{
    Precondition.Requires(CanChangeEmail() == null);

    if (Email == newEmail)      ← Адрес электронной почты
        return;                | может не изменяться

    /* ... */

// Контроллер
public string ChangeEmail(int userId, string newEmail)
{
    /* подготовка */

    user.ChangeEmail(newEmail, company);

    _database.SaveCompany(company);
    _database.SaveUser(user);
    _messageBus.SendEmailChangedMessage(   | Но контроллер все равно
        userId, newEmail);               | отправит сообщение

    return "OK";
}
```

Ошибка можно устранить, переместив проверку совпадения имейла в контроллер, но тогда опять возникнут проблемы с фрагментацией бизнес-логики. И переместить эту проверку в `CanChangeEmail()` не получится, потому что приложение не должно возвращать ошибку, если новый адрес совпадает со старым.

Следует заметить, что эта конкретная проверка, вероятно, не создаст слишком значительной фрагментации бизнес-логики, поэтому лично я бы не считал контроллер переусложненным, если бы он содержал такую проверку. Но вы можете оказаться в более сложной ситуации, в которой трудно предотвратить лишние вызовы от вашего приложения к внепроцессным зависимостям без передачи этих зависимостей доменной модели, что привело бы к переусложнению этой модели. Избежать такого переусложнения можно только одним способом: при помощи доменных событий.

С точки зрения реализации доменное событие представляет собой класс с данными, необходимыми для уведомления внешних систем. В нашем конкретном примере это идентификатор пользователя и новый имейл:

```
public class EmailChangedEvent
{
    public int UserId { get; }
    public string NewEmail { get; }
}
```

ПРИМЕЧАНИЕ

Имена доменных событий всегда должны записываться в прошедшем времени, потому что они представляют то, что уже произошло. Доменные события являются значениями — они неизменяемы и взаимозаменяемы.

User содержит коллекцию событий, в которую при изменении имейла добавляется новый элемент. В листинге 7.12 показано, как метод ChangeEmail() выглядит после рефакторинга.

Листинг 7.12. Класс User добавляет событие при изменении адреса электронной почты

```
public void ChangeEmail(string newEmail, Company company)
{
    Precondition.Requires(CanChangeEmail() == null);

    if (Email == newEmail)
        return;

    UserType newType = company.IsEmailCorporate(newEmail)
        ? UserType.Employee
        : UserType.Customer;

    if (Type != newType)
    {
        int delta = newType == UserType.Employee ? 1 : -1;
        company.ChangeNumberOfEmployees(delta);
    }

    Email = newEmail;
    Type = newType;
    EmailChangedEvents.Add(
        new EmailChangedEvent(UserId, newEmail)); | Новое событие обозначает
                                                    | изменение имейла
}

Контроллер затем преобразует события в сообщения, передаваемые по шине.
```

Листинг 7.13. Обработка доменных событий контроллером

```
public string ChangeEmail(int userId, string newEmail)
{
    object[] userData = _database.GetUserId(userId);
    User user = UserFactory.Create(userData);

    string error = user.CanChangeEmail();
    if (error != null)
        return error;

    object[] companyData = _database.GetCompany();
    Company company = CompanyFactory.Create(companyData);

    user.ChangeEmail(newEmail, company);

    _database.SaveCompany(company);
    _database.SaveUser(user);
    foreach (var ev in user.EmailChangedEvents)
    {
        _messageBus.SendEmailChangedMessage(
            ev.UserId, ev.NewEmail);
    }

    return "OK";
}
```

Обработка доменных
событий

Обратите внимание, что экземпляры `Company` и `User` все еще сохраняются в базе данных вне зависимости от доменных событий. Это объясняется различием между изменениями в базе данных и передачей сообщений по шине.

Если предположить, что никакое другое приложение, кроме CRM, не имеет доступа к базе данных, взаимодействия с базой данных не являются частью наблюдаемого поведения CRM — они являются деталями имплементации. Если итоговое состояние базы корректно, то неважно, сколько обращений к базе данных выдает ваше приложение. С другой стороны, взаимодействия по шине сообщений являются частью наблюдаемого поведения приложения. Чтобы не нарушать контракт с внешними системами, система CRM должна отправлять сообщения в шину только при изменении имейла.

Постоянное сохранение в базе данных также имеет некоторые последствия для быстродействия, но они относительно незначительны. Вероятность того, что после всех проверок новый адрес будет совпадать со старым, достаточно мала. Применение ORM-библиотеки также может помочь с этим. Большинство ORM-библиотек не будут обращаться к базе данных, если состояние объекта не изменилось.

Эту обработку доменных событий можно обобщить, выделив базовый класс `DomainEvent` и создав базовый класс для всех доменных классов, который будет содержать коллекцию таких событий: `List<DomainEvent> events`. Также можно написать отдельный диспетчер событий, вместо того чтобы отправлять события вручную

в контроллерах. Наконец, в более крупных проектах может понадобиться механизм для слияния доменных событий перед их отправкой. Впрочем, эта тема выходит за рамки книги. Если она вас заинтересует, прочитайте мою статью «Merging domain events before dispatching» по адресу <http://mng.bz/YeVe>.

Доменные события снимают ответственность за принятие решений с контроллера и возлагают ее на доменную модель, тем самым упрощая юнит-тестирование взаимодействий с внешними системами. Вместо того чтобы проверять сам контроллер и использовать моки для проверки внепроцессных зависимостей, можно протестировать только создание доменного события, как показано в листинге 7.14.

Листинг 7.14. Тестирование создания доменного события

```
[Fact]
public void Changing_email_from_corporate_to_non_corporate()
{
    var company = new Company("mycorp.com", 1);
    var sut = new User(1, "user@mycorp.com", UserType.Employee, false);

    sut.ChangeEmail("new@gmail.com", company);

    company.NumberOfEmployees.Should().Be(0);
    sut.Email.Should().Be("new@gmail.com");
    sut.Type.Should().Be(UserType.Customer);
    sut.EmailChangedEvents.Should().Equal(
        new EmailChangedEvent(1, "new@gmail.com"));
}

    |—————|—————|
    | Одновременно проверяет |
    | размер коллекции     |
    | и элемент коллекции |

```

Конечно, вы все равно должны протестировать контроллер, чтобы убедиться в том, что он правильно выполняет координацию, но это потребует намного меньшего количества тестов. Эта тема будет рассматриваться в следующей главе.

7.5. Заключение

В этой главе постоянно прослеживается одна тема: абстрагирование применения изменений к внешним системам. Такое абстрагирование достигается хранением этих изменений в памяти до самого конца бизнес-операции, чтобы их можно было тестировать простыми юнит-тестами без привлечения внепроцессных зависимостей. Доменные события представляют собой абстракции для сообщений, которые будут передаваться по шине. Изменения в классах предметной области представляют собой абстракции для предстоящих изменений в базе данных.

ПРИМЕЧАНИЕ

Тестировать абстракции проще, чем то, что они абстрагируют.

Хотя нам удалось успешно изолировать все принятия решений в доменной модели с помощью доменных событий и паттерна «CanExecute/Execute», это возможно не всегда. В некоторых ситуациях фрагментация бизнес-логики неизбежна.

Например, не существует способа проверки уникальности имейла за пределами контроллера без внедрения внепроцессных зависимостей в доменную модель. Другой пример — сбой внепроцессной зависимости, который должен изменить ход бизнес-операции. Решение относительно того, по какому пути пойти, не может находиться на уровне предметной области, потому что эти внепроцессные зависимости вызываются не слоем предметной области. Вам придется разместить эту логику в контроллере, а затем покрыть ее интеграционными тестами. Но даже с потенциальной фрагментацией отделение бизнес-логики от координирования весьма ценно, потому что оно кардинально упрощает процесс юнит-тестирования.

В дополнение к тому, что вам не удастся избежать размещения части бизнес-логики в контроллерах, вам обычно не удастся исключить всех коллaborаторов из классов предметной области — и это нормально. Один, два и даже три коллаборатора не превратят доменный класс в переусложненный код, при условии что эти коллабораторы не обращаются к внепроцессным зависимостям.

Тем не менее не используйте моки для проверки взаимодействий с такими коллабораторами. Эти взаимодействия не имеют никакого отношения к наблюдаемому поведению модели предметной области. Только самый первый вызов, который идет от контроллера к доменному классу, непосредственно связан с целью этого контроллера. Все последующие вызовы от доменного класса к соседним доменным классам в пределах той же операции являются деталями имплементации.

Рис. 7.13 поясняет сказанное. На нем показаны взаимодействия между компонентами CRM и их отношение к наблюдаемому поведению. Как вы, возможно, помните из главы 5, вопрос о том, является ли метод частью наблюдаемого поведения класса, зависит от того, кто его клиент и каковы его цели. Чтобы быть частью наблюдаемого поведения, метод должен удовлетворять одному из следующих двух критериев:

- он должен быть непосредственно связан с одной из целей клиента;
- он должен создавать изменение во внепроцессной зависимости, видимое для внешних приложений.

Метод `ChangeEmail()` контроллера является частью его наблюдаемого поведения, как и совершающее им обращение к шине сообщений. Метод является точкой входа для внешнего клиента, что обеспечивает выполнение первого критерия. Обращение к шине отправляет сообщения внешним приложениям, удовлетворяя второму критерию. Вы должны проверить оба вызова метода (что является темой следующей главы). Тем не менее последующий вызов от контроллера к `User` не имеет непосредственной связи с целями внешнего клиента. Этого клиента не интересует, как контроллер решит реализовать изменение адреса электронной почты, при условии,

что итоговое состояние системы корректно, а обращение к шине сообщений было успешно выполнено. А значит, вам не следует проверять вызовы, совершаемые контроллером `User`, при тестировании поведения этого контроллера.

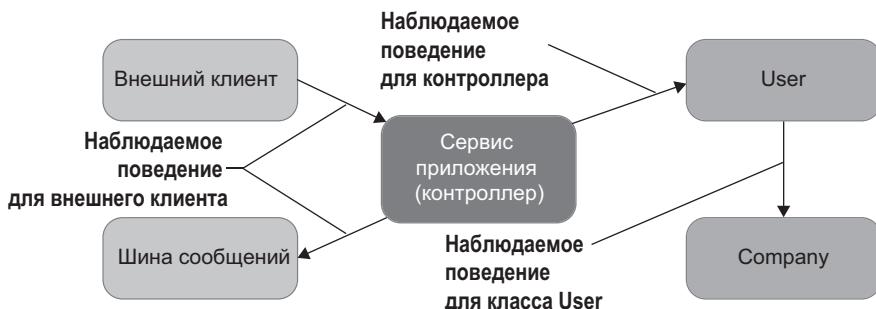


Рис. 7.13. Диаграмма взаимодействий между компонентами CRM, показывающая отношения между этими взаимодействиями и наблюдаемым поведением

Опустившись на один уровень вниз по стеку, вы получите аналогичную ситуацию. Теперь уже контроллер является клиентом, а метод `ChangeEmail` в `User` непосредственно связан с целью клиента по изменению имейла пользователя и, следовательно, должен быть протестирован. Однако последующие обращения от `User` к `Company` с точки зрения контроллера являются деталями имплементации. Таким образом, тест, покрывающий метод `ChangeEmail` в `User`, не должен проверять вызовы от `User` к `Company`. Аналогичные рассуждения применимы если опуститься еще на один уровень и протестировать два метода `Company` с точки зрения `User`.

Наблюдаемое поведение и подробности реализации можно представить в виде луковицы с несколькими слоями. Каждый слой тестируется исключительно с точки зрения внешнего слоя, а его взаимодействия с нижележащими слоями игнорируются. Снимая эти слои один за другим, вы меняете точку зрения: то, что прежде было деталями имплементации, становится наблюдаемым поведением, которое затем покрывается новыми тестами.

Итоги

- Сложность кода определяется количеством точек принятия решений в коде, как явных (в самом коде), так и неявных (в библиотеках, используемых кодом).
- Важность кода показывает, насколько он значим для проекта. Сложный код часто обладает высокой важностью и наоборот, но не в 100 % случаев.
- Тестирование сложного и важного кода дает наибольшую пользу, потому что соответствующие тесты лучше всего защищают от багов.

- Юнит-тесты, покрывающие код с большим количеством коллaborаторов, требуют высоких затрат на сопровождение. Чтобы привести коллабораторы к необходимому состоянию, а затем проверить их состояние или взаимодействия с ними, придется написать код, занимающий немало места.
- Весь рабочий код можно разделить на четыре типа по сложности или важности для проекта и количеству коллабораторов:
 - модель предметной области и алгоритмы (высокая сложность или важность для проекта, мало коллабораторов) обеспечивает наибольшую эффективность юнит-тестов;
 - тривиальный код (низкая сложность или важность для проекта, мало коллабораторов) вообще не следует тестировать;
 - контроллеры (низкая сложность или важность для проекта, много коллабораторов) должны тестируться интеграционными тестами;
 - переусложенный код (высокая сложность или важность для проекта, много коллабораторов) должен разделяться на контроллеры и сложный код.
- Чем важнее или сложнее код, тем меньше у него должно быть коллабораторов.
- Паттерн «Простой объект» помогает сделать переусложенный код пригодным для тестирования за счет извлечения бизнес-логики из этого кода в отдельный класс. В результате оставшийся код становится контроллером — тонкой, *простой оберткой* над бизнес-логикой.
- Гексагональные и функциональные архитектуры реализуют паттерн «Простой объект». Гексагональная архитектура требует разделения бизнес-логики и взаимодействий с внепроцессными зависимостями. Функциональная архитектура отделяет бизнес-логику от взаимодействий со всеми коллабораторами, не только внепроцессными.
- Бизнес-логику и координацию можно рассматривать в контексте глубины и ширины кода. Ваш код может быть либо глубоким (сложным или важным), либо широким (работающим со многими коллабораторами), но никогда не должен быть и тем и другим.
- Тестируйте предусловия, если они имеют смысл с точки зрения предметной области; в противном случае тестируть их не следует.
- В том, что касается отделения бизнес-логики от координации, существуют три важных атрибута:
 - *тестируемость доменной модели*, которая зависит от количества и типа коллабораторов в классах предметной области;
 - *простота контроллера*, зависящая от присутствия точек принятия решений (ветвления) в контроллере;
 - *быстродействие*, определяемое как количество обращений к внепроцессным зависимостям.

- В любой конкретной ситуации можно достичь только двух из этих трех атрибутов:
 - *перемещение всех внешних операций чтения и записи к границам бизнес-операции* сохраняет простоту контроллера и тестируемость доменной модели, но с потерями для быстродействия;
 - *внедрение внепроцессных зависимостей в доменную модель* сохраняет быстродействие и простоту контроллера, но с потерями для тестируемости доменной модели;
 - *разбиение процесса принятия решений на более мелкие шаги* помогает с быстродействием и тестируемостью модели предметной области, но с потерями для простоты контроллера.
- *Разбиение процесса принятия решений на более мелкие шаги* — компромисс с оптимальным набором достоинств и недостатков. Рост сложности контроллера можно преодолеть при помощи следующих двух паттернов:
 - паттерн «CanExecute/Execute» вводит для каждого метода `Do()` метод `CanDo()`, успешное выполнение которого становится предусловием для `Do()`. Этот паттерн фактически исключает принятие решений из контроллера, потому что вызов `Do()` без `CanDo()` невозможен;
 - события предметной области помогают отслеживать важные изменения в модели предметной области, а затем преобразовать эти изменения в обращения к внепроцессным зависимостям. Этот паттерн избавляет контроллер от обязанностей по отслеживанию изменений.
- Тестировать абстракции проще, чем то, что они абстрагируют. Доменные события — абстракции для предстоящих обращений к внепроцессным зависимостям. Изменения в доменных классах — абстракции для предстоящих изменений в базе данных.

Часть III

Интеграционное тестирование

Вы когда-нибудь оказывались в ситуации, когда все юнит-тесты проходят, а приложение все равно не работает? Проверка компонентов в изоляции друг от друга важна, но не менее важно проверить, как эти компоненты работают в интеграции друг с другом и внешними системами. Здесь вам потребуется интеграционное тестирование.

В главе 8 мы рассмотрим интеграционное тестирование вообще и вернемся к концепции пирамиды тестирования. Вы узнаете о достоинствах и недостатках интеграционного тестирования и о том, как маневрировать между ними. Затем в главах 9 и 10 обсуждаются более конкретные темы. Глава 9 научит вас использовать моки с максимальной эффективностью. В главе 10 более глубоко рассматривается работа с реляционными базами данных в тестах.

Для чего нужно интеграционное тестирование?

В этой главе:

- ✓ Роль интеграционного тестирования.
- ✓ Более глубокий анализ концепции пирамиды тестирования.
- ✓ Написание эффективных интеграционных тестов.

Если полагаться исключительно на юнит-тесты, вы никогда не будете уверены в том, что ваша система работает. Юнит-тесты прекрасно справляются с проверкой бизнес-логики, но проверять эту логику «в вакууме» недостаточно. Необходимо проверять, как разные ее части интегрируются друг с другом и внешними системами: базой данных, шиной сообщений и т. д.

В этой главе рассматривается роль интеграционных тестов, когда их следует использовать и когда лучше положиться на классические юнит-тесты (или даже другие средства — например, принцип Fail Fast). Вы увидите, какие внепроцессные зависимости можно использовать в интеграционных тестах в неизменном виде, а какие следует заменить моками. Также будут представлены методы интеграционного тестирования, которые помогут вам улучшить качество кода в целом: четкое определение границ доменной модели, сокращение количества слоев в приложении и устранение циклических зависимостей. Наконец, вы узнаете, почему интерфейсы с единственной имплементацией должны использоваться очень осторожно и как и когда тестировать функциональность логирования.

8.1. Что такое интеграционный тест?

Интеграционные тесты играют важную роль в проекте. Также важно иметь сбалансированное количество юнит- и интеграционных тестов. Вскоре вы узнаете, что это за роль и как выдерживать этот баланс, но сначала давайте вспомним, чем интеграционные тесты отличаются от юнит-тестов.

8.1.1. Роль интеграционных тестов

Как говорилось в главе 2, юнит-тест удовлетворяет следующим трем требованиям:

- проверяет правильность работы одной единицы поведения;
- делает это быстро
- и в изоляции от других тестов.

Тест, который не удовлетворяет хотя бы одному из этих трех требований, относится к категории интеграционных тестов. Таким образом, *интеграционным* оказывается любой тест, не являющийся юнит-тестом.

На практике интеграционные тесты почти всегда проверяют, как ваша система работает в интеграции с внепроцессными зависимостями. Другими словами, эти тесты покрывают код из четверти контроллеров (за информацией о классификации кода обращайтесь к главе 7). Диаграмма на рис. 8.1 представляет типичные обязанности юнит- и интеграционных тестов. Юнит-тесты покрывают доменную модель (модель предметной области), тогда как интеграционные тесты проверяют код, связывающий доменную модель с внепроцессными зависимостями.

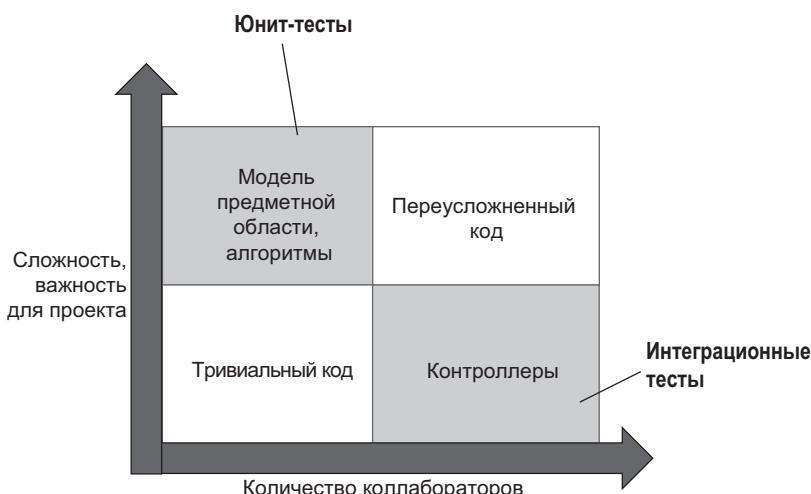


Рис. 8.1. Интеграционные тесты покрывают контроллеры, а юнит-тесты покрывают модель предметной области и алгоритмы. Тривиальный и переусложненный коды тестироваться не должны

Обратите внимание, что тесты, покрывающие четверть контроллеров, иногда также могут быть юнит-тестами. Если все внепроцессные зависимости заменить моками, никакие зависимости не будут совместно использоваться между тестами, благодаря чему эти тесты останутся быстрыми и сохранят свою изоляцию друг от друга. Тем не менее во многих приложениях существует внепроцессная зависимость, которую невозможно заменить моком. Обычно это база данных — зависимость, не видимая другими приложениями.

Как говорилось в главе 7, две другие четверти на рис. 8.1 (тривиальный и переусложненный коды) вообще не должны тестироваться. Тривиальный код не стоит затраченных усилий, тогда как переусложненный код должен быть разбит на алгоритмы и контроллеры. Следовательно, ваши тесты должны сконцентрироваться на четвертях модели предметной области и контроллеров.

8.1.2. Снова о пирамиде тестирования

Важно поддерживать баланс между юнит- и интеграционными тестами. Работа напрямую с внепроцессными зависимостями замедляет интеграционные тесты. Кроме того, их сопровождение также обходится дороже. Повышение затрат на сопровождение обусловлено:

- необходимостью поддержания внепроцессных зависимостей в работоспособном состоянии;
- большим количеством задействованных колабораторов, что приводит к увеличению размера теста.

С другой стороны, интеграционные тесты проходят через больший объем кода (как вашего, так и кода библиотек, используемых в приложении), что делает их более эффективными по сравнению с юнит-тестами в том, что касается защиты от багов. Они также более отделены от рабочего кода, а следовательно, обладают большей устойчивостью к его рефакторингу.

Соотношение между юнит- и интеграционными тестами зависит от особенностей проекта, но общее правило выглядит так: проверьте как можно больше пограничных случаев бизнес-сценария юнит-тестами; используйте интеграционные тесты для покрытия одного позитивного пути, а также всех граничных случаев, которые не покрываются юнит-тестами.

ОПРЕДЕЛЕНИЕ

Позитивный путь означает успешное выполнение бизнес-сценария. В пограничных случаях выполнение бизнес-сценария приводит к ошибке.

Перемещение основной части работы в юнит-тесты помогает удерживать затраты на сопровождение на низком уровне. В то же время наличие одного или двух

интеграционных тестов на бизнес-сценарий гарантирует правильность вашей системы в целом. Это правило формирует соотношение между количеством юнит- и интеграционных тестов, показанное на рис. 8.2 (как упоминалось в главе 2, сквозные тесты образуют подмножество интеграционных тестов).



Рис. 8.2. Пирамида тестирования представляет компромисс, который лучше всего работает в большинстве приложений. Быстрые и дешевые тесты покрывают большинство пограничных случаев, тогда как меньшее количество более медленных и затратных тестов обеспечивает правильность системы в целом

Пирамида тестирования может принимать разные формы в зависимости от сложности проекта. В простых приложениях объем кода в четверти доменной модели и алгоритмов минимален. В результате тесты образуют прямоугольник вместо пирамиды, с равным количеством юнит- и интеграционных тестов (рис. 8.3). В самых тривиальных случаях юнит-тестов может вообще не быть.

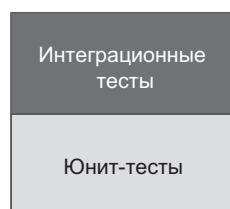


Рис. 8.3. Пирамида тестов в простом проекте. Малая сложность требует меньшего числа юнит-тестов по сравнению с обычной пирамидой

Обратите внимание: интеграционные тесты сохраняют свою полезность даже в простых приложениях. Независимо от того, насколько прост ваш код, важно проверять, как он работает в интеграции с другими подсистемами.

8.1.3. Интеграционное тестирование и принцип Fail Fast

В этом разделе более подробно рассматривается описанная выше рекомендация — использовать интеграционные тесты для покрытия одного позитивного пути на каждый бизнес-сценарий и любых пограничных случаев, которые не покрываются юнит-тестами.

Для интеграционного теста выберите самый длинный позитивный путь, проверяющий взаимодействия со всеми внепроцессными зависимостями. Если не существует одного пути, проходящего через все такие взаимодействия, напишите дополнительные интеграционные тесты — столько, сколько потребуется для отражения взаимодействий с каждой внешней системой.

Как и с пограничными случаями, которые не могут покрываться юнит-тестами, к этому правилу тоже есть свои исключения. Нет необходимости тестировать пограничный случай, если неправильное выполнение этого пограничного случая немедленно приводит к отказу всего приложения. Например, в главе 7 было показано, как класс `User` из CRM реализует метод `CanChangeEmail` и делает его успешное выполнение предусловием для `ChangeEmail()`:

```
public void ChangeEmail(string newEmail, Company company)
{
    Precondition.Requires(CanChangeEmail() == null);

    /* ... */
}
```

Контроллер вызывает `CanChangeEmail()` и прерывает операцию, если этот метод возвращает ошибку:

```
// UserController
public string ChangeEmail(int userId, string newEmail)
{
    object[] userData = _database.GetUserById(userId);
    User user = UserFactory.Create(userData);

    string error = user.CanChangeEmail();
    if (error != null) | Пограничный случай
        return error;

