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. Развертывание 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

3 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              ◄── Класс-контейнер для тестов
{
    [Fact]  ◄── Атрибут xUnit, обозначающий тест
    public void Sum_of_two_numbers()      ◄── Название юнит-теста
    {
        // Arrange
        double first = 10;                  | Секция
        double second = 20;                 | подготовки
        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 коллабораторов или методами, которые тестируемая система вызывает у этих коллабораторов.

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 непрограммиста. 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.

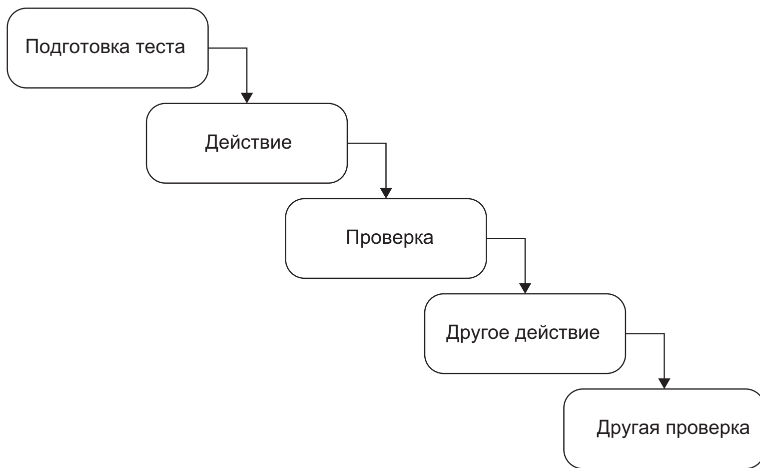

Рис. 3.1. Множественные секции подготовки, действий и проверки указывают на то, что тест пытается проверять слишком много всего. Проблема решается разбиением такого теста на несколько тестов

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 устранить 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 тесте 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 тесте. 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 тесте. 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;                      | Подготовка (arrange)
        var sut = new Calculator();              |

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

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

    public void Dispose()       | Вызывается
    {                           | после каждого
        _sut.CleanUp();         | 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;          <-- Общие тестовые данные
    private readonly Customer _sut;

    public CustomerTests()
    {
        _store = new Store();
        _store.AddInventory(Product.Shampoo, 10);      Выполняется
        _sut = new Customer();                          перед каждым
    }                                                   тестом в классе
```