Принципы юнит-тестирования

Владимир Хориков

MANNING

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

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

Оглавление

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 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.

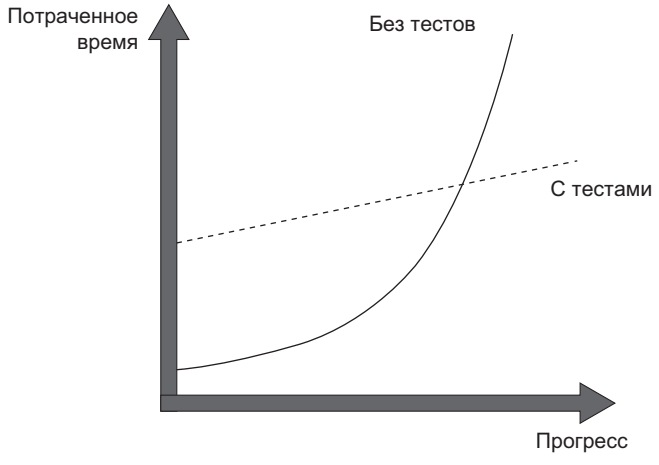


Рис. 1.1. Различия в динамике роста между проектами с тестами и без. Проект без тестов быстро стартует, но и быстро замедляется до состояния, в котором становится трудно двигаться вперед

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.

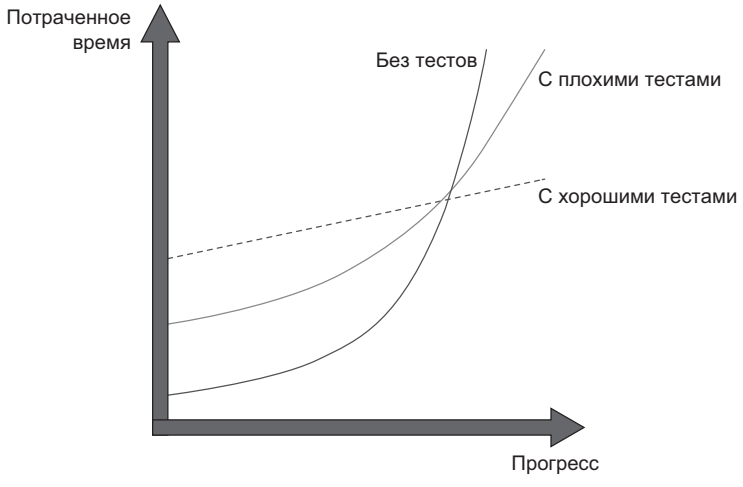


Рис. 1.2. Различия в динамике роста между проектами с плохими и хорошими тестами. Проект с плохо написанными тестами в начальной стадии проявляет свойства проекта с хорошими тестами, но со временем все равно попадает в фазу стагнации

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

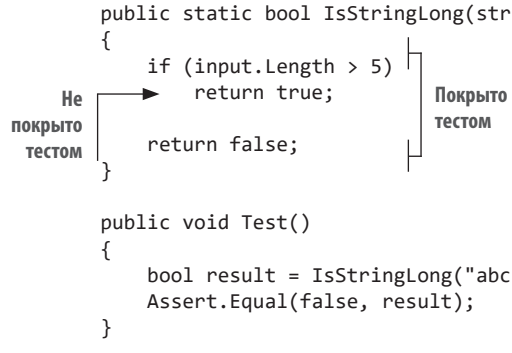
Первая и наиболее часто используемая метрика покрытия — code coverage, также известная как test coverage (рис. 1.3). Эта метрика равна отношению количества строк кода, выполняемых по крайней мере одним тестом, к общему количеству строк в основном коде проекта.

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

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



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).

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

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 отрабатывает только один из них.

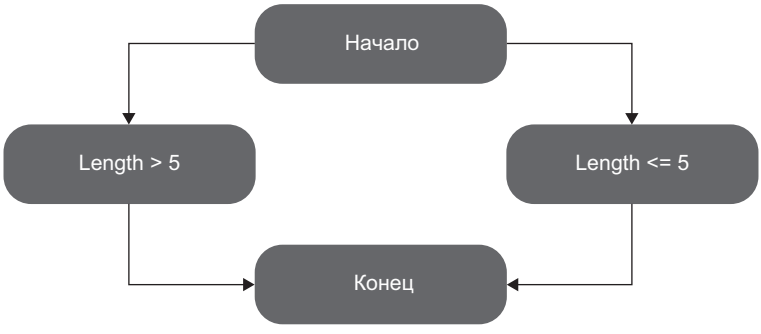


Рис. 1.5. Метод IsStringLong представлен в виде графа возможных путей выполнения кода. Тест покрывает только один из двух путей, обеспечивая таким образом 50%-ное покрытие

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
}
```