Модульное тестирование

Юрий Литвинов
y.litvinov@spbu.ru

1. Введение

Некоторые посылают на проверку задачи, даже не убедившись хоть как-то, что они работают, и я правда слышал от студентов фразу «она правильно написана, только чуть-чуть не компилится». Так, естественно, делать нельзя. Представьте себе, что вы пишете программное обеспечение, управляющее полётом ядерной ракеты. Понадеяться на то, что всё правильно написано с первого раза — весьма немудро. Есть известные шуточные аксиомы, относящиеся к качеству программ:

1. любая программа содержит ошибки;
2. если программа не содержит ошибок, их содержит алгоритм, который реализует эта программа;
3. если ни программа, ни алгоритм ошибок не содержат, такая программа даром никому не нужна.

В общем-то, практика показывает, что так оно и есть. Hello world можно написать и без ошибок, однако практическая ценность такой программы сомнительна. А любая достаточно сложная программная система ошибки (баги) в том или ином виде содержит. Надеяться на то, что вы умнее всех, и если будете кодить внимательно и осторожно, сумеете избежать ошибок — глупо. Даже очень опытные программисты, если просто напишут пару сотен строк кода, они, скорее всего, не будут даже компилиться, не то что уж работать правильно. Помочь обнаружить ошибки и призвано тестирование. На самом деле, первым барьером на пути ошибок является компилятор — если в программе написана явная чушь, компилятор это заметит и поругается. Разработчики языков программирования стараются сделать так, чтобы возможно большее количество ошибок обнаруживалось компилятором (или какими-то другими средствами статического анализа), потому как ошибка, дожившая до времени выполнения, может долго не проявляться. Поэтому, собственно, современные языки так сложны — они намеренно ограничивают возможности программиста, чтобы не дать тому допускать ошибки, которых можно было бы избежать.

Следующий этап жизненного цикла программы после разработки — тестирование. Тестирование, и это важно, не может доказать отсутствие ошибок в программе, оно может лишь помочь обнаружить ошибки, которые в программе присутствуют. Где-то в 70-е годы проводились исследования в области автоматического доказательства корректности программ, там обычно для каждой функции выписывались предусловия (то, что должно быть на входе), постусловия (то, что должно быть после того, как функция отработает), потом

автоматически строилось доказательство того, что из выполнения предусловий следует выполнение постусловий. В итоге оказалось, что выписывание предусловий и постусловий для сколько-нибудь сложной программы занимает больше времени, чем написание самой программы, к тому же в самих пред- и постусловиях могут быть ошибки. Да и автоматически доказательство строится не быстро, так что сейчас такие вещи особо не используются. Есть, правда, программирование по контракту, один из основополагающих принципов в языке Eiffel, там предусловия и постусловия выписываются явно, но проверяются во время выполнения (так что это не то). В принципе, контракты на уровне библиотеки поддерживаются много где: Microsoft CodeContracts[1], контракты в IntelliJ IDEA[2]; в Ada контракты поддерживаются на уровне языка, как в Eiffel.

Теперь, собственно, про тестирование. Тестирование бывает модульное — когда тестируются отдельные модули (функции, методы, классы, и т.д.), интеграционное (когда тестируется взаимодействие набора модулей системы), и системное (когда тестируется вся система целиком, в рабочих условиях). Интеграционное тестирование и системное тестирование обычно выполняются специально обученными людьми, а вот модульное тестирование — самими программистами. Модульное тестирование заключается в том, что для каждой нетривиальной функции или метода пишутся свои тесты, которые проверяют, что метод работает как надо. Предполагается, что модульные тесты будут запускаться после каждой сборки программы, поэтому они должны работать по возможности быстро. Принято не продолжать разработку, если есть хотя бы один не прошедший модульный тест.