    /* ... */
}
```

ПРИНЦИП FAIL FAST

Принцип Fail Fast («быстрый отказ») означает, что текущая операция должна останавливаться при возникновении любой неожиданной ошибки. Этот принцип повышает стабильность вашего приложения за счет следующих факторов:

- ✓ сокращения цикла обратной связи — чем скорее вы обнаружите ошибку, тем проще ее исправить. Исправление ошибки, попавшей в продуктив, обходится на порядки дороже, чем исправление той же ошибки в ходе разработки;
- ✓ защиты состояния базы данных — ошибки могут приводить к повреждению состояния приложения. После того как поврежденное состояние проникнет в базу данных, исправить эти ошибки будет намного труднее. Быстрые отказы помогут предотвратить распространение повреждений.

Прерывание текущей операции обычно осуществляется посредством выдачи исключений, потому что семантика исключений идеально подходит для принципа быстрого отказа — они прерывают выполнение программы и переходят на наивысший уровень стека выполнения, где эти исключения можно зарегистрировать и завершить выполнение или перезапустить операцию.

Предусловия являются одним из примеров принципа Fail Fast в действии. Нарушение предусловия обозначает неправильное предположение относительно состояния приложения, и это всегда является ошибкой. Другим примером служит чтение данных из конфигурационного файла. Логику чтения можно организовать так, чтобы она выдавала исключение при неполных или некорректных данных в конфигурационном файле. Также можно разместить эту логику поближе к началу приложения, чтобы при возникновении проблем с конфигурацией приложение даже не запускалось.

Этот пример демонстрирует пограничный случай, который теоретически может покрываться интеграционным тестом. Впрочем, такой тест не будет особенно полезен. Если контроллер попытается изменить адрес электронной почты без предварительного вызова `CanChangeEmail()`, в приложении произойдет сбой. Ошибка проявится при первом выполнении, будет хорошо заметна и легко исправляема. Кроме того, она не приводит к повреждению данных.

В отличие от вызова `CanChangeEmail()` из контроллера, присутствие предусловия в `User` должно тестироваться. Однако это лучше делать с помощью юнит-теста; в интеграционном teste необходимости нет.

Немедленное проявление ошибок в программе называется принципом Fail Fast; это допустимая альтернатива интеграционному тестированию.

СОВЕТ

Лучше вообще не писать тест, чем написать плохой. Тест, который не приносит пользы, — пример такого плохого теста.

8.2. Какие из внепроцессных зависимостей должны проверяться напрямую

Как упоминалось ранее, интеграционные тесты проверяют, как ваша система интегрируется с внепроцессными зависимостями. Такая проверка может быть реализована двумя способами: с использованием реальных внепроцессных зависимостей или с заменой таких зависимостей моками. В этом разделе будет показано, когда применяется каждый из двух подходов.

8.2.1. Два типа внепроцессных зависимостей

Все внепроцессные зависимости делятся на две категории.

- *Управляемые зависимости (внепроцессные зависимости, находящиеся под вашим полным контролем)*: эти зависимости доступны только через ваше приложение; взаимодействия с ними не видны внешнему миру. Типичный пример – база данных. Внешние системы обычно не обращаются к вашей базе данных напрямую, они используют для этого API, предоставленный вашим приложением.
- *Неуправляемые зависимости (внепроцессные зависимости, которые не находятся под вашим полным контролем)* – результат взаимодействия с такими зависимостями виден извне. В качестве примеров можно привести сервер SMTP и шину сообщений: обе зависимости производят изменения, видимые для других приложений.

В главе 5 я упоминал о том, что взаимодействия с управляемыми зависимостями относятся к деталям имплементации. И наоборот, взаимодействия с неуправляемыми зависимостями являются частью наблюдаемого поведения вашей системы (рис. 8.4).

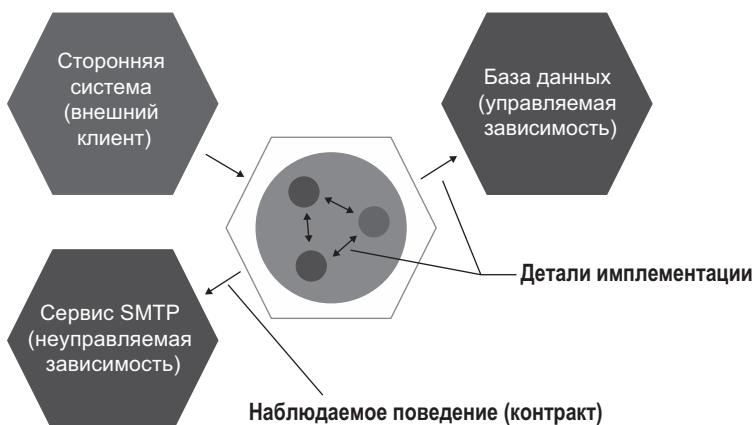


Рис. 8.4. Взаимодействия с управляемыми зависимостями являются деталями имплементации; используйте их в исходном виде в интеграционных тестах. Взаимодействия с неуправляемыми зависимостями являются частью наблюдаемого поведения вашей системы. Такие зависимости должны заменяться моками

Это различие приводит к тому, что такие зависимости по-разному обрабатываются в интеграционных тестах.

ПРЕДУПРЕЖДЕНИЕ

Используйте реальные экземпляры управляемых зависимостей; неуправляемые экземпляры заменяйте моками.

Как обсуждалось в главе 5, требование о сохранении схемы взаимодействий с неуправляемыми зависимостями обусловлено необходимостью поддержания обратной совместимости с такими зависимостями. Моки идеально подходят для этой задачи. Они позволяют обеспечить неизменность схемы взаимодействий в свете любых возможных рефакторингов.

Поддерживать обратную совместимость во взаимодействиях с управляемыми зависимостями не обязательно, потому что никто, кроме вашего приложения, с ними не работает. Внешних клиентов не интересует, как устроена ваша база данных; важно только итоговое состояние вашей системы. Использование реальных экземпляров управляемых зависимостей в интеграционных тестах помогает проверить это итоговое состояние с точки зрения внешних клиентов. Оно также упрощает рефакторинг базы данных — например, переименование столбцов или даже миграцию на другую базу данных.

8.2.2. Работа с управляемыми и неуправляемыми зависимостями

Иногда встречаются внепроцессные зависимости, обладающие свойствами как управляемых, так и неуправляемых зависимостей. Хорошим примером служит база данных, доступная для других приложений.

История обычно выглядит так: система начинает с использования собственной выделенной базы данных. Через какое-то время другая система начинает запрашивать информацию из той же базы данных. Тогда команда решает открыть общий доступ к ограниченному набору таблиц просто для удобства интеграции с другой системой. В результате база данных становится одновременно управляемой и неуправляемой зависимостью. Она все еще содержит части, видимые только для вашего приложения, но кроме них в базе также присутствуют таблицы, доступные для других приложений.

База данных — не лучший механизм для интеграции между системами, потому что она связывает эти системы друг с другом и усложняет дальнейшую их разработку. Используйте это решение только в случае, если других вариантов нет. Правильнее осуществлять интеграцию через API (для синхронных взаимодействий) или шину сообщений (для асинхронных взаимодействий).

Но что делать, если у вас уже имеется общая база данных, и вы не можете ничего с этим поделать в ближайшем будущем? В этом случае рассматривайте таблицы, видимые для

других приложений, как неуправляемую зависимость. Такие таблицы фактически выполняют функции шины сообщений, а их строки играют роль сообщений. Используйте моки, чтобы гарантировать неизменность схемы взаимодействий с этими таблицами. В то же время рассматривайте остальные части базы данных как управляемую зависимость и проверяйте ее итоговое состояние, а не взаимодействия с ней (рис. 8.5).

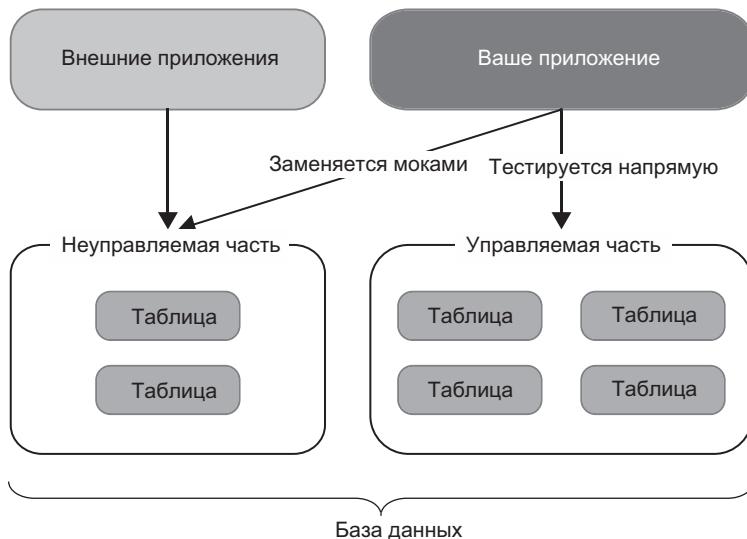


Рис. 8.5. Часть базы данных, видимая для внешних приложений, рассматривается как неуправляемая зависимость. В интеграционных тестах она заменяется моками. Оставшаяся часть базы данных рассматривается как управляемая зависимость. Проверяйте ее итоговое состояние, а не взаимодействия с ней

Важно различать эти две части вашей базы данных, так как общие таблицы видимы извне, и вы должны внимательно следить за тем, как ваше приложение взаимодействует с ними. Не изменяйте механизм взаимодействия вашей системы с этими таблицами без необходимости. Невозможно предсказать, как другие приложения отреагируют на такое изменение.

8.2.3. Что делать, если вы не можете использовать реальную базу данных в интеграционных тестах?

Иногда по причинам, вам неподконтрольным, реальная версия управляемой зависимости не может использоваться в интеграционных тестах. Пример — унаследованная база данных, которая не может быть развернута в среде автоматизированного тестирования, не говоря уже о машине разработчика, — например, из-за политики безопасности ИТ или из-за неприемлемо высоких затрат на настройку и сопровождение тестового экземпляра базы данных.

Что делать в такой ситуации? Заменить базу данных моком, несмотря на то что она является управляемой зависимостью? Нет, потому что замена управляемой зависимости на мок снижает устойчивость интеграционных тестов к рефакторингу. Более того, такие тесты уже не обеспечивают хорошей защиты от багов. А если база данных является единственной внепроцессной зависимостью в вашем проекте, то полученные интеграционные тесты не будут обеспечивать дополнительной защиты по сравнению с существующими юнит-тестами (предполагаем, что эти юнит-тесты следуют рекомендациям из главы 7).

Единственное, что сделают такие интеграционные тесты (помимо юнит-тестов), — они проверят, какие методы базы данных вызываются контроллером. Иначе говоря, вы не получите никакой дополнительной уверенности ни в чем, кроме правильности пары строк кода из контроллера, при том что для этого вам нужно будет проделать немало работы.

Если базу данных невозможно протестировать «как есть», вообще не пишите интеграционные тесты — вместо этого лучше сосредоточьтесь на юнит-тестировании модели предметной области. Всегда тщательно анализируйте свои тесты. Тестам, не обладающим достаточно высокой эффективностью, нет места в вашем проекте.

8.3. Интеграционное тестирование: пример

Вернемся к примеру системы CRM из главы 7 и посмотрим, как покрыть ее интеграционными тестами. Напомню, что эта система реализует всего одну функциональность: изменение адреса электронной почты пользователя. Она читает информацию пользователя и компании из базы данных, делегирует принятие решений модели предметной области, после чего сохраняет результаты в базе данных и передает сообщение по шине при необходимости (рис. 8.6).

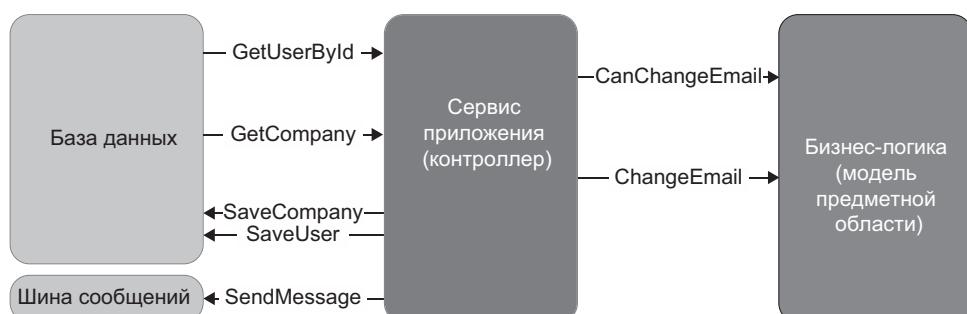


Рис. 8.6. Бизнес-сценарий изменения имейла. Контроллер координирует работу между базой данных,шиной сообщений и доменной моделью

В листинге 8.1 показано, как в настоящее время выглядит контроллер.

Листинг 8.1. Контроллер

```
public class UserController
{
    private readonly Database _database = new Database();
    private readonly MessageBus _messageBus = new MessageBus();

    public string ChangeEmail(int userId, string newEmail)
    {
        object[] userData = _database.GetUserById(userId);
        User user = UserFactory.Create(userData);

        string error = user.CanChangeEmail();
        if (error != null)
            return error;

        object[] companyData = _database.GetCompany();
        Company company = CompanyFactory.Create(companyData);

        user.ChangeEmail(newEmail, company);

        _database.SaveCompany(company);
        _database.SaveUser(user);
        foreach (EmailChangedEvent ev in user.EmailChangedEvents)
        {
            _messageBus.SendEmailChangedMessage(ev.UserId, ev.NewEmail);
        }

        return "OK";
    }
}
```

В следующем разделе я сначала опишу сценарии, которые должны проверяться с использованием интеграционных тестов, а затем покажу, как работать с базой данных и шиной сообщений в тестах.

8.3.1. Какие сценарии тестировать?

Как упоминалось ранее, общая рекомендация для интеграционного тестирования заключается в том, чтобы покрыть самый длинный позитивный путь и любые граничные случаи, которые не могут отрабатываться юнит-тестами. Самый длинный позитивный путь — тот, который проходит через все внепроцессные зависимости.

В проекте CRM самым длинным позитивным путем является замена корпоративного имейла некорпоративным. Такое изменение приводит к максимальному количеству побочных эффектов:

- в базе данных обновляются как данные пользователя, так и данные компании: пользователь изменяет свой тип (с корпоративного на обычный), а компания изменяет свое количество работников;
- передается сообщение на шину сообщений.

Что касается пограничных случаев, не тестируемых юнит-тестами, существует только один такой случай: сценарий, в котором адрес не может быть изменен. Тем не менее в тестировании этого сценария нет необходимости, потому что при отсутствии такой проверки в контроллере в приложении произойдет быстрый отказ. Остается всего один интеграционный тест:

```
public void Changing_email_from_corporate_to_non_corporate()
```

8.3.2. Классификация базы данных и шины сообщений

Прежде чем писать интеграционные тесты, необходимо классифицировать две внепроцессные зависимости и решить, какую из них следует тестиировать напрямую, а какую — заменить моком. База данных приложения является управляемой зависимостью, потому что никакая другая система не может обращаться к ней. А следовательно, вам следует использовать ее реальный экземпляр. Интеграционный тест должен:

- вставить данные пользователя и компании в базу данных;
- выполнить сценарий изменения имейла с этой базой данных;
- проверить состояние базы данных.

С другой стороны, шина сообщений является неуправляемой зависимостью — она нужна исключительно для обеспечения взаимодействия с другими системами. Интеграционный тест заменит шину сообщений моком, после чего проверит взаимодействия между контроллером и этим моком.

8.3.3. Как насчет сквозного тестирования?

В нашем примере не будет ни одного сквозного (end-to-end) теста. Сквозной тест в сценарии с API станет тестом, применяемым к развернутой, полнофункциональной версии этого API, что означает отсутствие моков для любых внепроцессных зависимостей (рис. 8.7). С другой стороны, при интеграционных тестах приложение размещается в том же процессе, а неуправляемые зависимости заменяются моками (рис. 8.8).

Как упоминалось в главе 2, решение о том, нужно ли использовать сквозные тесты, принимается на основании здравого смысла. Как правило, при включении управляемых зависимостей в зону интеграционного тестирования и замене моками только неуправляемых зависимостей интеграционные тесты предоставляют уровень защиты, близкий к уровню защиты сквозных тестов, так что без сквозного тестирования можно обойтись. Впрочем, вы можете создать один или два общих сквозных теста, которые обеспечат проверку работоспособности проекта после развертывания. Многие такие тесты также проходят по самому длинному позитивному пути,

чтобы гарантировать, что ваше приложение правильно взаимодействует со всеми внепроцессными зависимостями. Чтобы эмулировать поведение внешнего клиента, сквозные тесты должны проверять шину сообщений напрямую, но состояние базы данных лучше проверять через само приложение.

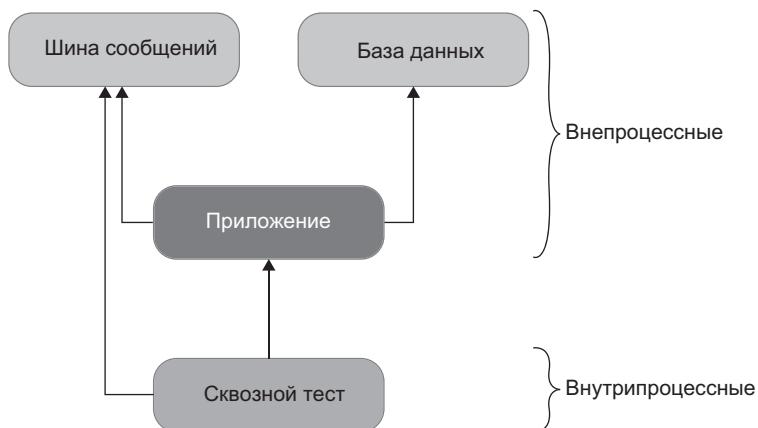


Рис. 8.7. Сквозные тесты эмулируют внешнего клиента и, как следствие, проверяют развернутую версию приложения вместе со всеми внепроцессными зависимостями, включаемыми в сферу тестирования. Сквозные тесты не должны проверять управляемые зависимости (например, базы данных) напрямую — только косвенно, через приложение

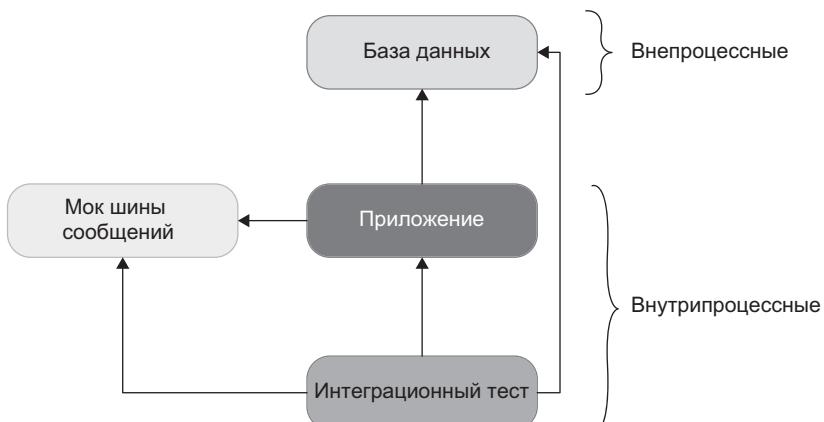


Рис. 8.8. Интеграционные тесты размещают приложение в том же процессе. В отличие от сквозных тестов, они заменяют неуправляемые зависимости моками. Единственными внепроцессными компонентами для интеграционных тестов являются управляемые зависимости

8.3.4. Интеграционное тестирование: первая версия

В листинге 8.2 приведена первая версия интеграционного теста.

Листинг 8.2. Интеграционный тест

```
[Fact]
public void Changing_email_from_corporate_to_non_corporate()
{
    // Arrange
    var db = new DatabaseConnectionString(); ← База данных
    User user = CreateUser(← Создает пользователя
        "user@mycorp.com", UserType.Employee, db); и компанию в базе
    CreateCompany("mycorp.com", 1, db); данных

    var messageBusMock = new Mock<IMessageBus>(); ← Настраивает
    var sut = new UserController(db, messageBusMock.Object); мок для шины
                                                               сообщений

    // Act
    string result = sut.ChangeEmail(user.UserId, "new@gmail.com");

    // Assert
    Assert.Equal("OK", result);

    object[] userData = db.GetUserById(user.UserId); ← Проверяет
    User userFromDb = UserFactory.Create(userData); состояние
    Assert.Equal("new@gmail.com", userFromDb.Email); пользователя
    Assert.Equal(UserType.Customer, userFromDb.Type);

    object[] companyData = db.GetCompany(); ← Проверяет
    Company companyFromDb = CompanyFactory
        .Create(companyData); состояние
    Assert.Equal(0, companyFromDb.NumberOfEmployees); компании

    messageBusMock.Verify(← Проверяет
        x => x.SendEmailChangedMessage(
            user.UserId, "new@gmail.com"),
        Times.Once);
}
```

Очень важно проверять состояние базы данных независимо от данных, использованных как входные параметры. Для этого интеграционный тест отдельно запрашивает информацию о пользователе и компании в секции проверки, создает новые экземпляры `userFromDb` и `companyFromDb` и только потом проверяет их состояние. Такой подход гарантирует, что тест проверит как запись, так и чтение из базы данных, и таким образом обеспечивает максимальную защиту от багов. Само чтение должно быть выполнено с использованием того же кода, который используется во внутренней реализации контроллера; в данном примере это код классов `Database`, `UserFactory` и `CompanyFactory`.

СОВЕТ

Обратите внимание: в секции подготовки тест не вставляет информацию пользователя и компании в базу данных самостоятельно, а вместо этого вызывает вспомогательные методы `CreateUser` и `CreateCompany`. Эти методы могут переиспользоваться в нескольких интеграционных тестах.

Хотя этот интеграционный тест справляется со своим делом, его можно улучшить. Например, вы можете воспользоваться вспомогательными методами также и в секции проверки для сокращения размера секции. Кроме того, `messageBusMock` не предоставляет максимально возможной защиты от багов. Эти улучшения будут рассмотрены в следующих двух главах, когда речь пойдет о моках и практиках тестирования базы данных.

8.4. Использование интерфейсов для абстрагирования зависимостей

Одна из самых неверно понимаемых тем в сфере юнит-тестирования — использование интерфейсов. Разработчики часто пользуются интерфейсами по неверным причинам и в результате используют эти интерфейсы слишком часто. В этом разделе я подробнее расскажу об этих неверных причинах и покажу, в каких обстоятельствах стоит (и не стоит) использовать интерфейсы.

8.4.1. Интерфейсы и слабая связность

Многие разработчики пишут интерфейсы для внепроцессных зависимостей (например, баз данных или шин сообщений), даже если эти интерфейсы имеют только одну реализацию. В наши дни эта практика стала настолько распространенной, что никто даже не оспаривает ее. В коде часто встречаются пары «класс — интерфейс», которые выглядят примерно так:

```
public interface IMessageBus  
public class MessageBus : IMessageBus  
  
public interface IUserRepository  
public class UserRepository : IUserRepository
```

Обычное объяснение для таких интерфейсов заключается в том, что они помогают:

- абстрагировать внепроцессные зависимости и таким образом достигнуть слабой связности (*loose coupling*);
- добавлять новую функциональность без изменения существующего кода в соответствии с принципом открытости/закрытости (*OCP, Open-Closed Principle*).

Обе причины ошибочны. Интерфейсы с одной реализацией не являются абстракциями и способствуют слабой связности не более чем конкретные классы, реализующие эти интерфейсы. Подлинные абстракции *открываются, не изобретаются*. Открытие по определению происходит тогда, когда абстракция уже существует, но еще не имеет четкого места в коде. Таким образом, чтобы интерфейс был полноценной абстракцией, он должен иметь как минимум две реализации.

Вторая причина (возможность добавления новой функциональности без изменения существующего кода) также является заблуждением, потому что она нарушает более фундаментальный принцип: *YAGNI*. Сокращение YAGNI означает «*You aren't gonna need it*» (то есть «*Вам это не понадобится*»); этот принцип предписывает не тратить время на функциональность, которая не нужна прямо сейчас. Вам не следует ни разрабатывать такую функциональность, ни изменять существующий код с расчетом на появление такой функциональности в будущем. Две основные причины для такого подхода:

- *Упущенные возможности.* Если вы тратите время на функциональность, которая прямо сейчас не нужна бизнесу, это время будет отнято у действительно нужной функциональности — той, которая нужна в данный момент. Кроме того, когда бизнес попросит эту функциональность, требования к ней, с большой вероятностью, уже изменятся, и вам придется вносить изменения в написанный код. Такая работа неэффективна. Лучше реализовать функциональность «с нуля», когда возникнет непосредственная необходимость в ней.
- *Чем меньше кода в проекте, тем лучше.* Написание кода «на всякий случай», без непосредственной необходимости, повышает стоимость поддержки кода приложения. Лучше отложить написание новой функциональности настолько, насколько это возможно.

СОВЕТ

Написание кода — дорогостоящий способ решения задач. Чем меньше кода требует решение и чем проще этот код, тем лучше.

Существуют особые ситуации, в которых принцип YAGNI неприменим, но они встречаются редко. Для того чтобы прочитать о таких ситуациях, обращайтесь к моей статье «*OCP vs YAGNI*» (<https://enterprise-craftsmanship.com/posts/ocp-vs-yagni>).

8.4.2. Зачем использовать интерфейсы для внепроцессных зависимостей?

Итак, зачем вообще использовать интерфейсы для внепроцессных зависимостей, если каждый из этих интерфейсов имеет всего одну реализацию? Настоящая причина оказывается намного более практической и приземленной. Это делается для того,

чтобы сделать возможным использование моков. Без интерфейса вы не сможете создать тестовую заглушку, а следовательно, проверить взаимодействия между тестируемой системой и внепроцессной зависимостью.

Не пишите интерфейсы для внепроцессных зависимостей, если только вам не нужно заменять эти зависимости на моки. Так как моки должны использоваться только для неуправляемых зависимостей, то эта рекомендация сводится к следующему: *используйте интерфейсы только для неуправляемых зависимостей*. Управляемые зависимости по-прежнему следует внедрять в контроллер явно, но для этого следует использовать конкретные классы.

Обратите внимание, что полноценные абстракции (абстракции, имеющие более одной реализации) могут иметь интерфейсы независимо от того, заменяются они моками или нет. При этом добавление интерфейса с единственной реализацией по любым другим причинам, кроме мокирования, является нарушением принципа YAGNI.

Вы, возможно, обратили внимание, что в листинге 8.2 `UserController` теперь явно получает как шину сообщений, так и базу данных в конструкторе, но только шина сообщений имеет соответствующий интерфейс. База данных является управляемой зависимостью, а следовательно, не требует такого интерфейса. Код контроллера выглядит так:

```
public class UserController
{
    private readonly Database _database;
    private readonly IMessageBus _messageBus;

    public UserController(Database database, IMessageBus messageBus)
    {
        _database = database;
        _messageBus = messageBus;
    }

    public string ChangeEmail(int userId, string newEmail)
    {
        /* метод использует _database и _messageBus */
    }
}
```



ПРИМЕЧАНИЕ

Вы можете заменить зависимость на мок, не прибегая к интерфейсам, путем объявления методов в ней виртуальными и используя сам класс в качестве базового для мока. Такое решение хуже решения с интерфейсами. Тема сравнения интерфейсов с базовыми классами будет рассмотрена в главе 11.

8.4.3. Использование интерфейсов для внутрипроцессных зависимостей

Иногда встречается код, в котором интерфейсы добавляются не только для внепроцессных, но и внутрипроцессных зависимостей. Например:

```
public interface IUser
{
    int UserId { get; set; }
    string Email { get; }
    string CanChangeEmail();
    void ChangeEmail(string newEmail, Company company);
}

public class User : IUser
{
    /* ... */
}
```

Если предположить, что `IUser` имеет только одну реализацию (а такие подробные интерфейсы всегда имеют только одну реализацию), это становится серьезным индикатором проблем в коде. Как и в случае с внепроцессными зависимостями, единственной причиной для добавления интерфейса с единственной реализацией для доменного класса является возможность использования моков. Но в отличие от внепроцессных зависимостей вы никогда не должны проверять взаимодействия между классами предметной области, потому что это приводит к появлению хрупких тестов: тестов, связанных с деталями имплементации, а следовательно, не устойчивых к рефакторингу (за дополнительной информацией о моках и хрупкости тестов обращайтесь к главе 5).

8.5. Основные приемы интеграционного тестирования

Существует несколько общих рекомендаций, которые помогут вам извлечь максимальную пользу из интеграционных тестов:

- явное определение границ доменной модели (модели предметной области);
- сокращение количества слоев в приложении;
- устранение циклических зависимостей.

Как обычно, лучшие практики, приносящие пользу для тестов, также улучшают поддерживаемость кода приложения в целом.

8.5.1. Явное определение границ модели предметной области

Всегда старайтесь создавать явное, четко определенное место для доменной модели в вашей кодовой базе. Доменная модель представляет собой совокупность знаний о предметной области задачи, для решения которой предназначен ваш проект.

Определение четких границ доменной модели помогает визуализировать эту часть кода и рассуждать о ней.

Данная практика также помогает с тестированием. Как упоминалось ранее в этой главе, юнит-тесты должны ориентироваться на доменную модель и алгоритмы, тогда как интеграционные тесты — на контроллеры. Таким образом, четкое разграничение между доменными классами и контроллерами также помогает отделить юнит-тесты от интеграционных.

Сама граница может быть представлена в виде отдельной сборки или пространства имен. Конкретные детали не столь важны, при условии что вся логика предметной области размещена в одном месте с четко определенными границами и не рассеяна по всему коду.

8.5.2. Сокращение количества слоев

Многие программисты стремятся к абстрагированию и обобщению кода путем введения дополнительных уровней абстракции. В типичном корпоративном приложении можно легко найти несколько таких уровней (также называемых слоями) (рис. 8.9).

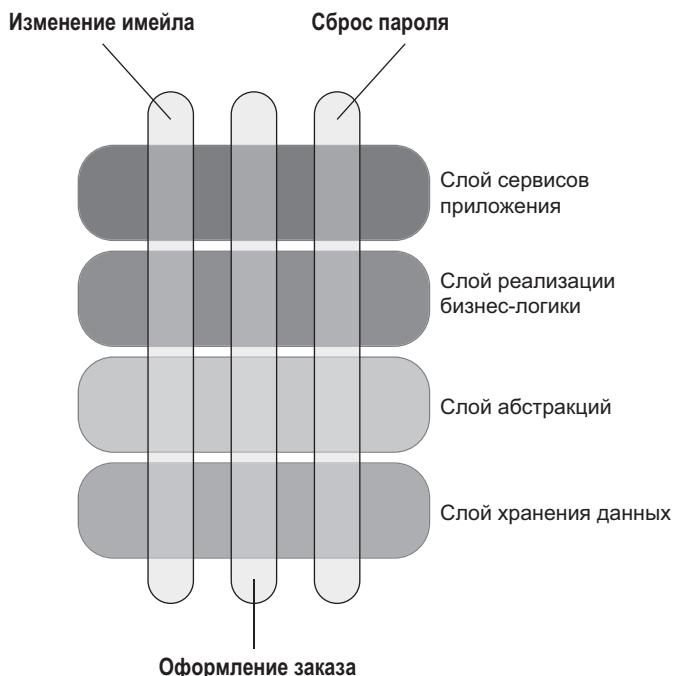


Рис. 8.9. Различные задачи часто решаются разными слоями. Типичная функциональность обычно занимает небольшую часть каждого слоя

В некоторых приложениях находится столько уровней абстракции, что разработчик уже не может разобраться в коде и понять логику даже простейших операций. В какой-то момент вы просто хотите добраться до конкретного решения вашей задачи, а не ее обобщенного решения «в вакууме».

«Все проблемы в программировании можно решить путем добавления нового уровня абстракции (кроме проблемы наличия слишком большого количества уровней абстракции)».

Дэвид Дж. Уилер

Уровни абстракции отрицательно влияют на вашу способность понимать код. Когда каждая функция представлена на каждом из этих уровней, вам придется потратить намного больше усилий, чтобы собрать все фрагменты в целостную картину. Таким образом, создается лишняя когнитивная нагрузка, которая тормозит весь процесс разработки.

Лишние абстракции также затрудняют юнит- и интеграционное тестирование. Кодовые базы со слишком большим количеством слоев обычно не имеют четкой границы между контроллерами и моделью предметной области (что, как говорилось в главе 7, является необходимым условием для эффективного тестирования). Также существует намного более сильная тенденция к раздельной проверке каждого слоя. Эта тенденция приводит к большому количеству интеграционных тестов, обладающих низкой эффективностью, где каждый из тестов проверяет только код конкретного слоя и заменяет моками нижележащие слои. Конечный результат всегда один: недостаточная защита от багов в сочетании с низкой устойчивостью к рефакторингу.

Старайтесь ограничиться минимально возможным количеством уровней абстракции. В большинстве серверных систем можно обойтись всего тремя: слоем доменной модели, слоем сервисов приложения (контроллеров) и слоем инфраструктуры. Слой инфраструктуры обычно состоит из алгоритмов, которые не принадлежат доменной модели, а также кода, обеспечивающего доступ к внепроцессным зависимостям (рис. 8.10).

8.5.3. Исключение циклических зависимостей

Еще одна практика, которая может улучшить сопровождаемость вашего кода и упростить тестирование, — исключение циклических зависимостей.

ОПРЕДЕЛЕНИЕ

Циклическая зависимость возникает в том случае, если два или более класса прямо или косвенно зависят друг от друга.

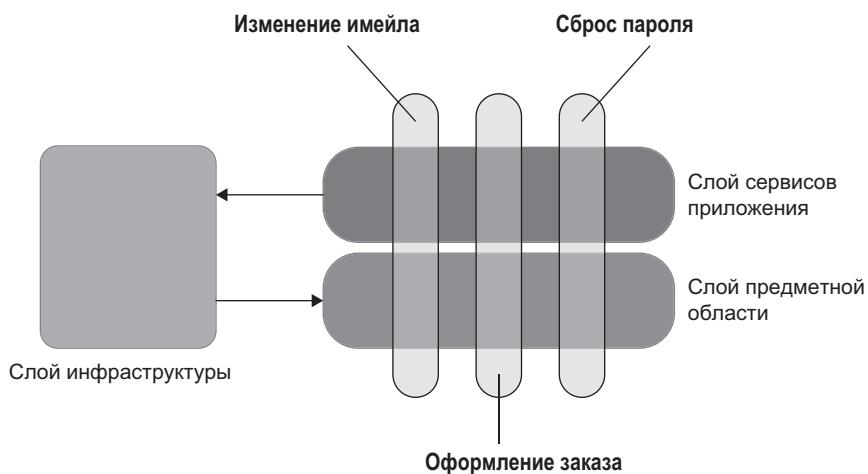


Рис. 8.10. Достаточно всего трех слоев: слоя предметной области (содержит доменную логику), уровня сервисов приложения (представляет точку входа для внешнего клиента и координирует работу между доменными классами и внепроцессными зависимостями) и слоя инфраструктуры (работает с внепроцессными зависимостями; на этом уровне находятся репозитории базы данных, настройки ORM, SMTP-шлюзы и т. п.)

Типичный пример циклической зависимости — обратный вызов:

```
public class CheckOutService
{
    public void CheckOut(int orderId)
    {
        var service = new ReportGenerationService();
        service.GenerateReport(orderId, this);
        /* остальной код */
    }
}

public class ReportGenerationService
{
    public void GenerateReport(
        int orderId,
        CheckOutService checkOutService)
    {
        /* вызывает checkOutService при завершении генерирования */
    }
}
```

Здесь `CheckOutService` создает экземпляр `ReportGenerationService` и передает себя этому экземпляру как аргумент. `ReportGenerationService` обращается с обратным вызовом к классу `CheckOutService`, чтобы уведомить его о результате генерирования отчета.

Как и в случае с избыточными уровнями абстракции, циклические ссылки создают дополнительную когнитивную нагрузку при попытке прочитать и понять код. Дело в том, что циклические зависимости не дают четкой отправной точки, с которой вы можете начать чтение кода. Чтобы понять всего один класс, необходимо прочитать и понять сразу весь график его соседей. Даже небольшой набор взаимозависимых классов быстро становится слишком сложным для понимания.

Циклические зависимости также усложняют тестирование. Вам часто приходится использовать интерфейсы и моки, для того чтобы разбить график классов и изолировать одну единицу поведения, что, как уже было сказано, недопустимо, когда дело доходит до тестирования доменной модели (подробнее об этом в главе 5).

Обратите внимание, что использование интерфейсов только маскирует проблему циклических зависимостей. Если добавить интерфейс для `CheckOutService` и заставить `ReportGenerationService` зависеть от этого интерфейса вместо конкретного класса, можно устраниТЬ циклическую зависимость на стадии компиляции (рис. 8.11), но цикл продолжит существовать на стадии выполнения. Даже при том что компилятор уже не рассматривает эту структуру классов как циклическую ссылку, когнитивная нагрузка, необходимая для понимания этого кода, не уменьшится. Она только увеличится из-за дополнительного интерфейса.

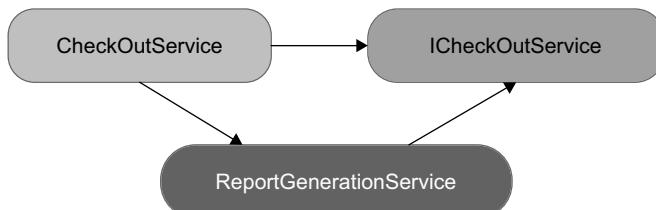


Рис. 8.11. С интерфейсом циклическая зависимость устраняется на стадии компиляции, но не на стадии выполнения. Когнитивная нагрузка, необходимая для понимания кода, от этого не уменьшается

Что же делать с циклическими зависимостями? Лучше всего совсем избавиться от них. Отрефакторите класс `ReportGenerationService`, чтобы он не зависел ни от `CheckOutService`, ни от интерфейса `ICheckOutService`, и сделайте так, чтобы `ReportGenerationService` возвращал результат работы в виде простого значения вместо вызова `CheckOutService`:

```
public class CheckOutService
{
    public void CheckOut(int orderId)
    {
        var service = new ReportGenerationService();
        Report report = service.GenerateReport(orderId);
        /* прочая работа */
    }
}
```

```
}

public class ReportGenerationService
{
    public Report GenerateReport(int orderId)
    {
        /* ... */
    }
}
```

Редко удается полностью устраниТЬ все циклические зависимости в коде. Но даже тогда можно свести ущерб к минимуму, уменьшив оставшиеся графы взаимозависимых классов настолько, насколько это возможно.

8.5.4. Использование нескольких секций действий в тестах

Как вы, возможно, помните из главы 3, наличие более одной секции подготовки, действий или проверки в тесте — плохой признак. Он указывает на то, что тест проверяет несколько единиц поведения, что, в свою очередь, ухудшает сопровождаемость теста. Например, если у вас имеются два связанных сценария использования (допустим, регистрация и удаление пользователя), возникает искушение проверить оба сценария в одном интеграционном тесте. Такой тест мог бы обладать следующей структурой:

- *подготовка* — подготовка данных для регистрации пользователя;
- *действие* — вызов `UserController.RegisterUser()`;
- *проверка* — запрос к базе данных для проверки успешного завершения регистрации;
- *действие* — вызов `UserController.DeleteUser()`;
- *проверка* — запрос к базе данных для проверки успешного удаления.

Такой подход выглядит разумно, потому что состояния пользователя естественным образом переходят из одного в другое, а первое действие (регистрация пользователя) может одновременно служить фазой подготовки для следующего действия (удаление пользователя). Проблема в том, что такие тесты теряют фокус и быстро становятся слишком большими.

Лучше всего разбить тест, выделив каждое действие в отдельный тест. На первый взгляд это может показаться лишней работой (в конце концов, зачем создавать два теста там, где хватит одного?), но эта работа окупается в долгосрочной перспективе. Фокусировка каждого теста на одной единице поведения упрощает понимание и изменение этих тестов при необходимости.

Искключение из этой рекомендации составляют тесты, работающие с внепроцессными зависимостями, трудно приводимыми в нужное состояние. Например, допустим,

что регистрация пользователя приводит к созданию банковского счета во внешней банковской системе. Банк предоставил вашей организации тестовую среду, которую вы хотите использовать для сквозных тестов. К сожалению, тестовая среда работает слишком медленно; также возможно, что банк ограничивает количество обращений к этой тестовой среде. В таком сценарии удобнее объединить несколько действий в один тест, чтобы сократить количество взаимодействий с проблемной внепроцессной зависимостью.

Такие проблемные внепроцессные зависимости — единственная причина сделать исключение из правила и написать тесты, содержащие несколько секций действий. Именно по этой причине юнит-тест никогда не должен содержать несколько действий — юнит-тесты не работают с внепроцессными зависимостями. Даже интеграционные тесты редко могут содержать множественные действия. На практике многошаговые тесты почти всегда принадлежат к категории сквозных.

8.6. Тестирование функциональности логирования

В области логирования существует немало неопределенностей. Не совсем ясно, что делать с этой функциональностью, когда речь заходит о тестировании.

Я разобью эту сложную тему на следующие вопросы:

- Нужно ли вообще тестировать функциональность логирования?
- Если нужно, то как именно?
- Какой объем логирования можно считать достаточным?
- Как передавать экземпляры логера?

В качестве примера будет использован наш проект CRM.

8.6.1. Нужно ли тестировать функциональность логирования?

Логирование — сквозная функциональность, которая может потребоваться в любой части вашего кода. Пример логирования в классе `User` представлен в листинге 8.3.

Класс `User` регистрирует в файле журнала каждое начало и завершение метода `ChangeEmail`, а также изменение типа пользователя. Нужно ли тестировать эту функциональность?

С одной стороны, логирование дает важную информацию о поведении приложения. Но с другой стороны, эта функциональность может встречаться так часто, что становится неясно — заслуживает ли она дополнительной, притом довольно значительной работы по тестированию?

Листинг 8.3. Пример логирования в User

```

public class User
{
    public void ChangeEmail(string newEmail, Company company)
    {
        _logger.Info( ← Начало метода
            $"Changing email for user {UserId} to {newEmail}");

        Precondition.Requires(CanChangeEmail() == null);

        if (Email == newEmail)
            return;

        UserType newType = company.IsEmailCorporate(newEmail)
            ? UserType.Employee
            : UserType.Customer;

        if (Type != newType)
        {
            int delta = newType == UserType.Employee?1:-1;
            company.ChangeNumberOfEmployees(delta);
            _logger.Info(
                $"User {UserId} changed type " + | Изменяет тип
                $"from {Type} to {newType}"); | пользователя
        }

        Email = newEmail;
        Type = newType;
        EmailChangedEvents.Add(new EmailChangedEvent(UserId, newEmail));

        _logger.Info( ← Конец метода
            $"Email is changed for user {UserId}");
    }
}

```

Ответ на вопрос о том, стоит ли тестировать функциональность логирования, сводится к следующему: *является ли протоколирование частью наблюдаемого поведения приложения или же это деталь имплементации?*

В этом смысле она не отличается от любой другой функциональности. Логирование создает изменения во внепроцессных зависимостях (таких как текстовый файл или база данных). Если эти изменения видны вашему заказчику, клиентам приложения или кому-то еще, кроме самих разработчиков, то логирование является наблюдаемым поведением, а следовательно, должно тестироваться. Если же единственной аудиторией являются разработчики, то это детали имплементации, которые можно изменять так, что этого никто не заметит; в этом случае оно не должно тестироваться.

Например, если вы пишете библиотеку логирования, то логи, создаваемые этой библиотекой, являются самой важной (и единственной) частью ее наблюдаемого

поведения. Другой пример — когда бизнес настаивает на логировании ключевых рабочих процессов приложения. В этом случае логи также становятся бизнес-требованиями, а следовательно, должны покрываться тестами. Тем не менее в этом случае также может быть реализовано отдельное логирование только для разработчиков.

Стив Фримен (Steve Freeman) и Нэт Прайс (Nat Pryce) в своей книге «Growing Object-Oriented Software, Guided by Tests» (Addison-Wesley Professional, 2009) называют эти два типа *служебным (support)* и *диагностическим (diagnostic)* логированием:

- служебное логирование создает сообщения, которые должны отслеживаться службой поддержки или системными администраторами;
- диагностическое логирование помогает разработчикам понять, что происходит внутри приложения.

8.6.2. Как тестировать функциональность логирования?

Так как в логировании задействованы внепроцессные зависимости, к тестированию здесь применяются те же правила, что и для любой другой функциональности, обращающейся к внепроцессной зависимости. Для проверки взаимодействий между приложением и хранилищем логов необходимо использовать моки.

Создание обертки поверх ILogger

Использовать мок для интерфейса `ILogger` было бы недостаточно. Так как служебное логирование является бизнес-требованием, это требование должно быть явно отражено в коде. Создайте специальный класс `DomainLogger`, в котором явно перечисляется все служебное логирование, необходимое бизнесу; проверяйте взаимодействия с этим классом вместо низкоуровневого `ILogger`.

Допустим, что бизнес требует регистрировать все изменения в типах пользователей, а логирование в начале и в конце метода используется исключительно для отладочных целей. В листинге 8.4 приведен класс `User` после добавления класса `DomainLogger`.

Листинг 8.4. Выделение служебного логирования в класс DomainLogger

```
public void ChangeEmail(string newEmail, Company company)
{
    _logger.Info( ← Диагностическое логирование
        $"Changing email for user {UserId} to {newEmail}");

    Precondition.Requires(CanChangeEmail() == null);

    if (Email == newEmail)
        return;
```

```
UserType newType = company.IsEmailCorporate(newEmail)
    ? UserType.Employee
    : UserType.Customer;

if (Type != newType)
{
    int delta = newType == UserType.Employee ? 1 : -1;
    company.ChangeNumberOfEmployees(delta);
    _domainLogger.UserTypeHasChanged( | Служебное
        UserId, Type, newType); | логирование
}

Email = newEmail;
Type = newType;
EmailChangedEvents.Add(new EmailChangedEvent(UserId, newEmail));

_logger.Info( ←———— Диагностическое логирование
    $"Email is changed for user {UserId}");
}
```

Для диагностического логирования используется старая реализация диспетчера протоколирования (с типом `ILogger`), но для служебного протоколирования теперь используется новый экземпляр `domainLogger` типа `IDomainLogger`. В листинге 8.5 приведена реализация `IDomainLogger`.

Листинг 8.5. DomainLogger как обертка для ILogger

```
public class DomainLogger : IDomainLogger
{
    private readonly ILogger _logger;

    public DomainLogger(ILogger logger)
    {
        _logger = logger;
    }

    public void UserTypeHasChanged(
        int userId, UserType oldType, UserType newType)
    {
        _logger.Info(
            $"User {userId} changed type " +
            $"from {oldType} to {newType}");
    }
}
```

Обертка `DomainLogger` работает поверх `ILogger`: она использует язык предметной области для объявления конкретных записей, требуемых бизнесом, что упрощает понимание и сопровождение служебного протоколирования. Эта реализация очень похожа на концепцию структурированного логирования, которая обеспечивает большую гибкость при последующей обработке и анализе логов.

Структурированное логирование

Структурированное логирование — метод логирования, при котором сохранение лог-данных отделено от отображения этих данных. Традиционное логирование работает с простым текстом. Вызов вида

```
logger.Info("User Id is " + 12);
```

сначала формирует строку, а затем записывает ее в хранилище. Недостаток такого подхода заключается в том, что отсутствие структуры усложняет анализ полученных файлов. Например, вы не сможете легко определить, сколько сообщений определенного типа сохранено в файле и сколько из этих сообщений относится к конкретному идентификатору пользователя. Для этого придется пользоваться специальными утилитами (или даже написать их самостоятельно).

С другой стороны, структурированное логирование формирует структуру для хранилища логов. На первый взгляд, использование библиотеки структурированного логирования выглядит очень похоже:

```
logger.Info("User Id is {UserId}", 12);
```

Однако поведение существенно отличается. Внутри этот метод вычисляет хеш-код шаблона сообщения (сам шаблон находится в хеш-таблице для оптимизации занимаемого места) и объединяет его со входными параметрами, формируя таким образом лог-данные. Следующим шагом становится генерирование представления. Можно использовать неструктуренный файл, как и при традиционном логировании, но это всего лишь один из возможных вариантов. Также можно настроить библиотеку для генерирования данных в формате JSON или CSV, в котором его будет проще анализировать (рис. 8.12).

Класс `DomainLogger` из листинга 8.5 формально не является структурированным логером, но работает похожим образом. Еще раз взгляните на метод:

```
public void UserTypeHasChanged(
    int userId, UserType oldType, UserType newType)
{
    _logger.Info(
        $"User {userId} changed type " +
        $"from {oldType} to {newType}");
}
```

`UserTypeHasChanged()` можно рассматривать как хеш-код шаблона сообщения. В сочетании с параметрами `userId`, `oldType` и `newType` этот хеш-код формирует лог-данные. На основании лог-данных метод создает неструктуренный файл. Кроме того, можно легко создать дополнительные представления, записав лог-данные в файл JSON или CSV.

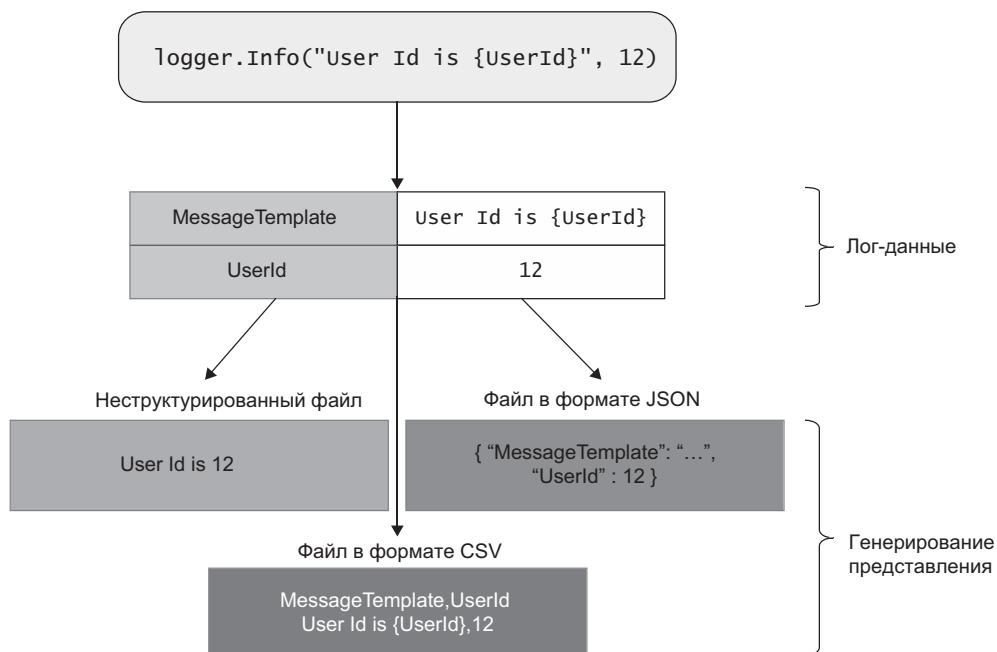


Рис. 8.12. Структурированное логирование отделяет лог-данные от представления этих данных. Вы можете настроить разные форматы представления: неструктурированный файл, JSON или CSV

Написание тестов для служебного и диагностического логирования

Как упоминалось ранее, класс `DomainLogger` представляет внепроцессную зависимость — хранилище логов. И это создает проблему: класс `User` теперь взаимодействует с этой зависимостью, а следовательно, нарушает разделение между бизнес-логикой и взаимодействием с внепроцессными зависимостями. Использование `DomainLogger` переводит `User` в категорию переусложненного кода, что затрудняет его тестирование и сопровождение (за подробностями о категориях кода обращайтесь к главе 7).

Эта проблема может решаться так же, как ранее было реализовано уведомление внешних систем об изменении имейла пользователя: с использованием доменных событий (за подробностями также обращайтесь к главе 7). Вы можете ввести отдельное событие для отслеживания изменений в типе пользователя. Контроллер затем будет преобразовывать эти изменения в вызовы `DomainLogger`, как показано в листинге 8.6.

Обратите внимание, что теперь существует два события предметной области `UserTypeChangedEvent` и `EmailChangedEvent`. Оба события реализуют один интерфейс (`IDomainEvent`), а следовательно, могут храниться в одной коллекции.

Листинг 8.6. Замена DomainLogger в User доменным событием

```

public void ChangeEmail(string newEmail, Company company)
{
    _logger.Info(
        $"Changing email for user {UserId} to {newEmail}");

    Precondition.Requires(CanChangeEmail() == null);

    if (Email == newEmail)
        return;

    UserType newType = company.IsEmailCorporate(newEmail)
        ? UserType.Employee
        : UserType.Customer;

    if (Type != newType)
    {
        int delta = newType == UserType.Employee?1:-1;
        company.ChangeNumberOfEmployees(delta);
        AddDomainEvent(
            new UserTypeChangedEvent(
                UserId, Type, newType)); | Использует доменное
                                         | событие вместо
                                         | DomainLogger
    }

    Email = newEmail;
    Type = newType;
    AddDomainEvent(new EmailChangedEvent(UserId, newEmail));

    _logger.Info($"Email is changed for user {UserId}");
}

```

В листинге 8.7 показано, как выглядит код контроллера.

Листинг 8.7. Новая версия UserController

```

public string ChangeEmail(int userId, string newEmail)
{
    object[] userData = _database.GetUserById(userId);
    User user = UserFactory.Create(userData);

    string error = user.CanChangeEmail();
    if (error != null)
        return error;

    object[] companyData = _database.GetCompany();
    Company company = CompanyFactory.Create(companyData);

    user.ChangeEmail(newEmail, company);

