



Принципы ЮНИТ- тестирования

Владимир Хориков

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. Именованное 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 скорости 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 логов?	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

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 не справляются 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.

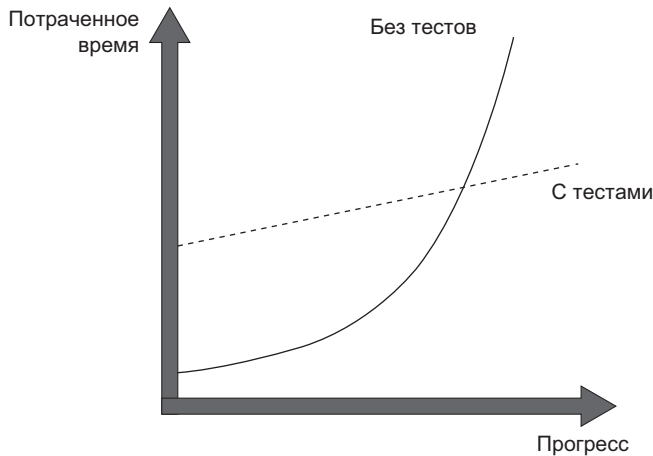


Рис. 1.1. Различия в динамике роста между проектами с тестами и без. Проект без тестов быстро стартует, но и быстро замедляется до состояния, в котором становится трудно двигаться вперед

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.

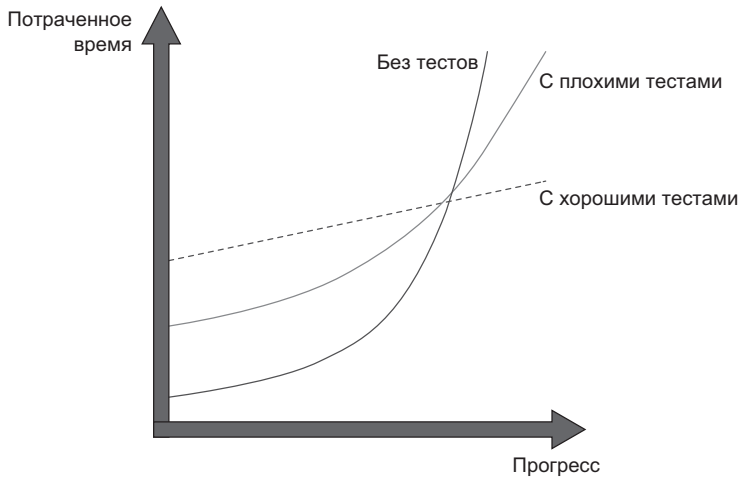


Рис. 1.2. Различия в динамике роста между проектами с плохими и хорошими тестами. Проект с плохо написанными тестами в начальной стадии проявляет свойства проекта с хорошими тестами, но со временем все равно попадает в фазу стагнации

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);
}
```

Покрытие в этом примере вычисляется легко. Общее количество строк в методе равно 5 (фигурные скобки тоже считаются). Количество строк, выполняемых в тесте, 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 пройдены тестами. В `IsStringLong` таких путей два, а тест обрабатывает только один из них.

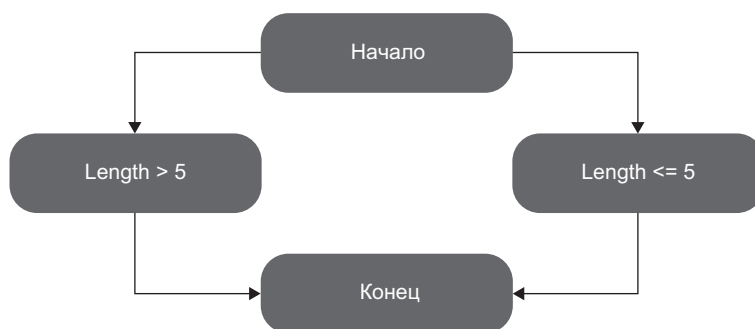


Рис. 1.5. Метод `IsStringLong` представлен в виде графа возможных путей выполнения кода. Тест покрывает только один из двух путей, обеспечивая таким образом 50%-ное покрытие

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;
    WasLastStringLong = result;
    return result;
}
  
```

Первый результат

Второй результат