Кроме того, тесты часто классифицируют по цели тестирования. Самые первые тесты, которые запускают тестировщики, называют «дымовыми» (smoke-test), они служат для проверки хотя бы базовой работоспособности программы (по аналогии с тестированием электроприбора, его втыкают в розетку и смотрят, не пошёл ли дым). Если дымовые тесты пройдены, программа начинает тестироваться более обстоятельно, обычно дымовые тесты длятся не больше нескольких минут. Ещё бывают приёмочные тесты — они, как правило, проводятся заказчиком или в присутствии заказчика при сдаче системы, и тестируется вся система в рабочих условиях. Они тоже, как правило, довольно короткие. Регрессионные тесты направлены на то, чтобы найти ошибки, привнесённые фиксами или изменениями в других частях программы (регресс). Часто любой тест, как только он начал проходить, добавляют к базе регрессионных тестов и запускают каждый раз, когда в системе что-нибудь поменялось. Если он не прошёл — мы нашли регресс. Естественно, регрессионные тесты, как правило, автоматические. Ещё бывают нагрузочные тесты, стресс-тесты, юзабилити-тесты и т.д. и т.п., тестирование всё-таки целое отдельное большое направление программной инженерии, а тестировщик — отдельная профессия.

2. Пример

Небольшой пример суперпростой программы и образа мыслей тестировщика, чтобы показать, на что вообще надо обращать внимание при тестировании и откуда, собственно, брать тесты. Пример приводится по книжке C. Kaner, «Testing Computer Software», которая очень старая, но всё ещё основная книжка для тестировщиков, рекомендую прочитать хотя бы первую главу, чтобы проникнуться духом. Задача такая — написать про-

1 https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts

2 https://www.jetbrains.com/help/idea/contract-annotations.html.

грамму, складывающую два двузначных числа. Называться она должна adder, при запуске она должна ждать ввода первого числа (который заканчивается Enter), потом второго, потом печатать на экране ответ. Проще только Hello, world.

Сначала смоук-тестирование, на работоспособность программы в самом частом сценарии её использования:

Что делаем	**Что происходит**
Вводим *adder* и жмём на *Enter*	Экран мигает, внизу появляется знак вопроса
Нажимаем 2	За знаком вопроса появляется цифра 2
Нажимаем *Enter*	В следующей строке появляется знак вопроса
Нажимаем 3	За вторым знаком вопроса появляется цифра 3
Нажимаем *Enter*	В третьей строке появляется 5, несколькими строками ниже — ещё один знак вопроса

Обратите внимание, сценарий тестирования (test case) составлен так, что в нём чётко прописано каждое действие тестировщика и отдельно чётко прописан ожидаемый результат (а в случае реального прогона теста эта графа содержит описание реального результата). Всё просто и понятно, вряд ли разработчик мог ошибиться в такой программе, однако хороший тестировщик после смоук-теста уже выявил бы три проблемы.

- Нет названия программы на экране, может, мы запустили не то.
- Нет никаких инструкций, пользователь без идей, что делать. Программа просто показывает чёрный экран и мигает курсором.
- Непонятно, как выйти.

Вы скажете, что в задании не было ничего про название, подсказки и т.д., так что программа полностью удовлетворяет спецификации и работает правильно. Но нет. Дело в том, что пользователям плевать на спецификации, которым соответствует программа, они хотят, чтобы ей было удобно пользоваться и чтобы она решала их проблемы. Так что ошибка — это чисто субъективное понятие, имеющее отношение к взаимодействию некоторого пользователя и программы, а не к отношению программы и условия, спецификации, ТЗ и т.д. Кстати, пользователи бывают разные, иногда их желания противоречат друг другу, поэтому любая программа содержит ошибки. Задача тестировщика — поставить себя на место пользователя и понять, что пользователю может не понравиться в программе, а не «правильно она работает или нет». Абсолютно правильно работающие программы могут быть абсолютно никому не нужны, а неправильно работающие программы наоборот, быть очень популярны. В игровой индустрии, например, есть масса тому примеров.

Окей, допустим, что смоук-тест пройден, программа суперпростая, поэтому можно закончить тестирование и сказать, что всё ок? Нет конечно, теперь надо проверить поведение программы на всех допустимых входных данных. На совсем всех — плохая идея, потому что всех возможных комбинаций двузначных чисел многовато, выберем из них самые интересные:

Ввод	**Ожидаемый результат**	**Замечания**
99 + 99	198	Пара наибольших допустимых чисел
-99 + -99	-198	Отрицательные числа, почему нет?
99 + -14	85	Большое первое число может влиять на интерпретацию второго
-38 + 99	61	Отрицательное плюс положительное
56 + 99	155	Большое второе число может повлиять на интерпретацию первого
9 + 9	18	Два наибольших числа из одной цифры
0 + 0	0	Программы часто не работают на нулях
0 + 23	23	0 — подозрительная штука, его надо проверить и как первое слагаемое,
-78 + 0	-78	и как второе