    _database.SaveCompany(company);
    _database.SaveUser(user);
    _eventDispatcher.Dispatch(user.DomainEvents); | Диспетчеризация
                                                 | доменных событий
    return "OK";
}

```

Новый класс `EventDispatcher` преобразует доменные события в обращения к вне-процессным зависимостям:

- `EmailChangedEvent` преобразуется в `_messageBus.SendEmailChangedMessage()`;
- `UserTypeChangedEvent` преобразуется в `_domainLogger.UserTypeHasChanged()`.

Использование `UserTypeChangedEvent` восстановило разделение между двумя обязанностями: логикой предметной области и взаимодействиями с вне-процессными зависимостями. Теперь тестирование служебного логирования не отличается от тестирования другой неуправляемой зависимости — шины сообщений.

- Юнит-тесты должны проверить экземпляр `UserTypeChangedEvent` в тестируемом экземпляре `User`.
- Интеграционный тест должен использовать мок для проверки взаимодействий с `DomainLogger`.

Обратите внимание, что если вам нужно обеспечить логирование в контроллере, но не в доменных классах, в использовании событий предметной области нет необходимости. Как вы, возможно, помните из главы 7, контроллеры координируют взаимодействие между доменной моделью и вне-процессными зависимостями. `DomainLogger` — одна из таких зависимостей, а следовательно, `UserController` может использовать этот класс напрямую.

Также заметьте, что я не изменил то, как класс `User` делает диагностическое логирование. `User` продолжает экземпляр `logger` напрямую в начале и в конце метода `ChangeEmail`. Это так и задумано. Диагностическое логирование предназначено только для разработчиков, вам не нужно применять юнит-тестирование к этой функциональности, а следовательно, не нужно отделять ее от доменной модели.

Тем не менее старайтесь воздерживаться от диагностического логирования в `User` и других доменных классах. Причины этого объясняются в следующем разделе.

8.6.3. Какой объем логирования можно считать достаточным?

Другой важный вопрос относится к оптимальному объему логирования. Сколько логирования следует считать достаточным? Служебное логирование должно присутствовать всегда, потому что оно относится к бизнес-требованиям. Но диагностическое логирование находится под вашим контролем.

Важно не злоупотреблять диагностическим логированием по следующим двум причинам:

- лишнее логирование загромождает код. Это особенно заметно в доменной модели. Именно по этой причине я не рекомендую использовать диагностическое логирование в `User`, несмотря на то что такое использование не противоречит принципам юнит-тестирования;

- сопоношение «сигнал/шум» в логах является ключевым. Чем больше данных вы сохраняете, тем сложнее найти нужную информацию. Максимизируйте сигнал; сведите к минимуму уровень шума.

Постарайтесь вообще не использовать диагностическое логирование в доменной модели. В большинстве случаев такое логирование можно легко переместить из классов предметной области в контроллеры. И даже тогда используйте диагностическое логирование только временно, когда требуется заниматься отладкой. После завершения отладки удалите его. В идеале диагностическое логирование должно использоваться только для необработанных исключений.

8.6.4. Как передавать экземпляры логеров?

И наконец, последний вопрос: как передавать экземпляры логеров в коде? Один из способов основан на использовании статических методов, как показано в листинге 8.8.

Листинг 8.8. Хранение ILogger в статическом поле

```
public class User
{
    private static readonly ILogger _logger = LogManager.GetLogger(typeof(User));
```

||| Получает ILogger при помощи статического метода и сохраняет в приватном статическом поле

```
    public void ChangeEmail(string newEmail, Company company)
    {
        _logger.Info(
            $"Changing email for user {UserId} to {newEmail}");

        /* ... */

        _logger.Info($"Email is changed for user {UserId}");
    }
}
```

Стивен ван Дюрзен (Steven van Deursen) и Марк Симан (Mark Seeman) в своей книге «Dependency Injection Principles, Practices, Patterns» (Manning Publications, 2018) называют такой тип получения зависимостей *неявным контекстом (ambient context)*. Они считают его антипаттерном, потому что:

- зависимость скрыта, и ее трудно изменить;
- тестирование также становится более сложным.

Я поддерживаю их точку зрения. Однако для меня главный недостаток неявного контекста заключается в том, что он маскирует потенциальные проблемы в коде. Если явное внедрение логера в класс предметной области создает такие неудобства, что вам приходится прибегать к неявному контексту, то это становится явным

признаком проблемы. Вы либо логируете слишком много, либо используете слишком много уровней абстракции. В любом случае неявный контекст решением не является. Вместо этого следует разбираться с корневой причиной происходящего.

В листинге 8.9 продемонстрирован один из способов явного внедрения логера в аргумент метода. Также можно воспользоваться конструктором класса.

Листинг 8.9. Явное внедрение диспетчера протоколирования

```
public void ChangeEmail(  
    string newEmail, | Внедрение  
    Company company, | через метод  
    ILogger logger)  
{  
    logger.Info(  
        $"Changing email for user {UserId} to {newEmail}");  
  
    /* ... */  
  
    logger.Info($"Email is changed for user {UserId}");  
}
```

8.7. Заключение

Взаимодействия с внепроцессными зависимостями следует рассматривать с точки зрения того, являются ли эти взаимодействия частью наблюдаемого поведения приложения или же деталью имплементации кода. Хранение логов ничем не отличается в этом отношении. Заменяйте функциональность логирования моком, если логи доступны непрограммистам; в противном случае тестировать ее не нужно. В следующей главе мы глубже погрузимся в тему моков и связанных с ними рекомендаций.

Итоги

- Интеграционным тестом является любой тест, который не является юнит-тестом. Интеграционные тесты проверяют, как ваша система работает в интеграции с внепроцессными зависимостями.
- Интеграционные тесты покрывают контроллеры; юнит-тесты покрывают алгоритмы и доменную модель.
- Интеграционные тесты обеспечивают лучшую защиту от багов и устойчивость к рефакторингу; юнит-тесты более просты в поддержке и дают более быструю обратную связь.
- «Порог» для написания интеграционных тестов выше, чем для юнит-тестов: их эффективность по метрике защиты от багов и устойчивости к рефакторингу

должна быть выше, чем у юнит-тестов, для того чтобы скомпенсировать дополнительную сложность в поддержке и медленную обратную связь. Пирамида тестирования отражает этот компромисс: большинство тестов должны составлять быстрые и простые в поддержке юнит-тесты при меньшем количестве медленных и более сложных в поддержке интеграционных тестов, проверяющих правильность системы в целом.

- Проверяйте как можно больше пограничных случаев бизнес-сценария юнит-тестами. Используйте интеграционные тесты для покрытия одного позитивного пути, а также всех пограничных случаев, которые не могут быть покрыты юнит-тестами.
- Форма пирамиды тестирования зависит от сложности проекта. Простые проекты содержат небольшой объем кода в доменной модели, а следовательно, могут иметь одинаковое количество юнит- и интеграционных тестов. В наиболее тривиальных случаях юнит-тестов может не быть вообще.

- Принцип Fail Fast призывает к быстрому обнаружению ошибок в программе; он является допустимой альтернативой интеграционному тестированию.
- Управляемые зависимости представляют собой внепроцессные зависимости, доступ к которым осуществляется только через ваше приложение. Взаимодействия с управляемыми зависимостями не видны извне. Типичный пример — база данных приложения.
- Неуправляемые зависимости — внепроцессные зависимости, доступные для других приложений. Взаимодействия с неуправляемыми зависимостями видны снаружи. Типичные примеры — сервер SMTP и шина сообщений.
- Взаимодействия с управляемыми зависимостями являются деталями имплементации; взаимодействия с неуправляемыми зависимостями являются частью наблюдаемого поведения вашей системы.
- Используйте реальные экземпляры управляемых зависимостей в интеграционных тестах; заменяйте неуправляемые зависимости моками.
- Иногда внепроцессная зависимость обладает свойствами как управляемых, так и неуправляемых зависимостей. Типичный пример — база данных, доступная для других приложений. Наблюданную часть такой базы следует интерпретировать как неуправляемую зависимость; заменяйте ее моками в тестах. Рассматривайте остальную часть зависимости как управляемую — проверяйте ее итоговое состояние, а не взаимодействия с ней.
- Интеграционный тест должен пройти через все уровни, работающие с управляемой зависимостью. В примере с базой данных это означает проверку состояния базы данных независимо от данных, используемых во входных параметрах.
- Интерфейсы с одной реализацией не являются абстракциями и способствуют слабой связности не более чем конкретные классы, реализующие эти интерфей-

сы. Попытки предвидеть будущие реализации таких интерфейсов нарушают принцип YAGNI.

- Единственная причина для использования интерфейсов с единственной реализацией — возможность использования моков. Используйте такие интерфейсы только для неуправляемых зависимостей. Используйте конкретные классы для управляемых зависимостей.
- Использование интерфейсов с одной реализацией для внутрипроцессных зависимостей — признак проблем с кодом. Такие интерфейсы обычно используют для мокирования и проверки взаимодействий между классами предметной области, что приводит к привязке тестов к деталям имплементации тестируемого кода.
- Выделите явное место для модели предметной области в коде. Четкая граница между классами предметной области и контроллерами помогает отличать юнит-тесты от интеграционных.
- Лишние уровни абстракции отрицательно влияют на вашу способность понимать код. Постарайтесь свести количество этих уровней к минимуму. В большинстве бэкенд-систем достаточно всего трех слоев: предметной области, сервисов приложения и инфраструктуры.
- Циклические зависимости увеличивают когнитивную нагрузку при попытках разобраться в коде. Типичный пример — обратный вызов (когда вызываемая сторона уведомляет вызывающую о результате своей работы). Разорвите цикл введением объекта-значения; используйте этот объект-значение для возвращения результата от вызываемой стороны к вызывающей.
- Множественные секции действий в тестах оправданы только в том случае, если тест работает с внепроцессными зависимостями, которые трудно привести в нужное состояние. Никогда не включайте несколько действий в юнит-тест, потому что юнит-тесты не работают с внепроцессными зависимостями. Многофазные тесты почти всегда принадлежат к категории сквозных.
- Служебное логирование предназначено для персонала службы поддержки и системных администраторов; оно является частью наблюдаемого поведения приложения. Диагностическое логирование помогает разработчику понять, что происходит внутри приложения; оно относится к деталям имплементации.
- Так как служебное логирование является бизнес-требованием, это требование следует явно отразить в коде. Добавьте специальный класс `DomainLogger`, в котором перечисляются все требования к служебному логированию с стороны бизнеса.
- Относитесь к служебному логированию как к любой другой функциональности, работающей с внепроцессными зависимостями. Используйте доменные события для отслеживания изменений в доменной модели, преобразовывайте эти события в обращения к `DomainLogger` в контроллерах.

- Не тестируйте диагностическое логирование. В отличие от служебного, диагностическое логирование может выполняться непосредственно в модели предметной области.
- Не злоупотребляйте диагностическим логированием. Избыточное диагностическое логирование загромождает код и снижает отношение «сигнал/шум» в логах. В идеале диагностическое логирование должно применяться только для необработанных исключений.
- Всегда явно внедряйте все зависимости (включая логеры) — либо через конструктор, либо через аргументы метода.

Рекомендации при работе с моками

В этой главе:

- ✓ Максимизация эффективности моков.
- ✓ Замена моков шпионами.
- ✓ Практики мокирования.

Как говорилось в главе 5, мок (mock) представляет собой тестовую заглушку, которая помогает эмулировать и проверять взаимодействия между тестируемой системой и ее зависимостями. В главе 8 я также упоминал о том, что моки должны применяться только к неуправляемым зависимостям (взаимодействия с такими зависимостями видимы внешним приложением). Применение моков для любых других целей приводит к хрупким тестам (тестам с плохим значением метрики устойчивости к рефакторингу). Соблюдение этого правила — это 70 % того, что нужно знать о моках.

В этой главе приведены остальные рекомендации, которые помогут вам сделать интеграционные тесты максимально эффективными за счет достижения максимальной устойчивости к рефакторингу и защиты от багов. Сначала я продемонстрирую типичное применение моков, опишу его недостатки, а затем покажу альтернативный вариант.

9.1. Достижение максимальной эффективности моков

Важно ограничить использование моков неуправляемыми зависимостями, но это только первый шаг на пути к максимизации ценности моков. Эту тему лучше всего

пояснить на примере, поэтому я продолжу использовать систему CRM, описанную ранее. Я напомню ее функциональность и приведу интеграционный тест, который у нас получился в итоге. После этого будет показано, как улучшить этот тест в отношении мокирования.

Как говорилось ранее, система CRM в настоящее время поддерживает только один бизнес-сценарий: изменение имейла пользователя. В листинге 9.1 приведен код контроллера, который у нас получился.

Листинг 9.1. Класс UserController

```
public class UserController
{
    private readonly Database _database;
    private readonly EventDispatcher _eventDispatcher;

    public UserController(
        Database database,
        IMessageBus messageBus,
        IDomainLogger domainLogger)
    {
        _database = database;
        _eventDispatcher = new EventDispatcher(
            messageBus, domainLogger);
    }

    public string ChangeEmail(int userId, string newEmail)
    {
        object[] userData = _database.GetUserById(userId);
        User user = UserFactory.Create(userData);

        string error = user.CanChangeEmail();
        if (error != null)
            return error;

        object[] companyData = _database.GetCompany();
        Company company = CompanyFactory.Create(companyData);

        user.ChangeEmail(newEmail, company);

        _database.SaveCompany(company);
        _database.SaveUser(user);
        _eventDispatcher.Dispatch(user.DomainEvents);

        return "OK";
    }
}
```

Обратите внимание, что диагностического логирования больше нет, но служебное логирование (интерфейс `IDomainLogger`) осталось (за подробностями обращайтесь к главе 8). Также в листинге 9.1 появился новый класс: `EventDispatcher`. Он

преобразует доменные события, генерируемые моделью предметной области, в обращения к неуправляемым зависимостям (то, что контроллер ранее делал сам).

Листинг 9.2. Диспетчер событий

```
public class EventDispatcher
{
    private readonly IMessageBus _messageBus;
    private readonly IDomainLogger _domainLogger;

    public EventDispatcher(
        IMessageBus messageBus,
        IDomainLogger domainLogger)
    {
        _domainLogger = domainLogger;
        _messageBus = messageBus;
    }

    public void Dispatch(List<IDomainEvent> events)
    {
        foreach (IDomainEvent ev in events)
        {
            Dispatch(ev);
        }
    }

    private void Dispatch(IDomainEvent ev)
    {
        switch (ev)
        {
            case EmailChangedEvent emailChangedEvent:
                _messageBus.SendEmailChangedMessage(
                    emailChangedEvent.UserId,
                    emailChangedEvent.NewEmail);
                break;

            case UserTypeChangedEvent userTypeChangedEvent:
                _domainLogger.UserTypeHasChanged(
                    userTypeChangedEvent.UserId,
                    userTypeChangedEvent.OldType,
                    userTypeChangedEvent.NewType);
                break;
        }
    }
}
```

Наконец, в листинге 9.3 приведен интеграционный тест. Этот тест проходит через все внепроцессные зависимости (как управляемые, так и неуправляемые).

Этот тест заменяет моками две неуправляемые зависимости: `IMessageBus` и `IDomainLogger`. Сначала рассмотрим `IMessageBus`. `IDomainLogger` будет рассматриваться позднее в этой главе.

Листинг 9.3. Интеграционный тест

```
[Fact]
public void Changing_email_from_corporate_to_non_corporate()
{
    // Arrange
    var db = new DatabaseConnectionString();
    User user = CreateUser("user@mycorp.com", UserType.Employee, db);
    CreateCompany("mycorp.com", 1, db);

    var messageBusMock = new Mock<IMessageBus>();           | Подготавливает
    var loggerMock = new Mock<IDomainLogger>();                | моки
    var sut = new UserController(
        db, messageBusMock.Object, loggerMock.Object);

    // Act
    string result = sut.ChangeEmail(user.UserId, "new@gmail.com");

    // Assert
    Assert.Equal("OK", result);

    object[] userData = db.GetUserById(user.UserId);
    User userFromDb = UserFactory.Create(userData);
    Assert.Equal("new@gmail.com", userFromDb.Email);
    Assert.Equal(UserType.Customer, userFromDb.Type);

    object[] companyData = db.GetCompany();
    Company companyFromDb = CompanyFactory.Create(companyData);
    Assert.Equal(0, companyFromDb.NumberOfEmployees);

    messageBusMock.Verify(
        x => x.SendEmailChangedMessage(
            user.UserId, "new@gmail.com"),
        Times.Once);
    loggerMock.Verify(
        x => x.UserTypeHasChanged(
            user.UserId,
            UserType.Employee,
            UserType.Customer),
        Times.Once);
}
```

Проверяет
взаимодействия
с моками

9.1.1. Проверка взаимодействий на границах системы

Давайте посмотрим, почему моки, используемые в интеграционном тесте из листинга 9.3, не идеальны в плане защиты от багов и устойчивости к рефакторингу, и как решить эту проблему.

СОВЕТ

При использовании моков всегда руководствуйтесь следующим принципом: проверяйте взаимодействия с неуправляемыми зависимостями на самых границах вашей системы.

Проблема с `messageBusMock` в листинге 9.3 заключается в том, что интерфейс `IMessageBus` не находится на границе системы. Посмотрите на реализацию этого интерфейса.

Листинг 9.4. Шина сообщений

```
public interface IMessageBus
{
    void SendEmailChangedMessage(int userId, string newEmail);
}

public class MessageBus : IMessageBus
{
    private readonly IBus _bus;

    public void SendEmailChangedMessage(
        int userId, string newEmail)
    {
        _bus.Send("Type: USER EMAIL CHANGED; " +
            $"Id: {userId}; " +
            $"NewEmail: {newEmail}");
    }
}

public interface IBus
{
    void Send(string message);
}
```

Интерфейсы `IMessageBus` и `IBus` (а также классы, реализующие их) принадлежат коду нашего проекта. `IBus` — обертка, работающая поверх библиотеки SDK шины сообщений (предоставленной компанией, которая разработала шину сообщений). Эта обертка инкапсулирует несущественные технические детали (например, настройка подключения) и предоставляет удобный лаконичный интерфейс для отправки произвольных текстовых сообщений по шине. `IMessageBus` — обертка над `IBus`; он определяет сообщения, специфические для вашей предметной области. `IMessageBus` помогает держать все такие сообщения в одном месте и переиспользовать их в приложении.

Интерфейсы `IBus` и `IMessageBus` можно объединить, но такое решение будет неоптимальным. Эти две обязанности — скрытие сложности внешней библиотеки и хранение всех сообщений приложения в одном месте — лучше держать раздельно. Аналогичная ситуация существовала с `ILogger` и `IDomainLogger` из главы 8. `IDomainLogger` реализует специфическую функциональность, требуемую бизнесом, для чего во внутренней реализации он использует обобщенный интерфейс `ILogger`.

На рис. 9.1 показано положение `IBus` и `IMessageBus` с точки зрения гексагональной архитектуры: `IBus` является последним звеном в цепочке типов между контроллером

и шиной сообщений, тогда как `IMessageBus` — всего лишь промежуточный тип в этой цепочке.

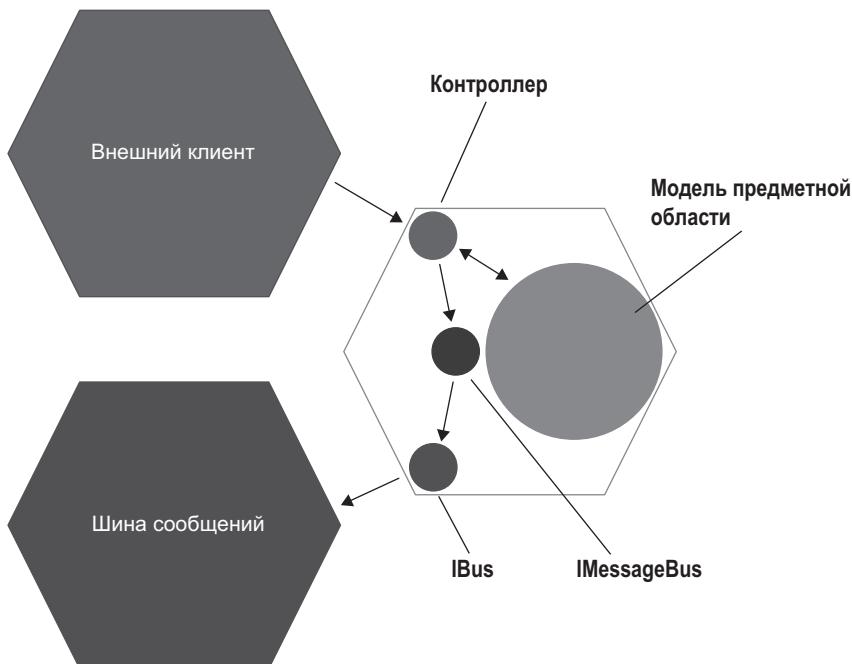


Рис. 9.1. `IBus` находится на границе системы; `IMessageBus` — всего лишь промежуточное звено в цепочке типов между контроллером и шиной сообщений. Мокирование `IBus` вместо `IMessageBus` обеспечивает лучшую защиту от багов

Мокирование `IBus` вместо `IMessageBus` обеспечивает максимальный уровень защиты от регрессий, предоставляемой моком. Как вы, возможно, помните из главы 4, защита от регрессий зависит от объема кода, выполняемого в ходе теста. Мокирование последнего типа, взаимодействующего с неуправляемой зависимостью, повышает количество классов, отрабатываемых интеграционным тестом, а следовательно, улучшает защиту. Эта рекомендация также является причиной, по которой не следует мокировать класс `EventDispatcher`. Он находится еще дальше от границы системы по сравнению с `IMessageBus`.

В листинге 9.5 приведен код интеграционного теста после перенаправления его с `IMessageBus` на `IBus`. Части, не изменившиеся по сравнению с листингом 9.3, опущены.

Обратите внимание: в тесте теперь используется конкретный класс `MessageBus` вместо соответствующего интерфейса `IMessageBus`. `IMessageBus` — интерфейс с одной

реализацией, и, как вы помните из главы 8, мокирование — единственная причина для добавления таких интерфейсов. Так как мы уже не мокируем `IMessageBus`, этот интерфейс можно удалить, заменив его использования на `MessageBus`.

Листинг 9.5. Интеграционный тест для IBus

```
[Fact]
public void Changing_email_from_corporate_to_non_corporate()
{
    var busMock = new Mock<IBus>();
    var messageBus = new MessageBus(busMock.Object);           ← Использует
                                                               конкретный класс
                                                               вместо интерфейса

    var loggerMock = new Mock<IDomainLogger>();
    var sut = new UserController(db, messageBus, loggerMock.Object);

    /* ... */

    busMock.Verify(
        x => x.Send(
            $"Type: USER EMAIL CHANGED; " +
            $"Id: {user.UserId}; " +
            $"NewEmail: new@gmail.com"),
        Times.Once);                                              | Проверяет сообщение,
                                                               отправленное по шине
}

```

Также обратите внимание на то, как тест в листинге 9.5 проверяет текстовое сообщение, отправленное по шине. Сравните его с предыдущей версией:

```
messageBusMock.Verify(
    x => x.SendEmailChangedMessage(user.UserId, "new@gmail.com"),
    Times.Once);
```

Существует большое различие между проверкой обращения к классу, который вы написали сами, и проверкой реального текста, передаваемого внешней системе. Внешние системы ожидают получать от вашего приложения текстовые сообщения, а не обращения к классам вроде `MessageBus`. Текстовые сообщения — единственный побочный эффект, наблюдаемый извне; классы, участвующие в производстве таких сообщений, являются всего лишь деталями имплементации. Таким образом, в дополнение к повышенной защите от багов проверка взаимодействий на самых границах системы также улучшает устойчивость к рефакторингу. Полученные тесты в меньшей степени подвержены потенциальным ложным срабатываниям: такие тесты упадут, только если изменится структура сообщения.

СОВЕТ

Обращение к неуправляемой зависимости проходит через несколько стадий, прежде чем покинуть ваше приложение. Выберите последнюю из этих стадий. Это лучший способ обеспечить обратную совместимость с внешними системами, что является целью использования моков.

Здесь действует тот же механизм, который наделяет интеграционные и сквозные тесты дополнительной устойчивостью к рефакторингу по сравнению с юнит-тестами. Они в большей степени отделены от кода, а следовательно, меньше страдают от низкоуровневых рефакторингов.

9.1.2. Замена моков шпионами

Как говорилось в главе 5, *шпион* (*spy*) представляет собой разновидность тестовой заглушки, которая служит той же цели, что и мок. Единственное различие заключается в том, что шпионы пишутся вручную, а моки создаются с помощью мок-фреймворков.

Для классов, находящихся возле границ системы, шпионы предпочтительнее моков. Шпионы помогают переиспользовать код в фазе проверок, сокращая тем самым размер и улучшая читаемость тестов. В листинге 9.6 приведен пример шпиона, работающего поверх *IBus*.

Листинг 9.6. Шпион (вручную написанный мок)

```
public interface IBus
{
    void Send(string message);
}

public class BusSpy : IBus
{
    private List<string> _sentMessages = new List<string>();
    public void Send(string message)
    {
        _sentMessages.Add(message);
    }

    public BusSpy ShouldSendNumberOfMessages(int number)
    {
        Assert.Equal(number, _sentMessages.Count);
        return this;
    }

    public BusSpy WithEmailChangedMessage(int userId, string newEmail)
    {
        string message = "Type: USER EMAIL CHANGED; " +
            $"Id: {userId}; " +
            $"NewEmail: {newEmail}";
        Assert.Contains(
            _sentMessages, x => x == message);
        return this;
    }
}
```

Сохраняет все отправленные сообщения локально

Проверяет, что сообщение было отправлено

В листинге 9.7 приведена новая версия интеграционного теста. Как и в предыдущем случае, я привожу только изменившиеся части.

Проверка взаимодействий с шиной сообщений стала лаконичной и выразительной благодаря fluent (текущему) интерфейсу, предоставляемому BusSpy. С таким fluent-интерфейсом можно объединить несколько проверок в цепочку, образуя связные выражения, напоминающие естественный язык.

Листинг 9.7. Использование шпиона из листинга 9.6

```
[Fact]
public void Changing_email_from_corporate_to_non_corporate()
{
    var busSpy = new BusSpy();
    var messageBus = new MessageBus(busSpy);
    var loggerMock = new Mock<IDomainLogger>();
    var sut = new UserController(db, messageBus, loggerMock.Object);

    /* ... */

    busSpy.ShouldSendNumberOfMessages(1)
        .WithEmailChangedMessage(user.UserId, "new@gmail.com");
}
```

СОВЕТ

BusSpy можно переименовать в BusMock. Как упоминалось ранее, различие между моком и шпионом является незначительным. Тем не менее многим программистам не знаком термин «шпион», так что переименование шпиона в BusMock может избавить ваших коллег от лишней путаницы.

И здесь возникает резонный вопрос: разве мы не сделали полный круг, вернувшись к отправной точке? Версия теста в листинге 9.7 очень похожа на более раннюю версию с моком для `IMessageBus`:

```
messageBusMock.Verify(
    x => x.SendEmailChangedMessage(
        user.UserId, "new@gmail.com"),
    Times.Once);
```

Аналог `WithEmailChangedMessage(user.UserId, "new@gmail.com")`
|
Аналог `ShouldSendNumberOfMessages(1)`

Эти проверки похожи, потому что и BusSpy, и `MessageBus` являются обертками для `IBus`. Однако между ними существует принципиальное отличие: BusSpy является частью тестового кода, тогда как `MessageBus` относится к рабочему коду. Это важное отличие, потому что при использовании проверок в тестах вы не должны полагаться на рабочий код.

Ваши тесты — аудиторы кода. Хороший аудитор не принимает слова аудируемого на веру просто так; он все проверяет и перепроверяет. То же относится и к шпиону:

он предоставляет независимую точку контроля, которая поднимает тревогу при изменении структуры сообщения. С другой стороны, мок для `IMessageBus` слишком «доверяет» рабочему коду.

9.1.3. Как насчет `IDomainLogger`?

Мок, который ранее проверял взаимодействия с `IMessageBus`, теперь ориентирован на интерфейс `IBus`, находящийся на границе системы. В листинге 9.8 приведены текущие проверки в интеграционном teste.

Листинг 9.8. Проверки моков

```
busSpy.ShouldSendNumberOfMessages(1)
    .WithEmailChangedMessage(
        user.UserId, "new@gmail.com");
```

Проверяет
взаимодействия
с `IBus`


```
loggerMock.Verify(
    x => x.UserTypeHasChanged(
        user.UserId,
        UserType.Employee,
        UserType.Customer),
    Times.Once);
```

Проверяет
взаимодействия
с `IDomainLogger`

Подобно тому как `MessageBus` является оберткой для `IBus`, `DomainLogger` является оберткой для `ILogger` (за подробностями обращайтесь к главе 8). Не следует ли переориентировать тест на `ILogger`, потому что этот интерфейс также находится на границе приложения?

В большинстве проектов такая переориентация не обязательна. Хотя логер и шина сообщений являются неуправляемыми зависимостями, а следовательно, требуют сохранения обратной совместимости, точность такой совместимости не обязана быть одинаковой для обоих зависимостей. С шиной сообщений важно не допустить никаких изменений в структуре сообщений, потому что вы никогда не знаете, как внешние системы отреагируют на такие изменения. Но точная структура текстовых логов не настолько важна для предполагаемой аудитории (персонала службы поддержки и системных администраторов). Важно само существование этих логов и хранящейся в них информации. Таким образом, мокирование `IDomainLogger` обеспечивает достаточный уровень защиты.

9.2. Практики мокирования

До настоящего момента были описаны две основные практики мокирования:

- применение моков только к неуправляемым зависимостям;
- проверка взаимодействий с этими зависимостями на границах системы.

В этом разделе рассматриваются остальные практики:

- использование моков только в интеграционных (но не юнит-) тестах;
- проверка количества обращений к моку;
- мокирование только тех типов, которые вам принадлежат.

9.2.1. Моки только для интеграционных тестов

Правило, гласящее, что моки должны использоваться только в интеграционных, но не в юнит-тестах, следует из фундаментального принципа, описанного в главе 7: разделения бизнес-логики и координации. Ваш код должен либо взаимодействовать с внепроцессными зависимостями, либо быть сложным — но не делать и то и другое одновременно. Этот принцип приводит к формированию двух слоев: слоя модели предметной области (который отвечает за сложность) и слоя контроллеров (который отвечает за взаимодействия).

Тесты доменной модели относятся к категории юнит-тестов; тесты, покрывающие контроллер, являются интеграционными. Так как моки предназначены только для неуправляемых зависимостей, а контроллеры — единственный код, работающий с такими зависимостями, моки должны применяться только при тестировании контроллеров — в интеграционных тестах.

9.2.2. Несколько моков на тест

Иногда можно слышать правило, что на каждый тест должно приходиться не более одного мока. Согласно этой рекомендации, если моков несколько, то вы с большой вероятностью тестируете сразу несколько вещей.

Это ошибочное утверждение происходит от более фундаментальной ошибки, упомянутой в главе 2: что термин «юнит» в «юнит-тесте» относится к единице кода, а все такие единицы должны тестироваться в изоляции друг от друга. На самом деле под «юнитом» понимается *единица поведения*, а не единица кода. Объем кода, необходимый для реализации такой единицы поведения, не имеет значения. Он может охватывать несколько классов, один класс или ограничиваться одним методом.

С моками действует тот же принцип: неважно, сколько моков потребуется для проверки единицы поведения. Ранее в этой главе для проверки сценария изменения имела с корпоративного на обычный нам потребовалось два мока: для логера и для шины сообщений. Впрочем, их могло бы быть и больше — количество моков, используемых в интеграционных тестах, вам неподконтрольно. Оно зависит исключительно от количества неуправляемых зависимостей, участвующих в операции.

9.2.3. Проверка количества вызовов

Что касается взаимодействий с неуправляемыми зависимостями, важно проверить оба следующих аспекта:

- наличие ожидаемых обращений к моку;
- отсутствие неожиданных обращений к моку.

Это требование также происходит от необходимости обеспечения обратной совместимости с неуправляемыми зависимостями. Совместимость должна распространяться в обоих направлениях: ваше приложение должно продолжать отправлять сообщения, ожидаемые внешней системой, но также и не должно отправлять неожиданные сообщения. Недостаточно проверить, что тестируемая система отправляет сообщение:

```
messageBusMock.Verify(  
    x => x.SendEmailChangedMessage(user.UserId, "new@gmail.com"));
```

Также необходимо убедиться в том, что сообщение отправлено ровно один раз:

```
messageBusMock.Verify(  
    x => x.SendEmailChangedMessage(user.UserId, "new@gmail.com"),  
    Times.Once); ← Проверяет, что метод вызывается только один раз
```

В большинстве мок-библиотек можно также явно проверить, что никакие другие вызовы к моку не совершаются. В Moq подобная проверка выглядит так:

```
messageBusMock.Verify(  
    x => x.SendEmailChangedMessage(user.UserId, "new@gmail.com"),  
    Times.Once);  
messageBusMock.VerifyNoOtherCalls(); ← Дополнительная проверка
```

Данная функциональность реализована и в BusSpy:

```
busSpy  
    .ShouldSendNumberOfMessages(1)  
    .WithEmailChangedMessage(user.UserId, "new@gmail.com");
```

Проверка `ShouldSendNumberOfMessages(1)` включает в себя как `Times.Once`, так и `VerifyNoOtherCalls()`.

9.2.4. Используйте моки только для принадлежащих вам типов

Последняя рекомендация, о которой мне хотелось бы упомянуть, — использование моков только для принадлежащих вам типов. Впервые о ней упомянули Стив Фримен (Steve Freeman) и Нэт Прайс (Nat Pryce)¹. Эта рекомендация гласит, что

¹ См. с. 69 в *Growing Object-Oriented Software, Guided by Tests*, Стив Фримен (Steve Freeman) и Нэт Прайс (Nat Pryce), Addison-Wesley Professional, 2009.

вы всегда должны писать собственные адаптеры на базе сторонних библиотек и мокировать эти адаптеры вместо нижележащих типов. Некоторые из их аргументов:

- часто вы не обладаете глубоким пониманием того, как работает сторонний код;
- даже если код уже предоставляет встроенные интерфейсы, мокирование этих интерфейсов часто сопряжено с риском, потому что вы должны быть уверены в том, что поведение мока совпадает с тем, что фактически делает внешняя библиотека;
- адаптеры абстрагируют несущественные технические детали стороннего кода и определяют отношения с библиотекой, используя доменный язык вашего приложения.

Я полностью согласен с этими утверждениями. По сути, адаптеры работают как антикоррупционный слой (*anti-corruption layer*) между вашим кодом и внешним миром¹. Они помогают:

- абстрагировать сложность используемой библиотеки;
- делать публичной только нужную вам функциональность из этой библиотеки;
- делать это с использованием языка предметной области вашего проекта.

Интерфейс `IBus` в проекте CRM служит именно этой цели. Даже если библиотека используемой шины сообщений предоставляет такой же лаконичный интерфейс, как `IBus`, все равно лучше создать собственную обертку поверх него. Вы никогда не знаете, как изменится сторонний код при обновлении библиотеки. Такое обновление может привести к множеству ошибок компиляции по всему коду проекта. Дополнительный уровень абстракции ограничивает эти ошибки только одним классом: самим адаптером.

Обратите внимание, что рекомендация «используйте моки только для своих типов» не относится к внутрипроцессным зависимостям. Как объяснялось ранее, моки предназначены только для неуправляемых зависимостей. А следовательно, в абстрагировании находящихся в памяти или управляемых зависимостей нет необходимости. Например, если библиотека предоставляет API для работы с датой и временем, вы можете использовать этот API в неизменном виде, потому что он не обращается к неуправляемым зависимостям. Аналогичным образом нет необходимости абстрагировать прослойку ORM, при условии что она используется для обращения к базе данных, невидимой для внешних приложений. Конечно, вы можете создать собственную обертку поверх любой библиотеки, но обычно потраченные усилия не оправдываются чем-либо, кроме неуправляемых зависимостей.

¹ См. *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Эрик Эванс (Eric Evans), Addison-Wesley, 2003.

Итоги

- Проверяйте взаимодействия с неуправляемыми зависимостями на самых границах вашей системы. Мокируйте последний тип в цепочке типов между контроллером и неуправляемой зависимостью. Это поможет улучшить как защиту от багов (ввиду того что интеграционные тесты проверяют больший объем кода), так и устойчивость к рефакторингу (так как мок отделяется от подробностей реализации кода).
- Шпионы представляют собой моки, написанные вручную. Для классов, находящихся на границах системы, шпионы предпочтительнее моков. Они способствуют переиспользованию кода проверок, что приводит к сокращению размера теста и улучшению читаемости.
- Не полагайтесь на рабочий код в тестовых проверках. Используйте отдельный набор литералов и констант в тестах. Если потребуется, скопируйте эти литералы и константы из рабочего кода. Тесты должны предоставить контрольную точку, которая не зависит от рабочего кода. В противном случае появляется риск создания *тавтологических* тестов (тестов, которые ничего не проверяют и содержат семантически бессмысленные проверки).
- Не все неуправляемые зависимости требуют одинакового уровня обратной совместимости. Если точная структура сообщения не важна и вы хотите только проверить факт существования сообщения и хранящейся в нем информации, вы можете игнорировать рекомендацию по проверке взаимодействий с неуправляемыми зависимостями на границах вашей системы. Типичный пример такого рода — логирование.
- Так как моки предназначены только для неуправляемых зависимостей, а контроллеры должны быть единственной частью кода, работающей с такими зависимостями, моки должны применяться только при тестировании контроллеров — в интеграционных тестах. Не используйте моки в юнит-тестах.
- Количество моков, используемых в teste, несущественно. Оно зависит исключительно от количества неуправляемых зависимостей, участвующих в операции.
- Проверяйте как существование *ожидаемых*, так и отсутствие *неожиданных* обращений к мокам.
- Используйте моки только с принадлежащими вам типами. Напишите для сторонних библиотек собственные адаптеры, предоставляющие доступ к неуправляемым зависимостям. Мокируйте эти адаптеры вместо нежележащих типов.

10

Тестирование базы данных

В этой главе:

- ✓ Предусловия для тестирования базы данных.
- ✓ Практики тестирования баз данных.
- ✓ Управление транзакциями в тестах.
- ✓ Жизненный цикл тестовых данных.

Последний фрагмент картины интеграционного тестирования — управляемые вне-процессные зависимости. Самым типичным примером управляемой зависимости служит база данных приложения — то есть база данных, недоступная для других приложений.

Запуск тестов на реальной базе данных обеспечивает стопроцентную защиту от регрессий, но организация таких тестов может оказаться довольно хлопотным делом. В этой главе описаны подготовительные действия, которые необходимо выполнить перед тем, как приступить к тестированию: в ней рассмотрено отслеживание изменений схемы базы данных, различия между двумя методами развертывания баз данных (на основе состояния и на основе миграций), а также показано, в каких случаях второй метод предпочтительнее первого.

После изучения основ вы узнаете, как управлять транзакциями в ходе тестирования, как чистить остатки тестовых данных, как сократить размер тестов за счет устранения несущественных частей и выведения на первый план самого важного. В этой главе мы сосредоточимся на реляционных базах данных, но многие принципы применимы к другим типам хранилищ данных — таким как документно-ориентированные базы и даже простые текстовые файлы.

10.1. Предусловия для тестирования базы данных

Как говорилось в главе 8, управляемые зависимости должны включаться в интеграционные тесты в исходном виде. Это усложняет работу с такими зависимостями (по сравнению с неуправляемыми), потому что использование моков исключается. Но даже до того как вы начнете писать тесты, необходимо выполнить некоторые подготовительные действия, чтобы интеграционное тестирование стало возможным. В этом разделе будут представлены такие действия:

- хранение базы данных в системе контроля версий;
- использование отдельных экземпляров базы данных каждым разработчиком;
- развертывание базы данных на основе миграций.

Как почти всегда бывает в тестировании, практики, упрощающие тестирование, также улучшают жизнеспособность вашей базы данных в целом. Эти практики принесут пользу даже в том случае, если вы не пишете интеграционные тесты.

10.1.1. Хранение базы данных в системе контроля версий

Первый шаг на пути к тестированию базы данных — отношение к схеме базы данных как к обычному коду. Как и в случае с обычным кодом, схему базы данных лучше хранить в системе контроля версий (например, в Git).

Я работал над проектами, в которых программисты поддерживали выделенный экземпляр базы данных, который служил эталоном. В ходе разработки все изменения схемы накапливались в этом экземпляре. При развертывании в продуктив команды сравнивали рабочую и эталонную базы данных, генерировала скрипты обновления при помощи специальной программы и запускала эти скрипты на рабочей копии (рис. 10.1).

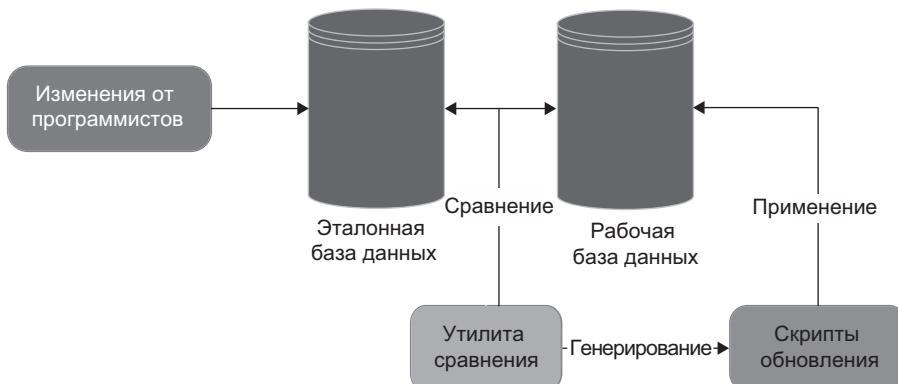


Рис. 10.1. Наличие выделенного экземпляра базы данных, который используется в качестве эталона, — антипаттерн. Схему базы данных необходимо хранить в системе контроля версий

Использование эталонной базы данных — плохой подход к поддержке схемы базы данных. Это объясняется следующими причинами:

- *отсутствие истории изменений* — схему базы данных невозможно отследить до определенной точки в прошлом, что может быть важно для воспроизведения ошибок в условиях реальной эксплуатации;
- *отсутствие единого источника истины* — эталонная база данных становится конкурирующим источником истины о состоянии разработки. Поддержание двух таких источников (Git и эталонная база данных) создает дополнительную нагрузку.

С другой стороны, хранение всех обновлений схемы базы данных в системе контроля версий помогает поддерживать единый источник истины, а также отслеживать изменения в базе данных вместе с изменениями обычного кода. Никакие изменения в структуре базы данных не должны осуществляться за пределами системы контроля версий.

10.1.2. Справочные данные являются частью схемы базы данных

Под схемой базы данных обычно подразумеваются таблицы, представления, индексы, хранимые процедуры и вообще все, что составляет описание того, как создать базу данных. Сама схема представляется в форме SQL-скриптов. Эти скрипты должны обеспечивать возможность создания полнофункциональной и актуальной базы данных в любой момент в ходе разработки. Тем не менее существует и другая часть базы данных, которая также принадлежит схеме, но редко рассматривается как таковая: *справочные данные*.

ОПРЕДЕЛЕНИЕ

Справочными данными называются данные, которые должны быть заранее помещены в базу для правильной работы приложения.

Для примера возьмем систему CRM из предыдущих глав. Пользователи системы могли относиться к одному из двух типов: *Customer* (клиент) или *Employee* (работник). Допустим, вы хотите создать таблицу со всеми типами пользователей и добавить ограничение по внешнему ключу от *User* к этой таблице. Такое ограничение предоставляет дополнительную гарантию того, что приложение никогда не назначит пользователю несуществующий тип. В этом сценарии содержимое таблицы *UserType* становится справочными данными, потому что приложению необходимы эти данные для сохранения информации о пользователях в базе.

Поскольку справочные данные критичны для вашего приложения, они должны храниться в системе контроля версий вместе с таблицами, представлениями и другими компонентами схемы базы данных в форме команд SQL `INSERT`.

СОВЕТ

Существует простой критерий, по которому можно отличить справочные данные от обычных. Если ваше приложение может изменить данные, то это обычные данные; если нет — справочные.

Хотя справочные данные обычно хранятся отдельно от обычных, эти данные в отдельных случаях могут существовать в одной таблице. Чтобы этот подход работал, необходимо добавить флаг, по которому изменяемые (обычные) данные можно было бы отличить от неизменяемых (справочных), и запретить вашему приложению изменять последние.

10.1.3. Отдельный экземпляр для каждого разработчика

Проводить тесты на реальной базе данных достаточно трудно — и становится еще труднее, если база данных используется совместно с другими разработчиками. Работа с совместной базой данных замедляет процесс разработки, потому что:

- тесты, выполняемые разными разработчиками, мешают друг другу;
- изменения, не обладающие обратной совместимостью, могут блокировать работу других разработчиков.

Поддерживайте отдельный экземпляр базы данных для каждого разработчика — желательно на собственной машине этого разработчика для максимизации скорости выполнения тестов.

10.1.4. Развёртывание базы данных на основе состояния и на основе миграций

Существуют два основных метода развертывания баз данных: на основе состояния и на основе миграций. Миграционный метод сложнее реализуется и поначалу создает больше проблем с сопровождением, но в долгосрочной перспективе работает намного лучше, чем метод на основе состояния.

Метод на основе состояния

Метод развертывания баз данных на основе состояния сходен с изображенным на рис. 10.1. У вас также имеется эталонная база данных, которую вы поддерживаете в процессе разработки. При развертывании программа генерирует сценарии для рабочей базы данных, которые приводят ее в соответствие с эталонной. Различие в том, что в методе на основе состояния у вас нет физической эталонной базы данных как источника истины. Вместо этого имеются скрипты SQL, которые используются для создания этой базы данных. Скрипты хранятся в системе контроля версий.

В методе на основе состояния утилита сравнения схемы выполняет всю тяжелую работу. В каком бы состоянии ни находилась рабочая база данных, утилита сделает все необходимое для того, чтобы синхронизировать его с эталонной базой данных: она удалит лишние таблицы, создаст новые таблицы, переименует столбцы и т. д.

Метод на основе миграций

С другой стороны, метод на основе миграций ориентирован на использование миграций, переводящих базу данных от одной версии к другой (рис. 10.2). С этим методом не используются утилиты, автоматически синхронизирующие рабочую базу данных с базой данных разработки; скрипты обновления приходится писать самостоятельно. Впрочем, утилита сравнения баз данных все равно может пригодиться для обнаружения недокументированных изменений в схеме рабочей базы данных.

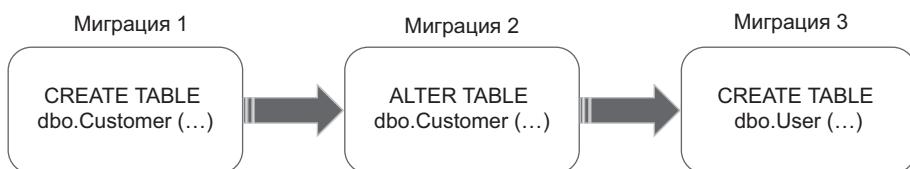


Рис. 10.2. Метод развертывания на основе миграций ориентирован на использование явных миграций, переводящих базу данных из одной версии в другую

При использовании метода на основе миграций именно миграции, а не состояние базы данных становятся артефактами, сохраняемыми в системе контроля версий. Миграции обычно представляются простыми скриптами SQL (популярные программы такого рода — Flyway [<https://flywaydb.org>] и Liquibase [<https://liquibase.org>]), но они также могут быть написаны на DSL-подобном языке, который транслируется в SQL. В следующем примере представлен класс C#, который представляет миграцию базы данных с помощью библиотеки FluentMigrator (<https://github.com/fluentmigrator/fluentmigrator>):

```

[Migration(1)] ← Номер миграции
public class CreateUserTable : Migration
{
    public override void Up() ← Прямая миграция
    {
        Create.Table("Users");
    }

    public override void Down() ← Обратная миграция (полезна при возврате
    {                                к более ранней версии базы данных
        Delete.Table("Users");      для воспроизведения ошибки
    }
}

```

Метод на основе миграций лучше метода на основе состояния

Различия между методом на основе состояния и методом на основе миграций сводятся к следующему (рис. 10.3):

- метод на основе состояния делает состояние базы данных явным (посредством сохранения этого состояния в системе контроля версий) и позволяет утилитам сравнения неявно управлять миграциями;
- метод на основе миграций делает миграции явными, но оставляет состояние неявным. Состояние базы данных невозможно просмотреть напрямую; его приходится собирать из миграций.

	Состояние базы данных	Механизм миграций
Метод на основе состояния	✓ Явно	✗ Неявно
Миграционный метод	✗ Неявно	✓ Явно

Рис. 10.3. Метод на основе состояния явно выражает состояние и неявно — миграции; миграционный метод действует наоборот

Такое различие ведет к разному набору компромиссов. Явное состояние базы данных упрощает обработку конфликтов слияния, тогда как явные миграции помогают при трансформации данных.

ОПРЕДЕЛЕНИЕ

Трансформация данных — процесс изменения формата существующих данных, для того чтобы они соответствовали новой схеме базы данных.

Хотя может показаться, что упрощенная обработка конфликтов и простота трансформации данных — равноценные преимущества, в подавляющем большинстве проектов трансформация данных намного важнее конфликтов слияния. Если только вы еще не запустили свое приложение в продуктив, у вас всегда будут данные, потерять которых попросту недопустима.

Например, если вы разбиваете столбец `Name` на `FirstName` и `LastName`, придется не только удалить столбец `Name` и создать новые столбцы `FirstName` и `LastName`, но и написать скрипт для разбиения всех существующих имён на две части. Реализовать такой рефакторинг методом на основе состояния будет сложно: утилиты сравнения

неэффективны в области управления данными. Дело в том, что хотя сама схема базы данных объективна (то есть может быть интерпретирована только одним способом), данные зависят от контекста. Ни одна программа не может сделать надежные предположения относительно данных при генерировании скриптов обновления. Для реализации правильных трансформаций должны применяться правила, специфические для предметной области.

В результате метод на основе состояния оказывается непрактичным в подавляющем большинстве проектов. Впрочем, он может использоваться временно, пока проект еще не был запущен в эксплуатацию. В конце концов, тестовые данные не настолько важны; их можно воссоздавать заново при каждом изменении схемы базы данных. Но после того как вы выпустите в продуктив первую версию, придется перейти на миграционный метод для нормальной обработки трансформации данных.

СОВЕТ

Применяйте все модификации к схеме базы данных (включая справочные данные) при помощи миграций. Не изменяйте миграции после того, как они будут зафиксированы в системе контроля версий. Если миграция окажется неправильной, создайте новую миграцию, исправляющую ошибку, вместо того чтобы исправлять старую миграцию. Исключения из этого правила допустимы только в том случае, если некорректная миграция может привести к потере данных.

10.2. Управление транзакциями

Тема управления транзакциями баз данных имеет важное значение как в рабочем коде, так и в коде тестов. Эффективное управление транзакциями в рабочем коде помогает избежать нарушения целостности данных. В тестах оно помогает проверить интеграцию с базой данных в условиях, близких к условиям реальной эксплуатации.

В этом разделе я сначала покажу, как обрабатывать транзакции в рабочем коде (контроллере), а затем покажу, как использовать их в интеграционных тестах. При этом в качестве примера будет использоваться проект CRM, уже знакомый вам по предыдущим главам.

10.2.1. Управление транзакциями в рабочем коде

В нашем проекте CRM класс `Database` используется для работы с `User` и `Company`. Класс `Database` создает отдельное SQL-подключение при каждом вызове его методов. Каждое такое подключение неявно открывает отдельную транзакцию, как видно из листинга 10.1.

В результате контроллер создает четыре транзакции в ходе одной бизнес-операции, как видно из листинга 10.2.

Листинг 10.1. Класс, обеспечивающий доступ к базе данных

```

public class Database
{
    private readonly string _connectionString;

    public Database(string connectionString)
    {
        _connectionString = connectionString;
    }

    public void SaveUser(User user)
    {
        bool isNewUser = user.UserId == 0;

        using (var connection =
            new SqlConnection(_connectionString))
        {
            /* Вставка или обновление данных в зависимости от isNewUser */
        }
    }

    public void SaveCompany(Company company)
    {
        using (var connection =
            new SqlConnection(_connectionString))
        {
            /* Только обновление; компания только одна */
        }
    }
}

```


Листинг 10.2. Код контроллера

```

public string ChangeEmail(int userId, string newEmail)
{
    object[] userData = _database.GetUserById(userId); ← Открывает новую транзакцию
    User user = UserFactory.Create(userData);

    string error = user.CanChangeEmail();
    if (error != null)
        return error;

    object[] companyData = _database.GetCompany();
    Company company = CompanyFactory.Create(companyData); ← Открывает новую транзакцию

    user.ChangeEmail(newEmail, company);

    _database.SaveCompany(company);
    _database.SaveUser(user);
    _eventDispatcher.Dispatch(user.DomainEvents);

    return "OK";
}

```

Открывать несколько транзакций в операциях, использующих только чтение, нормально — например, при возвращении информации о пользователе внешнему клиенту. Но если бизнес-операция подразумевает изменение данных, все обновления, выполняемые в ходе этой операции, должны быть атомарными для предотвращения нарушения целостности данных. Например, контроллер может успешно сохранить данные компании, а затем столкнуться с ошибкой при сохранении пользователя из-за проблем с подключением к базе данных. В результате значение `NumberOfEmployees` для компании может отличаться от общего количества пользователей типа `Employee` в базе данных.

ОПРЕДЕЛЕНИЕ

Атомарные обновления выполняются по принципу «все или ничего». Каждое обновление в группе атомарных обновлений должно либо быть завершено полностью, либо вообще ничего не делать.

Отделение подключений к базе данных от транзакций

Чтобы избежать потенциальных несоответствий, необходимо поддерживать разделение между двумя типами решений:

- какие данные обновлять;
- сохранить ли обновления или отменить их.

Это разделение важно, потому что контроллер не может принимать такие решения одновременно. Он будет знать, возможно ли сохранить обновления, только после успешного выполнения всех шагов бизнес-операции. А выполнить эти шаги можно только одним способом — обратившись к базе данных и попытавшись внести обновления. Вы можете разделить эти обязанности, разбив класс `Database` на репозитории и транзакцию:

- *репозитории* — классы, обеспечивающие обращение и модификацию данных в базе данных. В нашем примере будут использоваться два репозитория: для `User` и для `Company`;
- *транзакция* — класс, который либо полностью закрепляет, либо полностью отменяет обновления данных. Это специализированный класс, использующий механизмы базы данных для обеспечения атомарности изменений данных.

Репозитории и транзакции имеют не только разные обязанности, но и разную продолжительность жизни. Транзакция существует на протяжении всей бизнес-операции и уничтожается в самом ее конце. Репозитории существуют недолго. Репозиторий можно уничтожить сразу же после обращения к базе данных. В результате репозитории всегда работают поверх текущей транзакции. При подключении к базе данных

репозиторий регистрируется в транзакции, чтобы любые изменения данных, внесенные при этом подключении, могли быть позднее отменены транзакцией.

На рис. 10.4 показано, как выглядят взаимодействия между контроллером и базой данных из листинга 10.2. Каждое обращение к базе данных упаковывается в отдельную транзакцию; обновления не атомарны.

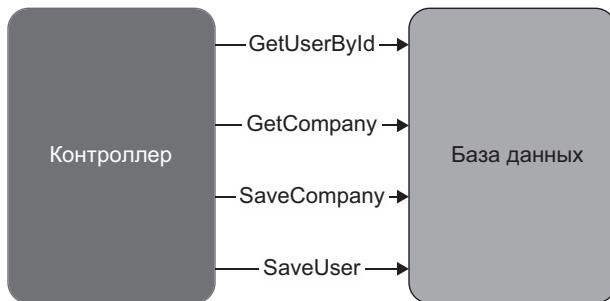


Рис. 10.4. Упаковка каждого обращения к базе данных в отдельную транзакцию создает риск нарушения целостности данных из-за аппаратных или программных сбоев.

Например, приложение может обновить количество работников в компании, но не данные самих работников

На рис. 10.5 изображено приложение после добавления явных транзакций. Транзакция становится связующим звеном между контроллером и базой данных. Все четыре обращения к базе данных остались на своих местах, но теперь модификации данных либо полностью закрепляются, либо полностью отменяются.

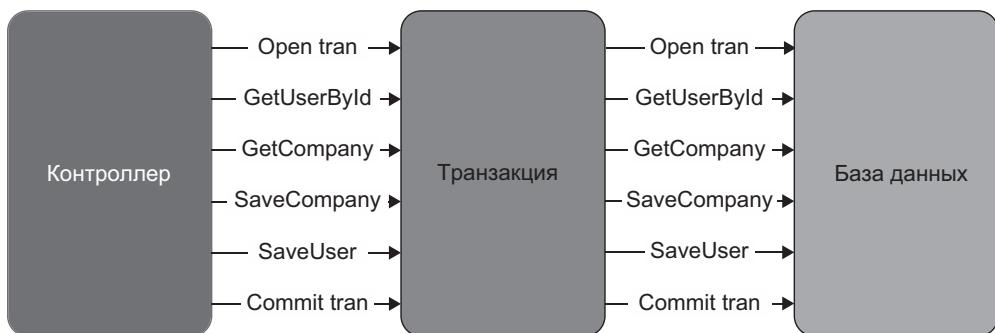


Рис. 10.5. Транзакция становится связующим звеном между контроллером и базой данных, благодаря чему становятся возможными атомарные модификации данных

В листинге 10.3 показан контроллер после добавления транзакции и репозиториев.

Листинг 10.3. Контроллер, репозитории и транзакции

```

public class UserController
{
    private readonly Transaction _transaction;
    private readonly UserRepository _userRepository;
    private readonly CompanyRepository _companyRepository;
    private readonly EventDispatcher _eventDispatcher;

    public UserController( ← Получает транзакцию
        Transaction transaction,
        MessageBus messageBus,
        IDomainLogger domainLogger)
    {
        _transaction = transaction;
        _userRepository = new UserRepository(transaction);
        _companyRepository = new CompanyRepository(transaction);
        _eventDispatcher = new EventDispatcher(
            messageBus, domainLogger);
    }

    public string ChangeEmail(int userId, string newEmail)
    {
        object[] userData = _userRepository
            .GetUserById(userId);
        User user = UserFactory.Create(userData);

        string error = user.CanChangeEmail();
        if (error != null)
            return error;

        object[] companyData = _companyRepository
            .GetCompany();
        Company company = CompanyFactory.Create(companyData);

        user.ChangeEmail(newEmail, company);

        _companyRepository.SaveCompany(company);
        _userRepository.SaveUser(user);
        _eventDispatcher.Dispatch(user.DomainEvents);

        _transaction.Commit(); ← Закрепляет транзакцию в случае успеха
        return "OK";
    }
}

public class UserRepository
{
    private readonly Transaction _transaction;

    public UserRepository(Transaction transaction) ← Внедряет транзакцию в репозиторий
    {

```

```
        _transaction = transaction;
    }

    /* ... */

}

public class Transaction : IDisposable
{
    public void Commit() { /* ... */ }
    public void Dispose() { /* ... */ }
}
```

Внутреннее строение класса `Transaction` не столь важно, но если вам это интересно, он использует стандартный класс .NET `TransactionScope`. Самое важное в классе `Transaction` заключается в том, что он содержит два метода:

- `Commit()` помечает транзакцию как успешную. Он вызывается только в том случае, если сама бизнес-операция завершилась успехом, а все модификации данных готовы к сохранению.
- `Dispose()` завершает транзакцию. Он вызывается независимо от хода бизнес-операции после ее завершения. Если метод `Commit()` был ранее вызван, `Dispose()` сохраняет все обновления данных; в противном случае эти обновления отменяются.

Такая комбинация `Commit()` и `Dispose()` гарантирует, что база данных изменяется только на *позитивных путях* (успешное выполнение бизнес-сценария). Вот почему вызов `Commit()` располагается в самом конце метода `ChangeEmail()`. При возникновении любой ошибки, будь то ошибка валидации или необработанное исключение, логика выполнения вернет управление до вызова `Commit()` и таким образом предотвратит закрепление транзакции.

`Commit()` вызывается контроллером, потому что этот вызов требует принятия решений. Однако вызов `Dispose()` не подразумевает принятия решений, поэтому этот вызов можно делегировать классу из слоя инфраструктуры. Тот же класс, который создал экземпляр контроллера и предоставил ему необходимые зависимости, также должен уничтожить транзакцию после завершения работы контроллера.

Обратите внимание: `UserRepository` требует передачи `Transaction` в параметре конструктора. Это явно указывает, что репозитории всегда работают поверх транзакций; репозиторий не может обращаться с вызовом к базе данных самостоятельно.

Обновление транзакции до unit of work

Использование репозиториев и транзакции — хороший способ предотвращения потенциального нарушения целостности данных, но существует лучший способ. Класс `Transaction` можно обновить до *unit of work* (единицы работы).

ОПРЕДЕЛЕНИЕ

Unit of work ведет список объектов, измененных бизнес-операцией. После того как операция завершена, unit of work определяет все обновления, которые необходимо выполнить для изменения базы данных, и проводит эти обновления как единое целое.

Главное преимущество паттерна unit of work над простой транзакцией — возможность отложенных обновлений. В отличие от транзакций, unit of work выполняет все обновления в конце бизнес-операции, сводя к минимуму продолжительность нежелющей транзакции и увеличивая пропускную способность приложения (рис. 10.6). Часто этот паттерн также помогает сократить количество обращений к базе данных.

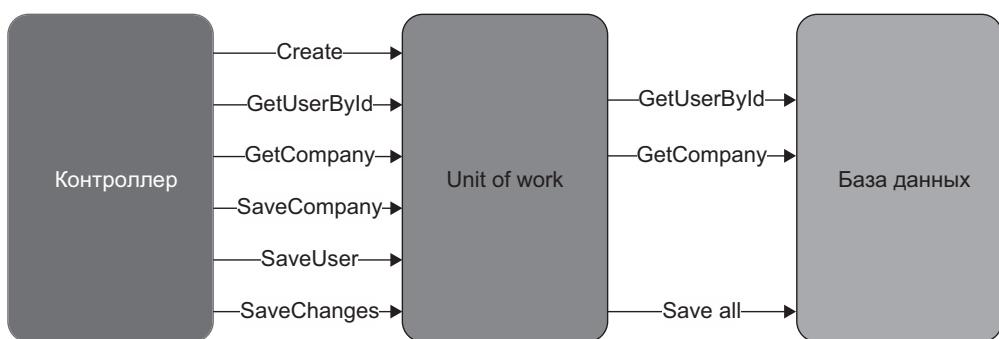


Рис. 10.6. Unit of work выполняет все обновления в конце бизнес-операции.

Обновления по-прежнему упаковываются в транзакцию, но эта транзакция существует в течение более короткого периода времени, тем самым увеличивая пропускную способность приложения

ПРИМЕЧАНИЕ

Транзакции баз данных также реализуют паттерн unit of work.

Может показаться, что ведение списка измененных объектов и последующая генерация SQL-скрипта — большой объем работы. Однако в действительности вам не нужно выполнять эту работу самостоятельно. Многие библиотеки объектно-реляционного отображения (ORM) реализуют паттерн «Unit of Work» за вас. Например, в .NET можно воспользоваться NHibernate или Entity Framework; оба фреймворка предоставляют классы для выполнения этой черновой работы (`ISession` и `DbContext` соответственно). В листинге 10.4 показано, как выглядит класс `UserController` в сочетании с Entity Framework.

Листинг 10.4. UserController с Entity Framework

```

public class UserController
{
    private readonly CrmContext _context;
    private readonly UserRepository _userRepository;
    private readonly CompanyRepository _companyRepository;
    private readonly EventDispatcher _eventDispatcher;

    public UserController(
        CrmContext context,
        MessageBus messageBus,
        IDomainLogger domainLogger)
    {
        _context = context;
        _userRepository = new UserRepository(
            context);
        _companyRepository = new CompanyRepository(
            context);
        _eventDispatcher = new EventDispatcher(
            messageBus, domainLogger);
    }

    public string ChangeEmail(int userId, string newEmail)
    {
        User user = _userRepository.GetUserById(userId);

        string error = user.CanChangeEmail();
        if (error != null)
            return error;

        Company company = _companyRepository.GetCompany();

        user.ChangeEmail(newEmail, company);

        _companyRepository.SaveCompany(company);
        _userRepository.SaveUser(user);
        _eventDispatcher.Dispatch(user.DomainEvents);

        _context.SaveChanges(); ← CrmContext заменяет Transaction
        return "OK";
    }
}

```

`CrmContext` – специальный класс, содержащий отображение между моделью предметной области и базой данных (он наследует от класса Entity Framework `DbContext`). Контроллер в листинге 10.4 использует `CrmContext` вместо `Transaction`. В результате:

- оба репозитория теперь работают на базе `CrmContext` подобно тому, как они работали на базе `Transaction` в предыдущей версии;
- контроллер закрепляет изменения в базе данных вызовом `context.SaveChanges()` вместо `transaction.Commit()`.

В **UserFactory** и **CompanyFactory** больше нет необходимости, потому что Entity Framework теперь обеспечивает отображение между данными из базы и доменными объектами.

НАРУШЕНИЕ ЦЕЛОСТНОСТИ ДАННЫХ В НЕРЕЛЯЦИОННЫХ БАЗАХ ДАННЫХ

Избежать нарушений целостности данных при использовании реляционной базы данных несложно: все популярные реляционные базы данных предоставляют атомарные обновления, которые могут охватывать любое количество строк. Но как добиться того же уровня защиты с нереляционными базами данных — например, MongoDB?

Проблема многих нереляционных баз данных — отсутствие транзакций в классическом смысле; атомарность обновлений гарантируется только в пределах одного документа. Если бизнес-операция затрагивает несколько документов, появляется риск нарушения целостности данных. (В нереляционных базах данных документ является эквивалентом строки.)

Нереляционные базы данных подходят к нарушениям целостности данных с другой стороны: они требуют проектировать документы так, чтобы никакая бизнес-операция не изменяла более одного такого документа за раз. Это возможно благодаря тому, что документы обладают большей гибкостью, чем строки реляционных баз данных. В одном документе могут храниться данные любого формата и сложности, что позволяет ему вмещать изменения даже самых сложных бизнес-операций.

В области предметно-ориентированного проектирования (DDD) существует рекомендация, которая гласит, что одна бизнес-операция должна изменять не более одного агрегата. Эта рекомендация служит той же цели: защите от нарушений целостности данных. Впрочем, она применима только к системам, работающим с документными базами данных, в которых один документ соответствует одному агрегату.

10.2.2. Управление транзакциями в интеграционных тестах

Что касается управления транзакциями в интеграционных тестах, руководствуйтесь следующим принципом: *транзакции баз данных или экземпляры unit of work не должны переиспользоваться в разных секциях теста*. В листинге 10.5 приведен пример переиспользования `CrmContext` в интеграционном teste после перевода этого теста на использование Entity Framework.

Листинг 10.5. Интеграционный тест с использованием CrmContext

```

var userRepository =
    new UserRepository(context);
var companyRepository =
    new CompanyRepository(context);
var user = new User(0, "user@mycorp.com",
    UserType.Employee, false);
userRepository.SaveUser(user);
var company = new Company("mycorp.com", 1);
companyRepository.SaveCompany(company);
context.SaveChanges();

var busSpy = new BusSpy();
var messageBus = new MessageBus(busSpy);
var loggerMock = new Mock<IDomainLogger>();
var sut = new UserController(
    context,           ← ...в секции действия...
    messageBus,
    loggerMock.Object);

// Act
string result = sut.ChangeEmail(user.UserId, "new@gmail.com");

// Assert
Assert.Equal("OK", result);

User userFromDb = userRepository           | ...и в секции
    .GetUserById(user.UserId);           | действия
Assert.Equal("new@gmail.com", userFromDb.Email);
Assert.Equal(UserType.Customer, userFromDb.Type);

Company companyFromDb = companyRepository | ...и в секции
    .GetCompany();                     | проверки
Assert.Equal(0, companyFromDb.NumberOfEmployees);

busSpy.ShouldSendNumberOfMessages(1)
    .WithEmailChangedMessage(user.UserId, "new@gmail.com");
loggerMock.Verify(
    x => x.UserTypeHasChanged(
        user.UserId, UserType.Employee, UserType.Customer),
    Times.Once);
}
}

```

Тест использует один экземпляр `CrmContext` во всех трех секциях: подготовки, действия и проверки. И это создает проблему, потому что такое переиспользование создает окружение, не соответствующее тому, что контроллер ожидает увидеть в рабочей среде. В рабочей среде каждая бизнес-операция обладает эксклюзивным контролем над `CrmContext`. Этот экземпляр создается непосредственно перед вызовом метода контроллера и уничтожается сразу же после него.

Чтобы избежать риска несогласованного поведения, интеграционные тесты должны воспроизводить тестовую среду как можно ближе к рабочей; это означает, что

секция действий не должна использовать `CrmContext` совместно с кем-либо. Секции подготовки и проверки тоже должны работать с отдельными экземплярами `CrmContext`, потому что, как вы помните из главы 8, важно проверять состояние базы данных независимо от данных, служивших входными параметрами. Несмотря на то что секция проверки запрашивает информацию пользователя и компании независимо от секции подготовки, они совместно используют один контекст базы данных (*unit of work*). Этот контекст может кэшировать запрашиваемые данные для повышения быстродействия.

СОВЕТ

В интеграционных тестах используйте как минимум три экземпляра *unit of work*: по одному для секций подготовки, действия и проверки.

10.3. Жизненный цикл тестовых данных

Совместная база данных создает проблему изоляции интеграционных тестов друг от друга. Чтобы решить эту проблему, необходимо:

- выполнять интеграционные тесты последовательно;
- удалять оставшиеся данные между запусками тестов.

Ваши тесты не должны зависеть от состояния базы данных. Они должны сами приводить базу к нужному им состоянию.

10.3.1. Параллельное или последовательное выполнение тестов?

Параллельное выполнение интеграционных тестов сопряжено со значительными усилиями. Вы должны следить за тем, чтобы все тестовые данные были уникальными, чтобы ограничения уровня базы не нарушались, а тесты случайно не использовали данные, оставшиеся после других тестов. Очистка тестовых данных также усложняется. Интеграционные тесты лучше выполнять последовательно. Не тратьте время на попытки выжать из них дополнительное быстродействие.

Многие фреймворки юнит-тестирования позволяют определять отдельные коллекции тестов и избирательно блокировать их параллелизацию. Создайте две такие коллекции (для юнит- и интеграционных тестов), а затем заблокируйте параллелизацию в коллекции с интеграционными тестами.

В качестве альтернативы можно параллелизировать тесты при помощи контейнеров. Например, можно поместить эталонную базу данных в образ Docker и создавать на основе этого образа экземпляр нового контейнера для каждого интеграционного

теста. Однако на практике этот метод слишком затратный в сопровождении. С Docker вам придется не только отслеживать изменения самой базы, но также:

- поддерживать образы Docker;
- отследить за тем, чтобы каждый тест получал собственный экземпляр контейнера;
- группировать интеграционные тесты (потому что вам, скорее всего, не удастся создать контейнеры для всех тестов сразу);
- удалять использованные контейнеры.

Я не рекомендую использовать контейнеры, если только у вас нет необходимости любой ценой минимизировать время выполнения ваших интеграционных тестов. Гораздо практичнее иметь только один экземпляр базы на разработчика. Впрочем, этот один экземпляр можно запускать в Docker. Я выступаю против преждевременной параллелизации, а не против технологии Docker как таковой.

10.3.2. Очистка данных между запусками тестов

Существуют четыре способа очистки остатков данных между запусками тестов.

- *Восстановление резервной копии базы данных перед каждым тестом* — такой подход решает проблему очистки данных, но он намного медленнее трех других. Даже с контейнерами удаление экземпляра контейнера и создание нового экземпляра обычно занимает несколько секунд, что приводит к быстрому увеличению времени выполнения тестов.
- *Очистка данных в конце теста* — этот метод быстр, но с ним существует риск пропуска фазы очистки. Если в середине теста на сервере сборки произойдет сбой или тест будет завершен в отладчике, тестовые данные останутся в базе и повлияют на дальнейшие запуски тестов.
- *Упаковка каждого теста в транзакцию без ее закрепления* — в этом случае все изменения, вносимые тестом и тестовой средой, автоматически отменяются. Такой подход решает проблему пропуска фазы очистки, но создает другую проблему: добавление транзакции может привести к несоответствию поведения между рабочей и тестовой средой. Здесь проявляется та же проблема, что и при переиспользовании экземпляра *unit of work*: дополнительная транзакция создает окружение, которое отличается от рабочего.
- *Очистка данных в начале теста* — лучший вариант. Он быстр, не приводит к непоследовательному поведению и не подвержен случайному пропуску фазы очистки.

СОВЕТ

Нет необходимости в отдельной фазе очистки — она реализуется как часть секции подготовки.

Удаление данных само по себе должно выполняться в определенном порядке для соблюдения ограничений базы данных (foreign key constraints). Иногда можно видеть, как разработчики используют хитроумные алгоритмы для определения связей между таблицами и автоматического генерирования скрипта удаления или даже удаляют все foreign key constraints с их последующим восстановлением. Все это лишнее. Напишите SQL-скрипт вручную: этот способ проще и предоставляет вам больше контроля над процессом удаления.

Добавьте базовый класс для всех интеграционных тестов и разместите в нем скрипт удаления тестовых данных. С таким базовым классом скрипт будет выполняться автоматически в начале каждого теста, как показано в листинге 10.6.

Листинг 10.6. Базовый класс для интеграционных тестов

```
public abstract class IntegrationTests
{
    private const string ConnectionString = "...";

    protected IntegrationTests()
    {
        ClearDatabase();
    }

    private void ClearDatabase()
    {
        string query =
            "DELETE FROM dbo.[User];" +           | Скрипт
            "DELETE FROM dbo.Company;" +          | удаления

        using (var connection = new SqlConnection(ConnectionString))
        {
            var command = new SqlCommand(query, connection)
            {
                CommandType = CommandType.Text
            };

            connection.Open();
            command.ExecuteNonQuery();
        }
    }
}
```

СОВЕТ

Скрипт удаления должен удалять все обычные данные, но не справочные данные. Справочными данными, равно как и всеми остальными элементами схемы базы данных, должны управлять исключительно миграции.

10.3.3. Не используйте базы данных в памяти

Другой способ изоляции интеграционных тестов друг от друга основан на замене базы данных ее аналогом, находящимся в памяти, — например, SQLite. Идея использования базы данных в памяти может показаться заманчивой, потому что такие базы данных:

- не требуют удаления тестовых данных;
- быстрее работают;
- могут создаваться перед каждым тестом.

Поскольку базы данных в памяти не являются совместными зависимостями, интеграционные тесты фактически становятся юнит-тестами (при условии что база данных является единственной управляемой зависимостью в проекте) по аналогии с контейнерами, о которых говорилось в разделе 10.3.1.

Несмотря на все эти преимущества, я не рекомендую использовать базы данных в памяти, потому что их функциональность сильно отличается от традиционных баз данных. А значит, здесь снова возникает проблема несоответствия между рабочей и тестовой средой. Ваши тесты могут выдавать ложные срабатывания или (что еще хуже) ложнопозитивные срабатывания из-за различий между традиционными базами и базами данных в памяти. Такие тесты никогда не обеспечат хорошей защиты, и в конечном итоге вам придется проводить регрессионное тестирование вручную.

СОВЕТ

Используйте в тестах ту же систему управления базами данных (СУБД), что и в рабочей версии. Номера версий могут различаться — обычно это приемлемо, но вендор СУБД должен быть одним и тем же.

10.4. Переиспользование кода в секциях тестов

Интеграционные тесты иногда быстро разрастаются, что приводит к ухудшению их сопровождаемости. Страйтесь, чтобы ваши интеграционные тесты были короткими — но без связывания их друг с другом и без ущерба для читаемости. Даже самые короткие тесты не должны зависеть друг от друга. Они также должны сохранять полный контекст тестового сценария и не должны заставлять разработчика анализировать разные части классов тестов, чтобы разобраться в происходящем.

Лучший способ сокращения размера тестов — выделение технических, не имеющих отношения к бизнес-операциям частей в приватные методы или вспомогательные классы. Дополнительно вы получаете возможность переиспользования этих фрагментов. В этом разделе я покажу, как сократить все три секции теста: подготовки, действия и проверки.

10.4.1. Переиспользование кода в секциях подготовки

В листинге 10.7 показано, как выглядит наш интеграционный тест после предоставления отдельного контекста базы данных (экземпляра unit of work) каждой из ее секций.

Листинг 10.7. Интеграционные тесты с тремя контекстами базы данных

```
[Fact]
public void Changing_email_from_corporate_to_non_corporate()
{
    // Arrange
    User user;

    using (var context = new CrmContext(ConnectionString))
    {
        var userRepository = new UserRepository(context);
        var companyRepository = new CompanyRepository(context);
        user = new User(0, "user@mycorp.com",
                       UserType.Employee, false);
        userRepository.SaveUser(user);
        var company = new Company("mycorp.com", 1);
        companyRepository.SaveCompany(company);

        context.SaveChanges();
    }

    var busSpy = new BusSpy();
    var messageBus = new MessageBus(busSpy);
    var loggerMock = new Mock<IDomainLogger>();

    string result;
    using (var context = new CrmContext(ConnectionString))
    {
        var sut = new UserController(
            context, messageBus, loggerMock.Object);

        // Act
        result = sut.ChangeEmail(user.UserId, "new@gmail.com");
    }

    // Assert
    Assert.Equal("OK", result);

    using (var context = new CrmContext(ConnectionString))
    {
        var userRepository = new UserRepository(context);
        var companyRepository = new CompanyRepository(context);

        User userFromDb = userRepository.GetUserById(user.UserId);
        Assert.Equal("new@gmail.com", userFromDb.Email);
        Assert.Equal(UserType.Customer, userFromDb.Type);
    }
}
```

```

    Company companyFromDb = companyRepository.GetCompany();
    Assert.Equal(0, companyFromDb.NumberOfEmployees);

    busSpy.ShouldSendNumberOfMessages(1)
        .WithEmailChangedMessage(user.UserId, "new@gmail.com");
    loggerMock.Verify(
        x => x.UserTypeHasChanged(
            user.UserId, UserType.Employee, UserType.Customer),
        Times.Once);
}
}

```

Как вы, возможно, помните из главы 3, для переиспользования кода между секциями подготовки лучше всего добавить приватные фабричные методы. Например, листинг 10.8 создает пользователя.

Листинг 10.8. Отдельный метод, создающий пользователя

```

private User CreateUser(
    string email, UserType type, bool isEmailConfirmed)
{
    using (var context = new CrmContext(ConnectionString))
    {
        var user = new User(0, email, type, isEmailConfirmed);
        var repository = new UserRepository(context);
        repository.SaveUser(user);

        context.SaveChanges();

        return user;
    }
}

```

OBJECT MOTHER И TEST DATA BUILDER

Паттерн, представленный в листинге 10.9 и 10.10, называется Object Mother («Мать объектов»). Это класс или метод, который помогает создавать тестовые данные.

Существует и другой паттерн, который тоже помогает переиспользовать код в секциях подготовки: Test Data Builder («Построитель тестовых данных»). Он похож на паттерн Object Mother, но при этом предоставляет fluent-интерфейс вместо простых методов. Пример использования паттерна Test Data Builder:

```

User user = new UserBuilder()
    .WithEmail("user@mycorp.com")
    .WithType(UserType.Employee)
    .Build();

```

Test Data Builder слегка улучшает удобочитаемость тестов, но требует слишком большого количества дополнительного кода. По этой причине я рекомендую придерживаться паттерна Object Mother (по крайней мере в C#, где необязательные аргументы стали одной из возможностей языка).

Также можно определить значения по умолчанию для аргументов метода, как показано в листинге 10.9.

Листинг 10.9. Добавление значений по умолчанию для фабрики

```
private User CreateUser(  
    string email = "user@mycorp.com",  
    UserType type = UserType.Employee,  
    bool isEmailConfirmed = false)  
{  
    /* ... */  
}
```

Со значениями по умолчанию можно задать аргументы избирательно, чтобы сократить тест еще больше. Избирательное использование аргументов также подчеркивает, какие из этих аргументов актуальны для тестового сценария.

Листинг 10.10. Использование фабричного метода

```
User user = CreateUser(  
    email: "user@mycorp.com",  
    type: UserType.Employee);
```

Где размещать фабричные методы

Когда вы начинаете выделять наиболее существенные аспекты тестов и выделять технические подробности в фабричные методы, вы сталкиваетесь с вопросом о том, где размещать эти методы. Должны ли они находиться в одном классе с тестами? В базовом классе `IntegrationTests`? А может, в отдельном вспомогательном классе?

Начните с простого варианта — размещения фабричных методов в том же классе. Перемещайте их в отдельные вспомогательные классы только в том случае, если дублирование кода создает серьезную проблему. Не размещайте фабричные методы в базовом классе; зарезервируйте этот класс для кода, который должен выполняться в каждом teste (например, для кода очистки данных).

10.4.2. Переиспользование кода в секциях действий

Каждая секция действий в интеграционных тестах требует создания отдельной транзакции или экземпляра unit of work. В настоящее время секция действия из листинга 10.7 выглядит так:

```
string result;  
using (var context = new CrmContext(ConnectionString))  
{  
    var sut = new UserController(  
        context, messageBus, loggerMock.Object);
```

```
// Act
result = sut.ChangeEmail(user.UserId, "new@gmail.com");
}
```

Эту секцию тоже можно сократить. Вы можете добавить метод, который принимает на вход делегат с информацией о том, какая функция контроллера должна быть вызвана. Метод декорирует обращение к контроллеру созданием контекста базы данных, как показано в листинге 10.11.

Листинг 10.11. Метод-декоратор

```
private string Execute(
    Func<UserController, string> func, ← Делегат определяет функцию контроллера
    MessageBus messageBus,
    IDomainLogger logger)
{
    using (var context = new CrmContext(ConnectionString))
    {
        var controller = new UserController(
            context, messageBus, logger);
        return func(controller);
    }
}
```

С таким методом секция действия теста сокращается до пары строк:

```
string result = Execute(
    x => x.ChangeEmail(user.UserId, "new@gmail.com"),
    messageBus, loggerMock.Object);
```

10.4.3. Переиспользование кода в секциях проверки

Наконец, секцию проверки тоже можно сократить. Проще всего для этого ввести вспомогательные методы, сходные с `CreateUser` и `CreateCompany`, как показано в листинге 10.12.

Листинг 10.12. Проверка данных после выделения логики запросов

```
User userFromDb = QueryUser(user.UserId);
Assert.Equal("new@gmail.com", userFromDb.Email);
Assert.Equal(UserType.Customer, userFromDb.Type);

Company companyFromDb = QueryCompany();
Assert.Equal(0, companyFromDb.NumberOfEmployees);
```

Можно пойти еще дальше и создать fluent-интерфейс для этих проверок (по аналогии с тем, что было сделано в главе 9 для `BusSpy`). На языке C# fluent-интерфейс поверх существующих классов предметной области может быть реализован с использованием методов-расширений, как показано в листинге 10.13.

Листинг 10.13. Fluent-интерфейс для проверки данных

```
public static class UserExtensions
{
    public static User ShouldExist(this User user)
    {
        Assert.NotNull(user);
        return user;
    }

    public static User WithEmail(this User user, string email)
    {
        Assert.Equal(email, user.Email);
        return user;
    }
}
```

С этим fluent-интерфейсом тестовые проверки читаются намного проще:

```
User userFromDb = QueryUser(user.UserId);
userFromDb
    .ShouldExist()
    .WithEmail("new@gmail.com")
    .WithType(UserType.Customer);

Company companyFromDb = QueryCompany();
companyFromDb
    .ShouldExist()
    .WithNumberOfEmployees(0);
```

10.4.4. Не создает ли тест слишком много транзакций?

После всех упрощений, описанных выше, интеграционный тест стал более читаемым — следовательно, простым в сопровождении. Впрочем, у этой версии также есть недостаток: тест теперь создает пять транзакций (экземпляров unit of work), тогда как ранее использовал только три (листинг 10.14).

Является ли увеличение количества транзакций проблемой? И если да, то что с этим делать? Да, дополнительные контексты создают некоторые проблемы, потому что они замедляют выполнение теста, но с этим мало что можно сделать. Перед нами еще один пример компромисса между разными атрибутами эффективного теста — на этот раз между быстротой обратной связи и простотой поддержки. В данном конкретном случае есть смысл пойти на этот компромисс и пожертвовать быстрым действием ради простоты поддержки. Снижение быстродействия здесь не должно быть сколько-нибудь существенным, особенно если база данных находится на машине разработчика. В то же время выигрыш по сопровождаемости оказывается весьма значительным.

Листинг 10.14. Интеграционный тест после вынесения всех технических подробностей

```
public class UserControllerTests : IntegrationTests
{
    [Fact]
    public void Changing_email_from_corporate_to_non_corporate()
    {
        // Arrange
        User user = CreateUser(
            email: "user@mycorp.com",
            type: UserType.Employee);
        CreateCompany("mycorp.com", 1);

        var busSpy = new BusSpy();
        var messageBus = new MessageBus(busSpy);
        var loggerMock = new Mock<IDomainLogger>();

        // Act
        string result = Execute(
            x => x.ChangeEmail(user.UserId, "new@gmail.com"),
            messageBus, loggerMock.Object);

        // Assert
        Assert.Equal("OK", result);

        User userFromDb = QueryUser(user.UserId);
        userFromDb
            .ShouldExist()
            .WithEmail("new@gmail.com")
            .WithType(UserType.Customer);

        Company companyFromDb = QueryCompany();
        companyFromDb
            .ShouldExist()
            .WithNumberOfEmployees(0);

        busSpy.ShouldSendNumberOfMessages(1)
            .WithEmailChangedMessage(user.UserId, "new@gmail.com");
        loggerMock.Verify(
            x => x.UserTypeHasChanged(
                user.UserId, UserType.Employee, UserType.Customer),
            Times.Once);
    }
}
```

10.5. Типичные вопросы при тестировании баз данных

В последней части этой главы я хочу ответить на некоторые вопросы, часто возникающие при тестировании баз данных, а также кратко напомнить некоторые важные обстоятельства, упоминавшиеся в главах 8 и 9.

10.5.1. Нужно ли тестировать операции чтения?

В нескольких последних главах использовался сценарий изменения имейла. Это пример операции записи (операции, которая оставляет побочный эффект (изменение) в базе данных и других внепроцессных зависимостях). Многие приложения содержат как операции записи, так и операции чтения. Пример операции чтения — возврат информации о пользователе внешнему клиенту. Нужно ли тестировать такие операции чтения?

Очень важно тщательно тестировать операции записи, потому что ставки высоки. Ошибка записи часто приводит к повреждению данных, что может отразиться не только на вашей базе данных, но и на внешних приложениях. Тесты, покрывающие операции записи, обладают высокой эффективностью из-за защиты, предоставляемой ими против таких ошибок.

Но это в меньшей степени относится к операциям чтения: ошибка при операции чтения обычно не имеет таких разрушительных последствий. Следовательно, порог для тестирования операций чтения должен быть выше, чем для записи. Тестируйте только самые сложные или важные операции чтения, остальные можно игнорировать.

Также обратите внимание, что при чтении не нужна доменная модель. Одной из главных целей моделирования домена (предметной области) является инкапсуляция. И как вы, вероятно, помните из глав 5 и 6, суть инкапсуляции — сохранение целостности данных в свете любых изменений. Так как при чтении отсутствуют изменения данных, инкапсуляция таких операций бессмысленна. Более того, для чтения вам не понадобится полнофункциональная ORM-система — такая как NHibernate или Entity Framework. Лучше ограничиться простым SQL-скриптом. Такой скрипт также будет работать быстрее, чем ORM, благодаря отсутствию лишних уровней абстракции (рис. 10.7).

Так как в операциях чтениях обычно не задействованы уровни абстракции (а доменная модель относится к их числу), юнит-тесты здесь пользы не принесут. Если вы решите тестировать операции чтения, сделайте это, используя интеграционные тесты в комбинации с реальной базой данных.

10.5.2. Нужно ли тестировать репозитории?

Репозитории формируют полезную абстракцию поверх базы данных. Пример использования репозиториев из нашего проекта CRM:

```
User user = _userRepository.GetUserById(userId);
 userRepository.SaveUser(user);
```

Нужно ли тестировать репозитории независимо от других интеграционных тестов? Может показаться, что было бы полезно протестировать отображение (mapping)

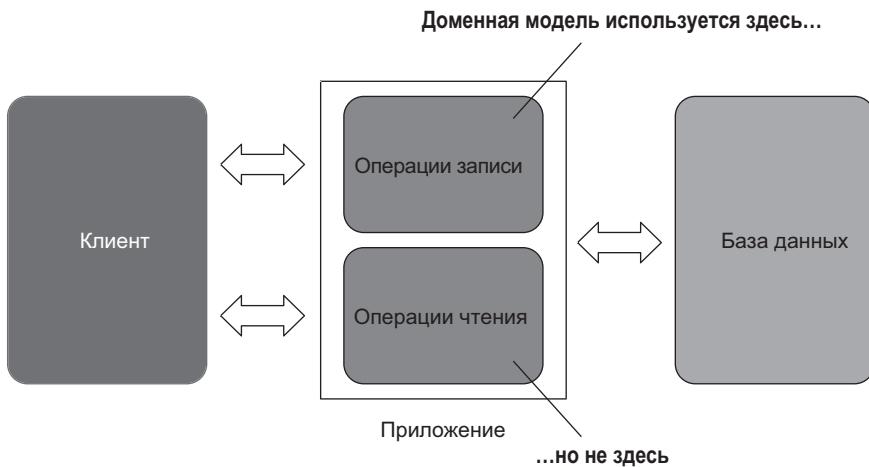


Рис. 10.7. В операциях чтения модель предметной области не нужна. А поскольку цена ошибки при чтении меньше, чем при записи, необходимость в интеграционном тестировании также будет ниже

объектов предметной области на базу данных. В конце концов, в этой функциональности могут присутствовать ошибки. Тем не менее такие тесты не добавят пользы проекту из-за повышенных затрат на сопровождение и ухудшенной защиты от багов. Обсудим эти два недостатка более подробно.

Высокие затраты на сопровождение

На диаграмме классификации кода из главы 7 репозитории относятся к четверти контроллеров (рис. 10.8). Они обладают низкой сложностью и взаимодействуют с внепроцессной зависимостью — базой данных. Именно присутствие внепроцессной зависимости увеличивает затраты на сопровождение тестов.

Что касается простоты поддержки, тестирование репозиториев требует таких же затрат на сопровождение, как и обычные интеграционные тесты. Но обеспечивает ли оно такую же защиту от багов? К сожалению, нет.

Ухудшенная защита от багов

Репозитории не обладают особой сложностью, и значительная часть выигрыша в защите от багов перекрывается с защитой, предоставляемой обычными интеграционными тестами. Таким образом, тесты репозиториев не дают существенной дополнительной пользы.

Оптимальный путь при тестировании репозитория — выделение той незначительной сложности, которой он обладает, в автономный алгоритм и последующее

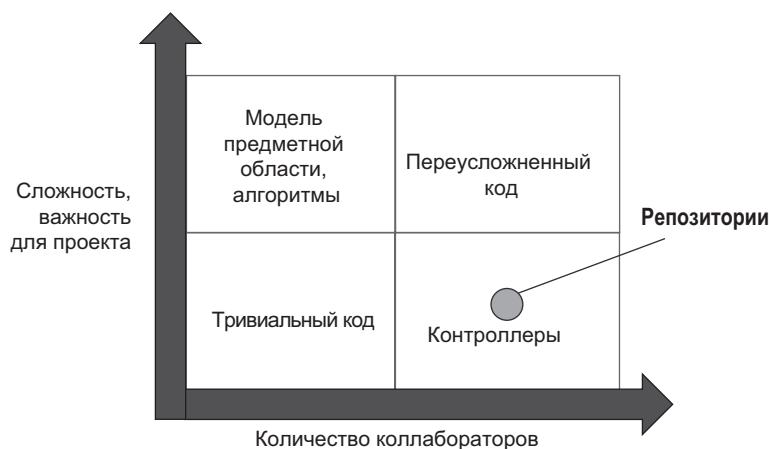


Рис. 10.8. Репозитории обладают низкой сложностью и взаимодействуют с внепроцессной зависимостью, вследствие чего попадают в четверть контроллеров на диаграмме классификации кода

тестирование этого алгоритма. Именно для этого были предназначены классы `UserFactory` и `CompanyFactory` в предыдущих главах. Эти два класса содержали отображения (mappings), но сами не работали с колабораторами. Репозитории же (класс `Database`) содержали только простые запросы SQL.

К сожалению, такое разделение между отображением данных (которое ранее обеспечивалось фабриками) и взаимодействиями с базой данных (которое ранее выполнялось `Database`) становится невозможным при использовании ORM. Отображения ORM невозможно протестировать без обращения к базе данных, по крайней мере не без ущерба для устойчивости к рефакторингу. А следовательно, необходимо придерживаться следующего правила: *не тестируйте репозитории напрямую, только в составе общих интеграционных тестов*.

Также не тестируйте отдельно `EventDispatcher` (этот класс преобразует события предметной области в обращения к неуправляемым зависимостям). Выигрыш по защите от ошибок слишком мал по сравнению с высокими затратами, необходимыми для сопровождения моков.

10.6. Заключение

Хорошо продуманные тесты базы данных обеспечивают надежную защиту от ошибок. По моему опыту, это один из самых эффективных инструментов, без которого невозможно быть уверенным в качестве продукта. Такие тесты оказывают большую

помощь при рефакторинге базы данных, переходе на другую технологию ORM или базу данных от другого вендора.

Наш проект перешел на Entity Framework ORM ранее в этой главе, и мне потребовалось изменить всего пару строк в интеграционном тесте, чтобы убедиться в правильности рефакторинга. Интеграционные тесты, работающие напрямую с управляемыми зависимостями, — самый эффективный способ защиты от ошибок, возникающих при масштабном рефакторинге.

Итоги

- Храните схему базы данных в системе контроля версий наряду с вашим исходным кодом. Схема базы данных состоит из таблиц, представлений, индексов, хранимых процедур и вообще всего, что составляет описание построения базы данных.
- Справочные данные также являются частью схемы базы данных. Эти данные должны быть предварительно заполнены, чтобы приложение могло нормально функционировать. Чтобы отличить справочные данные от обычных, определите, может ли ваше приложение изменить эти данные. Если может — это обычные данные; в противном случае это справочные данные.
- Создайте отдельный экземпляр базы данных для каждого разработчика. Еще лучше, если этот экземпляр будет размещаться на машине разработчика для максимальной скорости выполнения тестов.
- Метод развертывания базы данных на основе состояния явно выражает состояние и позволяет утилите сравнения неявно управлять миграциями. Метод на основе миграций ориентирован на использование явных миграций, переводящих базу данных из одного состояния в другое. Явность состояния БД упрощает обработку конфликтов слияния, тогда как явные миграции помогают с трансформацией данных.
- Отдавайте предпочтение методу на основе миграций, а не методу на основе состояния, потому что обработка трансформаций данных намного важнее конфликтов слияния. Применяйте все модификации к схеме базы данных (включая справочные данные) посредством миграций.
- Бизнес-операции должны обновлять данные атомарно. Для обеспечения атомарности положитесь на механизм транзакций используемой базы данных.
- Используйте паттерн unit of work там, где это возможно. Unit of work полагается на транзакции используемой базы данных. Этот паттерн также откладывает все обновления до конца бизнес-операции, что способствует повышению быстродействия.

- Не переиспользуйте транзакции или экземпляры unit of work между секциями теста. Каждая секция подготовки, действия и проверки должна содержать собственную транзакцию или экземпляр unit of work.
- Выполняйте интеграционные тесты последовательно. Параллельное выполнение требует значительных усилий, которые обычно неоправданы.
- Проводите очистку тестовых данных в начале теста. Такое решение отрабатывает быстро, позволяет протестировать код в окружении, близком к продуктиву, и не может пропустить фазу очистки. С этим методом вам также не придется вводить отдельную фазу deinициализации (teardown).
- Избегайте баз данных в памяти (таких как SQLite). Вы никогда не обеспечите хорошей защиты, если ваши тесты будут тестировать базу данных другого вендора. Используйте в тестах ту же систему управления базами данных, что и в рабочей версии.
- Для сокращения размера тестов выделите несущественные части в приватные методы или вспомогательные классы:
 - для секции подготовки используйте паттерн Object Mother, а не Test Data Builder;
 - для секции действия создайте метод-декоратор;
 - для секции проверки используйте fluent-интерфейс.
- Порог для тестирования операций чтения должен быть выше, чем для операций записи. Тестируйте только самые сложные или важные операции чтения; остальные можно игнорировать.
- Не тестируйте репозитории напрямую, только как часть общих интеграционных тестов. Тесты репозиториев создают слишком высокие затраты на сопровождение при слишком незначительном выигрыше в защите от багов.

Часть IV

Антипаттерны юнит-тестирования

В последней части книги рассматриваются популярные антипаттерны юнит-тестирования. Скорее всего, вы уже сталкивались с ними в прошлом. Тем не менее будет интересно рассмотреть эту тему в контексте четырех атрибутов хорошего юнит-теста, определенных в главе 4. Эти атрибуты могут использоваться для анализа любых концепций или паттернов юнит-тестирования; антипаттерны не являются исключением.

11

Антипаттерны юнит-тестирования

В этой главе:

- ✓ Юнит-тестирование приватных методов.
- ✓ Раскрытие приватного состояния тестируемого класса.
- ✓ Утечка доменных знаний в тесты.
- ✓ Мокирование конкретных классов.

В этой главе приведена подборка тем (прежде всего антипаттернов), которые не вписывались ни в одну из предшествующих глав. *Антипаттерн* представляет собой популярное решение типичной задачи, которое на первый взгляд кажется уместным, но приводит к проблемам в будущем.

Вы узнаете, как работать с временем в тестах, а также научитесь выявлять и обходить такие антипаттерны, как юнит-тестирование приватных методов, загрязнение кода, мокирование конкретных классов и т. д. Многие из этих тем вытекают из основополагающих принципов, описанных в части 2. Тем не менее они заслуживают того, чтобы выразить их явно. Вероятно, ранее вы уже слышали хотя бы о некоторых из этих антипаттернов, но эта глава поможет вам сформировать цельную картину и понять, на чем они базируются.

11.1. Юнит-тестирование приватных методов

Один из самых распространенных вопросов в области юнит-тестирования звучит так: как тестировать приватные методы? Самый короткий ответ: этого делать вообще не следует. Однако данная тема сопряжена с целым рядом нюансов.

11.1.1. Приватные методы и хрупкость тестов

Не стоит делать публичными методы, которые иначе остались бы приватными, только для облегчения юнит-тестирования. Это нарушает один из фундаментальных принципов, описанных в главе 5: тестирование только наблюдаемого поведения. Раскрытие приватных методов ведет к привязке тестов к деталям имплементации и в конечном итоге вредит устойчивости ваших тестов к рефакторингу, что является самой важной метрикой из четырех. (На всякий случай напомню эти четыре метрики: защита от багов, устойчивость к рефакторингу, быстрая обратная связь и простота поддержки.) Вместо того чтобы тестировать приватные методы напрямую, тестируйте их косвенно, как часть наблюдаемого поведения.

11.1.2. Приватные методы и недостаточное покрытие

Иногда приватные методы оказываются слишком сложными, и тестирование их как части наблюдаемого поведения не обеспечивает достаточного покрытия. Если предположить, что наблюдаемое поведение уже обладает неплохим тестовым покрытием, такая ситуация возможна по двум причинам:

- *мертвый код*. Если не покрытый тестами код не используется, скорее всего, это лишний код, оставшийся после рефакторинга. Такой код лучше удалить;
- *отсутствие абстракции*. Если приватный метод слишком сложен (и, как следствие, протестировать его через публичный API класса слишком сложно), это указывает на отсутствие абстракции, которая должна быть выделена в отдельный класс.

Продемонстрирую вторую проблему на примере.

Листинг 11.1. Класс со сложным приватным методом

```
public class Order
{
    private Customer _customer;
    private List<Product> _products;

    public string GenerateDescription()
    {
        return $"Customer name: {_customer.Name}, " +
            $"total number of products: {_products.Count}, " +
            $"total price: {GetPrice()}"; ← Сложный приватный метод
    }

    private decimal GetPrice() ← Сложный приватный метод
    {
        decimal basePrice = /* Вычисление на основании _products */;
        decimal discounts = /* Вычисление на основании _customer */;
        decimal taxes = /* Вычисление на основании _products */;
        return basePrice - discounts + taxes;
    }
}
```

Сложный приватный метод
используется намного
более простым публичным
методом

Метод `GenerateDescription()` очень прост: он возвращает общее описание заказа. Но он использует приватный метод `GetPrice()`, который намного более сложен: он содержит важную бизнес-логику и нуждается в тщательном тестировании. Эта логика и является недостающей абстракцией. Вместо того чтобы раскрывать метод `GetPrice`, сделайте эту абстракцию явной, выделив ее в отдельный класс, как показано в листинге 11.2.

Листинг 11.2. Выделение сложного приватного метода

```
public class Order
{
    private Customer _customer;
    private List<Product> _products;

    public string GenerateDescription()
    {
        var calc = new PriceCalculator();

        return $"Customer name: {_customer.Name}, " +
            $"total number of products: {_products.Count}, " +
            $"total price: {calc.Calculate(_customer, _products)}";
    }
}

public class PriceCalculator
{
    public decimal Calculate(Customer customer, List<Product> products)
    {
        decimal basePrice = /* Вычисление на основании products */;
        decimal discounts = /* Вычисление на основании customer */;
        decimal taxes = /* Вычисление на основании products */;
        return basePrice - discounts + taxes;
    }
}
```

Теперь `PriceCalculator` можно тестировать независимо от `Order`. Вы также можете использовать функциональный стиль юнит-тестирования (тестирование выходных данных), потому что `PriceCalculator` не имеет никаких скрытых входных или выходных данных. За дополнительной информацией о стилях юнит-тестирования обращайтесь к главе 6.

11.1.3. Когда тестирование приватных методов допустимо

У правила, запрещающего тестирование приватных методов, есть исключения. Чтобы понять эти исключения, необходимо вернуться к связи между публичностью кода и его назначением (см. главу 5). В таблице 11.1 приведена краткая сводка этих отношений (эта таблица уже приводилась в главе 5, я скопировал ее для удобства).

Как вы, возможно, помните из главы 5, если сделать наблюдаемое поведение публичным, а подробности реализации приватными, вы получите хорошо спроектированный API. Утечка деталей имплементации вредит инкапсуляции кода. Комбинация наблюдаемого поведения и приватных методов помечена в таблице как «→», потому что для того, чтобы метод стал частью наблюдаемого поведения, он должен использоваться клиентским кодом, а это невозможно, когда метод является приватным.

Таблица 11.1. Связь между публичностью кода и его назначением

	Наблюдаемое поведение	Детали имплементации
Публичный код	Хорошо	Плохо
Приватный код	–	Хорошо

В тестировании приватных методов как таковом нет ничего плохого. Оно плохо только потому, что приватные методы служат посредниками для деталей имплементации. Тестирование деталей имплементации — вот что в конечном итоге приводит к хрупкости тестов. Тем не менее в отдельных редких случаях метод может одновременно быть и приватным, и частью наблюдаемого поведения (таким образом, пометка «→» в таблице 11.1 не совсем корректна).

Для примера возьмем систему, обрабатывающую заявки на получение кредита. Новые запросы загружаются большим пакетом в базу данных один раз в день. Затем администраторы просматривают эти запросы один за одним и решают, стоит ли одобрить их. В такой системе класс `Inquiry` может выглядеть так, как показано в листинге 11.3.

Приватный конструктор объявлен приватным, потому что класс восстанавливается из базы данных библиотекой объектно-реляционного отображения (ORM). Этой библиотеке не нужен открытый конструктор; она может нормально работать и с приватным. В то же время нашей системе конструктор тоже не нужен, потому что она не отвечает за создание этих заявок.

Как протестировать класс `Inquiry`, если вы не можете создавать его экземпляры? С одной стороны, логика одобрения заявок важна для проекта, а следовательно, должна быть протестирована. Но с другой стороны, объявление конструктора открытым нарушит правило о нераскрытии приватных методов.

Конструктор `Inquiry` — пример метода, который одновременно является и приватным, и частью наблюдаемого поведения. Этот конструктор является контрактом с ORM, и его приватность не делает этот контракт менее важным: без него библиотека ORM не сможет восстанавливать заявки из базы данных.

А следовательно, объявление конструктора `Inquiry` открытым не приведет к хрупкости тестов в этом конкретном примере. Более того, API класса даже улучшится и приблизится к состоянию хорошо спроектированного. Просто убедитесь в том, что конструктор содержит все предусловия, необходимые для поддержания его

инкапсуляции. В листинге 11.3 таким предусловием станет требование о том, чтобы во всех одобренных заявках присутствовало время одобрения.

Листинг 11.3. Класс с приватным конструктором

```
public class Inquiry
{
    public bool IsApproved { get; private set; }
    public DateTime? TimeApproved { get; private set; }

    private Inquiry(
        bool isApproved, DateTime? timeApproved) | Приватный
    {
        if (isApproved && !timeApproved.HasValue)
            throw new Exception();

        IsApproved = isApproved;
        TimeApproved = timeApproved;
    }

    public void Approve(DateTime now)
    {
        if (IsApproved)
            return;