Теперь всё? Нет конечно, ведь если бы все всегда вводили правильные данные, компьютеры были бы не нужны. Любая программа должна ожидать от пользователя, что он будет ошибаться, и не должна издеваться над ним за его неаккуратность, а корректно работать, при этом помогая пользователю исправить ошибку. Поэтому тестируем ещё и некорректный ввод и проверяем, что не произошло ничего плохого:

Ввод	**Замечания**
100 + 100	Поведение сразу за диапазоном допустимых значений
Enter + *Enter*	Что будет, если данные не вводить вообще
123456 + 0	Введём побольше цифр
1.2 + 5	Вещественные числа, пользователь может решить, что так можно
A + b	Недопустимые символы, что будет?
Ctrl-A, Ctrl-D, F1, Esc	Управляющие клавиши часто источник проблем в консольных программах

Но и это ещё не всё. Зная, как примерно могла бы быть реализована такая программа, мы можем целенаправленно тестировать случаи, опасные для конкретной реализации.

- Внутреннее хранение данных — двузначные числа есть большой соблазн хранить в **byte**, от -128 до 127, но тогда результат сложения может вызывать переполнение, надо это проверить.
  - 99 + 99, этот случай покрыли.
- Кодовая страница ввода: символы '/' и '0', '9' и ':' находятся там рядом, поэтому программист может напутать со строгостью неравенства при проверке того, цифра введённый символ или нет (метод int.TryParse сам разберётся, но а вдруг мы руками парсим число).
  - Можно упростить тесты: не надо вводить A + b, достаточно граничные символы.

Вот некоторая статистика из книжки Канера, которая была собрана в 80-х, но с некоторыми корректировками актуальна и по сей день (более свежие отчёты её подтверждают).

- Программа из сотни строк может иметь 10^{18} путей исполнения, в книжке Канера приводится ссылка на программу, специально созданную, чтобы продемонстрировать такую ситуацию.
  - Времени жизни вселенной не хватило бы, чтобы их покрыть, так что писать тесты, покрывающие всё — бесполезно.
- После передачи на тестирование в программах в среднем от 1 до 3 ошибок на 100 строк кода.
- В процессе разработки — 1.5 ошибок на 1 строку кода (!). Это до запуска компилятора и даже до исправления опечаток, большая часть этих ошибок мгновенно исправляется.
- Если для исправления ошибки надо изменить не более 10 операторов, с первого раза это делают правильно в 50% случаев.
- Если для исправления ошибки надо изменить не более 50 операторов, с первого раза это делают правильно в 20% случаев. Так что каждый фикс необходимо перепроверить, и иметь набор регрессионных тестов, проверяющих, что ничего больше не отвалилось.

3. Модульные тесты

Собственно, модульные тесты — это то, что пишет сам программист. Модульные тесты проверяют код, причём маленькие его кусочки — классы, методы и т.д., в отрыве от остальной системы. Тест занимается тем, что ищет ошибки в конкретных строках кода, а не проверяет абстрактную работоспособность абстрактной подсистемы. Модульные тесты, тем не менее, не должны напрямую проверять внутреннее состояние и поведение тестируемого класса, а проверять только его обозреваемое поведение (то есть public-методы). Почему — это позволит менять реализацию класса, не переписывая каждый раз тесты к нему.

Модульные тесты должны быть полностью автоматическими — они запускаются часто без участия программиста и должны просто говорить «да» или «нет». Как правило, тесты живут в отдельном проекте, и тестируемая система должна прекрасно работать и собираться даже если тестов нет. Вставлять в программу какой-то код, который помогает её тестировать, а тем более, включать в код сами тесты — дурной тон, пользователю юнит-тесты в собранной программе поставляться не должны (они ему не нужны, зачем ему скачивать ещё сколько-то мегабайт бинарного кода?).