        IsApproved = true;
        TimeApproved = now;
    }
}
```

Если же вы предпочитаете делать API класса как можно меньшим, экземпляр `Inquiry` можно создать с использованием механизма отражения (reflection) в тестах. Хотя такое решение на первый взгляд кажется хаком, вы всего лишь повторяете за ORM-библиотекой, которая уже использует reflection в своей внутренней реализации.

11.2. Раскрытие приватного состояния

Еще один распространенный антипаттерн — раскрытие приватного состояния для юнит-тестирования. Правило здесь то же, что и с приватными методами: не раскрывайте состояние, которое вы бы без этого предпочли оставить приватным, — тестируйте только наблюдаемое поведение. Взгляните на листинг 11.4.

В этом примере представлен класс `Customer`. Каждый новый клиент создается со статусом `Regular`, а затем может быть повышен до привилегированного статуса `Preferred`, при котором он получает 5%-ную скидку.

Как протестировать метод `Promote()`? Побочным эффектом этого метода является изменение поля `_status`, но само поле является приватным, а следовательно, недоступно в тестах. Разумное на первый взгляд решение — сделать это поле открытым

(`public`). В конце концов, разве изменение статуса не является конечной целью вызова `Promote()`?

Листинг 11.4. Класс с приватным состоянием

```
public class Customer
{
    private CustomerStatus _status = CustomerStatus.Regular; | Приватное
                                                               | состояние

    public void Promote()
    {
        _status = CustomerStatus.Preferred;
    }

    public decimal GetDiscount()
    {
        return _status == CustomerStatus.Preferred ? 0.05m : 0m;
    }
}

public enum CustomerStatus
{
    Regular,
    Preferred
}
```

Тем не менее такое изменение было бы антипаттерном. Помните, что ваши тесты должны взаимодействовать с тестируемой системой (SUT) точно так же, как рабочий код, не имея никаких особых привилегий. В листинге 11.4 поле `_status` скрыто от рабочего кода и не является частью наблюдаемого поведения SUT. Раскрытие этого поля привело бы к привязке тестов к деталям имплементации. Как же тогда тестировать `Promote()`?

Нужно посмотреть, как рабочий код использует этот класс. В этом конкретном примере статус клиента не важен для рабочего кода; в противном случае это поле было бы публичным. Единственное, что интересует рабочий код, — скидка, которую клиент получит после повышения статуса. А значит, именно это нужно проверять в тестах. Необходимо убедиться в том, что:

- только что созданный клиент не пользуется скидкой;
- после того как клиент будет повышен, скидка достигает 5 %.

Позднее, если рабочий код начнет использовать поле статуса клиента, вы сможете использовать это поле и в тестах, потому что оно официально станет частью наблюдаемого поведения тестируемой системы.

ПРИМЕЧАНИЕ

Расширение публичного API класса ради удобства его тестирования — плохая практика.

11.3. Утечка доменных знаний в тесты

Утечка доменных знаний (знаний предметной области) в тесты — еще один распространенный антипаттерн. Обычно она происходит в тестах, покрывающих сложные алгоритмы. Для примера возьмем следующий вычислительный алгоритм:

```
public static class Calculator
{
    public static int Add(int value1, int value2)
    {
        return value1 + value2;
    }
}
```

В листинге 11.5 показан *неправильный* способ его тестирования.

Листинг 11.5. Утечка реализации алгоритма

```
public class CalculatorTests
{
    [Fact]
    public void Adding_two_numbers()
    {
        int value1 = 1;
        int value2 = 3;
        int expected = value1 + value2;      ← Утечка

        int actual = Calculator.Add(value1, value2);

        Assert.Equal(expected, actual);
    }
}
```

Тест также можно параметризовать, чтобы включить пару дополнительных тестовых сценариев.

Листинг 11.6. Параметризованная версия того же теста

```
public class CalculatorTests
{
    [Theory]
    [InlineData(1, 3)]
    [InlineData(11, 33)]
    [InlineData(100, 500)]
    public void Adding_two_numbers(int value1, int value2)
    {
        int expected = value1 + value2;      ← Утечка

        int actual = Calculator.Add(value1, value2);

        Assert.Equal(expected, actual);
    }
}
```

Листинги 11.5 и 11.6 на первый взгляд выглядят нормально, но в действительности они являются примерами антипаттерна: эти тесты дублируют реализацию алгоритма из рабочего кода. Может показаться, что ничего страшного в этом нет, — всего-то одна строка. Но это объясняется только тем, что пример сильно упрощен. Я видел тесты, которые покрывали сложные алгоритмы и не делали ничего, кроме повторной реализации этих алгоритмов в секции подготовки. По сути они были копией рабочего кода.

Такие тесты — еще один пример привязки к деталям имплементации. Они обладают почти нулевой устойчивостью к рефакторингу и, как следствие, не эффективны. Такие тесты не смогут отличить настоящие ошибки от ложных срабатываний. Если в результате изменения алгоритма тесты перестанут проходить, разработчики с большой вероятностью просто скопируют новую версию этого алгоритма в тест, даже не пытаясь понять, почему тест стал падать.

Как же правильно протестировать алгоритм? *Тест не должен настаивать на конкретной реализации тестируемого алгоритма.* Вместо того чтобы дублировать алгоритм, зафиксируйте его результаты в teste, как показано в листинге 11.7.

Листинг 11.7. Тест без знаний предметной области

```
public class CalculatorTests
{
    [Theory]
    [InlineData(1, 3, 4)]
    [InlineData(11, 33, 44)]
    [InlineData(100, 500, 600)]
    public void Adding_two_numbers(int value1, int value2, int expected)
    {
        int actual = Calculator.Add(value1, value2);
        Assert.Equal(expected, actual);
    }
}
```

На первый взгляд это может показаться неправильным, но кодирование ожидаемого результата — хорошая практика в юнит-тестировании. При этом важно заранее вычислить этот результат с использованием чего-либо, кроме кода самой тестируемой системы (в идеале с помощью эксперта предметной области). Конечно, это относится только к достаточно сложным алгоритмам (мы все эксперты по сложению двух чисел). Как вариант, если вы рефакторите унаследованное (legacy) приложение, вы можете получить эти результаты с помощью старого кода, а затем воспользоваться ими, как ожидаемыми значениями в тестах.

11.4. Загрязнение кода

Перейдем к следующему антипаттерну — загрязнению кода.

Загрязнение кода часто принимает форму различных переключателей. Для примера возьмем логер.

ОПРЕДЕЛЕНИЕ

Загрязнение кода (code pollution) — добавление рабочего кода, который необходим только для тестирования.

Листинг 11.8. Логер с переключателем

```
public class Logger
{
    private readonly bool _isTestEnvironment;

    public Logger(bool isTestEnvironment)      ←———— Переключатель
    {
        _isTestEnvironment = isTestEnvironment;
    }

    public void Log(string text)
    {
        if (_isTestEnvironment)      ←———— Переключатель
            return;

        /* Логирование текста */
    }
}

public class Controller
{
    public void SomeMethod(Logger logger)
    {
        logger.Log("SomeMethod is called");
    }
}
```

В этом примере классу `Logger` в конструкторе передается параметр, который указывает, выполняется ли класс в рабочей среде. Если это так, то в файл записывается сообщение; в противном случае не происходит ничего. С таким переключателем можно выключить логирование во время тестовых запусков, как показано в листинге 11.9.

Листинг 11.9. Тест, использующий переключатель

```
[Fact]
public void Some_test()
{
    var logger = new Logger(true);      ←———— Параметру присваивается true —
    var sut = new Controller();          признак выполнения в тестовой среде

    sut.SomeMethod(logger);

    /* Проверка */
}
```

Проблема с загрязнением кода заключается в том, что тестовый код смешивается с рабочим кодом, что повышает затраты на сопровождение последнего. Чтобы избежать этого антипаттерна, следует вынести тестовый код из рабочего кода.

В примере с `Logger` добавьте интерфейс `ILogger` и создайте две его реализации: реальную для рабочей версии и фиктивную для тестовых целей. После этого переработайте класс `Controller`, чтобы он получал интерфейс вместо конкретного класса, как показано в листинге 11.10.

Листинг 11.10. Версия без переключателя

```
public interface ILogger
{
    void Log(string text);
}

public class Logger : ILogger
{
    public void Log(string text)
    {
        /* Логирование текста */
    }
}

public class FakeLogger : ILogger
{
    public void Log(string text)
    {
        /* Ничего не происходит */
    }
}

public class Controller
{
    public void SomeMethod(ILogger logger)
    {
        logger.Log("SomeMethod is called");
    }
}
```

Принадлежит
рабочему коду

Принадлежит
тестовому коду

Такое разделение упрощает код рабочего логера, потому что ему больше не нужно знать о тестовой среде. Обратите внимание: сам интерфейс `ILogger` может тоже рассматриваться как разновидность загрязнения кода: он находится в рабочей кодовой базе, но нужен только для тестирования. Чем же лучше новая реализация?

Загрязнение, вносимое `ILogger`, причиняет меньше вреда, и с ним проще справиться. В отличие от исходной реализации с `Logger`, теперь вы не сможете случайно активизировать выполнение ветки, не предназначенной для рабочего использования. Кроме того, интерфейсы не могут содержать ошибок, потому что это всего лишь

контракты без кода. В отличие от переключателей, вероятность ошибки с интерфейсами намного меньше.

11.5. Мокирование конкретных классов

До настоящего момента в этой книге приводились примеры мокирования с использованием интерфейсов, но существует и альтернативный подход: вы можете заменять моками конкретные классы и таким образом сохранить часть функциональности исходных классов, что иногда может быть полезно. Однако у этой альтернативы есть значительный недостаток: она нарушает принцип единственной ответственности. Эта идея продемонстрирована в листинге 11.11.

Листинг 11.11. Класс для вычисления статистики

```
public class StatisticsCalculator
{
    public (double totalWeight, double totalCost) Calculate(
        int customerId)
    {
        List<DeliveryRecord> records = GetDeliveries(customerId);
        double totalWeight = records.Sum(x => x.Weight);
        double totalCost = records.Sum(x => x.Cost);

        return (totalWeight, totalCost);
    }

    public List<DeliveryRecord> GetDeliveries(int customerId)
    {
        /* Обращение к внепроцессной зависимости
        для получения списка доставок */
    }
}
```

`StatisticsCalculator` собирает и вычисляет статистику клиента: вес и стоимость всех товаров, отправленных конкретному покупателю. Класс выполняет вычисления на основании списка товаров, полученного от внешнего сервиса (метод `GetDeliveries`). Также допустим, что существует контроллер, который использует класс `StatisticsCalculator` (листинг 11.12).

Листинг 11.12. Использование `StatisticsCalculator` контроллером

```
public class CustomerController
{
    private readonly StatisticsCalculator _calculator;

    public CustomerController(StatisticsCalculator calculator)
    {
        _calculator = calculator;
    }
}
```

```

public string GetStatistics(int customerId)
{
    (double totalWeight, double totalCost) = _calculator
        .Calculate(customerId);

    return
        $"Total weight delivered: {totalWeight}. " +
        $"Total cost: {totalCost}";
}
}

```

Как протестировать этот контроллер? Ему нельзя передать реальный экземпляр `StatisticsCalculator`, потому что этот экземпляр обращается к неуправляемой внешнепроцессной зависимости. Неуправляемая зависимость должна быть заменена стабом. В то же время полностью заменять `StatisticsCalculator` тоже нежелательно. Этот класс содержит важную вычислительную функциональность, которая должна оставаться неизменной.

Один из способов справиться с этой дилеммой основан на мокировании класса `StatisticsCalculator` и переопределении только метода `GetDeliveries()`, что можно сделать объявлением этого метода виртуальным, как в листинге 11.13.

Листинг 11.13. Тест, мокирующий конкретный класс

```

[Fact]
public void Customer_with_no_deliveries()
{
    // Arrange
    var stub = new Mock<StatisticsCalculator> { CallBase = true };
    stub.Setup(x => x.GetDeliveries(1))           ←
        .Returns(new List<DeliveryRecord>());
    var sut = new CustomerController(stub.Object);

    // Act
    string result = sut.GetStatistics(1);

    // Assert
    Assert.Equal("Total weight delivered: 0. Total cost: 0", result);
}

```

Метод `GetDeliveries()`
должен быть объявлен
виртуальным

Настройка `CallBase = true` позволяет моку сохранить поведение базового класса, если оно не было явно переопределено. С таким подходом можно заменить только часть класса, оставляя все остальное как есть. Как упоминалось ранее, это антипаттерн.

ПРИМЕЧАНИЕ

Необходимость в мокировании конкретного класса для сохранения части его функциональности является результатом нарушения принципа единственной ответственности.

`StatisticsCalculator` объединяет две ответственности, не связанные между собой: взаимодействие с неуправляемой зависимостью и вычисление статистики. Еще раз взгляните на листинг 11.11. Логика предметной области находится в методе `Calculate()`. `GetDeliveries()` только собирает входные данные для этой логики. Вместо того чтобы мокировать `StatisticsCalculator`, разбейте класс на два, как показано в листинге 11.14.

Листинг 11.14. Разбиение `StatisticsCalculator` на два класса

```
public class DeliveryGateway : IDeliveryGateway
{
    public List<DeliveryRecord> GetDeliveries(int customerId)
    {
        /* Обращение к внепроцессной зависимости
        для получения списка товаров */
    }
}

public class StatisticsCalculator
{
    public (double totalWeight, double totalCost) Calculate(
        List<DeliveryRecord> records)
    {
        double totalWeight = records.Sum(x => x.Weight);
        double totalCost = records.Sum(x => x.Cost);

        return (totalWeight, totalCost);
    }
}
```

В листинге 11.15 показан контроллер после рефакторинга.

Листинг 11.15. Контроллер после рефакторинга

```
public class CustomerController
{
    private readonly StatisticsCalculator _calculator;
    private readonly IDeliveryGateway _gateway;

    public CustomerController(
        StatisticsCalculator calculator, | Две разных
        IDeliveryGateway gateway) | зависимости
    {
        _calculator = calculator;
        _gateway = gateway;
    }

    public string GetStatistics(int customerId)
    {
        var records = _gateway.GetDeliveries(customerId);
        (double totalWeight, double totalCost) = _calculator
            .Calculate(records);
```

```

        return
            $"Total weight delivered: {totalWeight}. " +
            $"Total cost: {totalCost}";
    }
}

```

Обязанность по взаимодействию с неуправляемой зависимостью перешла к *DeliveryGateway*. Обратите внимание: этот шлюз теперь работает на базе интерфейса, который может использоваться для мокирования вместо конкретного класса. Код в листинге 11.15 демонстрирует применение паттерна «Простой объект» (*Humble Object*) в действии. За дополнительной информацией об этом паттерне обращайтесь к главе 7.

11.6. Работа со временем

Многие функциональные аспекты приложения требуют обращения к текущей дате и времени. Тестирование функциональности, зависящей от времени, может привести к ложным срабатываниям: время в фазе действия может не совпасть со временем в фазе проверки. Эту зависимость можно стабилизировать тремя способами. Один из этих трех способов является антипаттерном; из двух других один предпочтительнее другого.

11.6.1. Время как неявный контекст

Первый вариант — использование паттерна «Неявный контекст» (*Ambient context*). Этот паттерн уже был описан в главе 8 — в разделе, посвященном тестированию логирования. В контексте времени неявным контекстом должен быть специальный класс, который вы используете в коде вместо встроенного в фреймворк свойства *DateTime.Now* (листинг 11.16).

Листинг 11.16. Текущая дата и время как неявный контекст

```

public static class DateTimeServer
{
    private static Func<DateTime> _func;
    public static DateTime Now => _func();

    public static void Init(Func<DateTime> func)
    {
        _func = func;
    }
}

DateTimeServer.Init(() => DateTime.Now); ← Код инициализации для рабочего кода
DateTimeServer.Init(() => new DateTime(2020, 1, 1)); ← Код инициализации для юнит-тестов

```

Как и в случае с функциональностью логирования, использование неявного контекста для времени также является антипаттерном. Неявный контекст загрязняет рабочий код и усложняет тестирование. Кроме того, статическое поле вводит зависимость, общую для тестов, в результате чего эти тесты переходят в сферу интеграционного тестирования.

11.6.2. Время как явная зависимость

Правильнее будет явно внедрить зависимость времени (вместо того чтобы обращаться к ней через статический метод в неявном контексте) — либо в виде сервиса, либо в виде значения, как показано в листинге 11.17.

Листинг 11.17. Текущая дата и время как явная зависимость

```
public interface IDateTimeServer
{
    DateTime Now { get; }
}

public class DateTimeServer : IDateTimeServer
{
    public DateTime Now => DateTime.Now;
}

public class InquiryController
{
    private readonly IDateTimeServer _dateTimeServer;

    public InquiryController(
        IDateTimeServer dateTimeServer) ← Внедряет время как сервис
    {
        _dateTimeServer = dateTimeServer;
    }

    public void ApproveInquiry(int id)
    {
        Inquiry inquiry = GetById(id);
        inquiry.Approve(_dateTimeServer.Now); ← Внедряет время как значение
        SaveInquiry(inquiry);
    }
}
```

Из этих двух вариантов следует отдать предпочтение внедрению времени в виде значения (вместо сервиса). С простыми значениями проще иметь дело в рабочем коде, к тому же их проще заменять стабами в тестах.

Скорее всего, вам не всегда удастся внедрять время в виде значения, потому что фреймворки внедрения зависимостей обычно плохо сочетаются с объектами-значениями. Хорошим компромиссом служит внедрение времени как сервиса в начале

бизнес-операции и последующая передача его в виде значения в оставшейся части операции. Этот метод продемонстрирован в листинге 11.17; контроллер получает `IDateTimeServer` (сервис), но затем передает значение `DateTime` классу предметной области `Inquiry`.

11.7. Заключение

В этой главе мы рассмотрели некоторые популярные сценарии юнит-тестирования и проанализировали их с использованием четырех атрибутов хорошего теста. Попытки применить сразу все идеи и рекомендации из этой книги могут оказаться непосильными. Кроме того, ваша ситуация может быть не столь очевидной, как те, что я приводил в примерах. Я публикую отзывы о коде других разработчиков и отвечаю на вопросы, относящиеся к юнит-тестированию и проектированию кода в целом, в своем блоге по адресу <https://enterprisecraftsmanship.com>. Также вы можете задать мне свой вопрос по электронной почте: vlad@enterprisecraftsmanship.com. У меня также есть учебный курс, в котором я описываю процесс построения приложения с применением всех принципов, описанных в книге (<https://unittestingcourse.com>).

Вы всегда можете связаться со мной в твиттере (@vkhorikov) или же обратиться напрямую на странице <https://enterprisecraftsmanship.com/about>.

Итоги

- Раскрытие приватных методов ради юнит-тестирования приводит к привязке тестов к деталям имплементации и в конечном итоге снижает устойчивость тестов к рефакторингу. Вместо того чтобы тестировать приватные методы напрямую, тестируйте их косвенно, как часть наблюдаемого поведения.
- Если приватный метод слишком сложен для того, чтобы тестируться как часть открытого API, в котором он используется, это указывает на недостающую абстракцию. Выделите эту абстракцию в отдельный класс, вместо того чтобы делать приватный метод публичным.
- В редких случаях приватные методы принадлежат наблюдаемому поведению класса. Такие методы обычно являются контрактом между классом и ORM-библиотекой или фабрикой.
- Не раскрывайте состояние, которое вы бы без этого предпочли оставить приватным, только ради юнит-тестирования. Ваши тесты должны взаимодействовать с тестируемой системой в точности так же, как и рабочий код; они не должны иметь особых привилегий.
- Не ориентируйтесь ни на какую конкретную реализацию при написании тестов. Проверяйте рабочий код с точки зрения «черного ящика»; избегайте утечек

знаний предметной области в тесты (за дополнительной информацией о тестировании методами «черного ящика» и «белого ящика» обращайтесь к главе 4).

- Загрязнение кода — добавление рабочего кода, который необходим только для тестирования. Это антипаттерн, потому что код тестов смешивается с рабочим кодом, что повышает затраты на сопровождение последнего.
- Необходимость в мокировании конкретного класса для сохранения части его функциональности является результатом нарушения принципа единственной ответственности. Разделите этот класс на два: один содержит логику предметной области, а другой — взаимодействия с внепроцессной зависимостью.
- Использование времени в виде неявного контекста загрязняет рабочий код и усложняет тестирование. Внедряйте время в виде явной зависимости — либо сервиса, либо простого значения. Там, где это возможно, отдавайте предпочтение простым значениям.

Владимир Хориков

Принципы юнит-тестирования

Перевел с английского Е. Матвеев

Заведующая редакцией	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>С. Давид</i>
Ведущий редактор	<i>К. Тульцева</i>
Литературный редактор	<i>А. Руденко</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>М. Молчанова, Г. Шкатова</i>
Верстка	<i>Е. Неволайнен</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: 11.2020. 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 20.11.20. Формат 70x100/16. Бумага офсетная. Усл. п. л. 25,800.
Тираж 700. Заказ

Экстремальное программирование: разработка через тестирование

К. Бек



КУПИТЬ

Возвращение знаменитого бестселлера. Изящный, гибкий и понятный код, который легко модифицировать, который корректно работает и который не подкидывает своим создателям неприятных сюрпризов. Неужели подобное возможно? Чтобы достичь цели, попробуйте тестировать программу еще до того, как она написана. Именно такая парадоксальная идея положена в основу методики TDD (Test-Driven-Development – разработка, основанная на тестировании). Бессмыслица? Не спешите делать скороспелые выводы. Рассматривая применение TDD на примере разработки реального программного кода, автор демонстрирует простоту и мощь этой методики. В книге приведены два программных проекта, целиком и полностью реализованных с использованием TDD. За рассмотрением примеров следует обширный каталог приемов работы в стиле TDD, а также паттернов и рефакторингов, имеющих отношение к TDD. Книга будет полезна для любого программиста, желающего повысить производительность своей работы и получить удовольствие от программирования.

Ловушка для багов. Полевое руководство по веб-хакингу

Питер Яворски



КУПИТЬ

«Ловушка для багов» познакомит вас с белым хакингом – поиском уязвимостей в системе безопасности. Неважно, являетесь ли вы новичком в области кибербезопасности, который хочет сделать интернет безопаснее, или опытным разработчиком, который хочет писать безопасный код, Питер Яворски покажет вам, как это делается. В книге рассматриваются распространенные типы ошибок и реальные хакерские отчеты о таких компаниях, как Twitter, Facebook, Google, Uber и Starbucks. Из этих отчетов вы поймете принципы работы уязвимостей и сможете сделать безопасней собственные приложения. Вы узнаете: как работает интернет, и изучите основные концепции веб-хакинга; как злоумышленники взламывают веб-сайты; как подделка запросов заставляет пользователей отправлять информацию на другие веб-сайты; как получить доступ к данным другого пользователя; с чего начать охоту за уязвимостями; как заставить веб-сайты раскрыть информацию с помощью фейковых запросов.

Мифический человеко-месяц, или Как создаются программные системы

Фредерик Брукс



КУПИТЬ

Немногие книги по управлению проектами можно назвать столь же значимыми, как «Мифический человеко-месяц». Смешение примеров из реальной разработки ПО, мнений и размышлений создает яркую картину управления сложными проектами. Эти эссе основаны на пятидесятилетнем опыте работы Брукса менеджером проектов в IBM System/360, а затем в OS/360. Первое издание книги вышло 45 лет назад, второе — 25 лет назад. Возникают новые методологии, появляются новые языки программирования, растет количество процессоров, но эта книга продолжает оставаться актуальной. Почему? Спустя полвека мы продолжаем повторять ошибки, которые описал Брукс. Некоторые темы, поднимаемые в книге, кажутся устаревшими, но это лишь видимость. Фундаментальные проблемы, стоящие за ними, все так же актуальны в наше время. Важно знать свое прошлое, чтобы понимать, куда развивается индустрия разработки программного обеспечения. Поэтому спустя 45 лет мы и читаем Брукса. Многое изменилось в мире, но девять женщин все так же не могут выносить ребенка за один месяц. ;)

Современный язык Java. Лямбда-выражения, потоки и функциональное программирование

Рауль-Габриэль Урма, Марио Фуско, Алан Майкрофт



КУПИТЬ

Преимущество современных приложений — в передовых решениях, включающих микросервисы, реактивные архитектуры и потоковую обработку данных. Лямбда-выражения, потоки данных и долгожданная система модулей платформы Java значительно упрощают их реализацию. Пришло время повысить свою квалификацию и встретить любой вызов во всеоружии! Книга поможет вам овладеть новыми возможностями современных дополнений, таких как API Streams и система модулей платформы Java. Откройте для себя новые подходы к конкурентности и узнайте, как концепции функциональности улучшают работу с кодом. В этой книге: новые возможности Java; потоковые данные и реактивное программирование; система модулей платформы Java.

Конкурентность и параллелизм на платформе .NET. Паттерны эффективного проектирования

Рикардо Террелл



Рикардо Террелл научит вас писать идеальный код, с которым любые приложения будут просто летать. Книга содержит примеры на языках C# и F#, описывает паттерны проектирования конкурентных и параллельных программ как в теории, так и на практике. Вы начнете с теоретических основ параллелизма, после чего перейдете к примерам и проверенным решениям, помогающим создавать и оптимизировать код для современных многопроцессорных систем. В этой книге автор раскрыл важнейшие конкурентные абстракции, реализацию потоковой обработки событий в реальном времени и наилучшие конкурентные паттерны и практики, применимые на любых платформах.

КУПИТЬ

Высокопроизводительный код на платформе .NET. 2-е издание

Бен Уотсон



Хотите выжать из вашего кода на .NET максимум производительности? Эта книга развеивает мифы о CLR, рассказывает, как писать код, который будет просто летать. Воспользуйтесь ценнейшим опытом специалиста, участвовавшего в разработке одной из крупнейших .NET-систем в мире. В этом издании перечислены все достижения и улучшения, внесенные в .NET за последние несколько лет, в нем также значительно расширен охват инструментов, содержатся дополнительные темы и руководства. Вот лишь некоторые из тем, рассматриваемых в книге: различные способы анализа куч и выявления проблем, связанных с памятью; профессиональное использование Visual Studio и других инструментов; дополнительные сведения об эталонном тестировании; новые варианты настройки сборки мусора; приемы предварительной подготовки кода; более подробный анализ LINQ; советы, касающиеся функциональных областей высокого уровня, таких как ASP.NET, ADO.NET и WPF; новый функционал платформы .NET, включая возвращения по ссылке, структурные кортежи и SIMD.

КУПИТЬ