Модульные тесты помогают найти кучу ошибок на ранних этапах. Если тесты запускаются достаточно часто, вы будете помнить, что вы такое поменяли, после чего тесты перестали проходить. К тому же тест — это часто единственная возможность запустить код, который вы только что написали, когда не готова ещё вся остальная система, которая по идее должна его вызывать.

Кроме того, тесты облегчают изменение программы. Вы что-то поменяли, что не должно отразиться на наблюдаемом поведении вашего класса или метода, запускаете тесты и

смотрите, что они действительно все прошли. Кстати, изменения, направленные на улучшение сопровождаемости без изменений в поведении программы с точки зрения пользователя, называются рефакторингом, вот юнит-тесты для рефакторинга чрезвычайно полезны. Они дают некую уверенность, что всё хорошо. Но помните, что тесты не доказывают отсутствие ошибок в программе.

Тесты можно рассматривать как документацию к коду — любой, кто захочет воспользоваться вашим классом, может посмотреть в ваши юнит-тесты к этому классу и понять, как его использовать.

Тесты помогают улучшить структуру программы — дикую мешанину невозможно толком оттестировать, так что волей-неволей придётся задуматься об аккуратной архитектуре.

4. Модульные тесты на C#

Рассмотрим в качестве примера модульное тестирование в языке C# в IDE Visual Studio. В Rider всё работает примерно также, хоть и выглядит немного по-другому, да и C# достаточно идеологически похож на Java/Kotlin и даже C++, чтобы можно было без проблем перенести знания. Тем более что с точки зрения юнит-тестов в .NET и в JVM-мире самая популярная библиотека юнит-тестирования — jUnit/NUnit, так что, хоть NUnit вроде как успели полностью переписать, даже методы будут называться одинаково в большинстве случаев. Помимо NUnit, есть ещё Microsoft Unit Test Framework и xUnit, и разные другие библиотеки, но NUnit всё-таки доминирует. jUnit — библиотека для юнит-тестирования Java-приложений, разработанная Эрихом Гамма и Кентом Беком (членами «банды чётырёх», товарищей, активно пропагандировавших объектно-ориентированное программирование в 90-х и, фактически, стоявших у истоков большинства современных практик разработки). Идея, положенная в её основу (и всех её аналогов), очень проста — каждому классу соответствует класс с таким же именем + Test на конце, который содержит тесты для всех методов класса. Есть программа, которая умеет последовательно запускать все тесты в тестовом классе и как-то обрабатывать результат. В случае Microsoft Unit Test Framework это всё встроено прямо в Visual Studio, в NUnit это правда отдельная программа, хотя он тоже интегрируется в IDEшки, конечно. Принцип, в обоих вариантах, такой — если все тесты прошли, загорается зелёная полоска, если хоть один не прошёл — красная, и пишется информация, какой тест не прошёл и почему. Сами тесты — это просто методы тестового класса, помеченные некоторым специальным атрибутом, имя их обычно совпадает с именем тестируемого метода, но начинается с Test, а внутри что-то делается (готовятся данные), вызывается метод, который хотим протестировать, и смотрим на результат — вызывая специальные методы типа *AssertEquals*, *Fail* и т.д. В общем, идея простая, так что помимо NUnit и его прародителя jUnit, бывают аналоги для Delphi, для C++ (google test framework, cppunit, например, и даже QTest в Qt, для C++ таких систем много). Несложная система модульного тестирования пишется за выходные, а особо сложных и не бывает.

5. Демонстрация

Будем для демонстрации использовать Microsoft Unit Test Framework и среду Visual Studio, для других сред детали будут несколько отличаться, но всё на самом деле очень похоже.

Пишем в Visual Studio какой-нибудь класс, типа стека, как-то так (детали синтаксиса C# тут не критичны, но вроде как интуитивно всё должно быть понятно — если нет, не стесняйтесь спрашивать):

```
namespace Stack;

public class Stack
{
    private class StackElement
    {
        /// Свойства, поле с геттером и сеттером.
        public int Value { get; set; }
        public StackElement Next { get; set; }
    }

    public void Push(int value)
    {
        // Инициализатор, позволяет сразу инициализировать свойства
        var newElement = new StackElement()
        {
            Next = head,
            Value = value
        };

        head = newElement;
    }

    public int Pop()
    {
        var temp = head.Value;
        head = head.Next;
        return temp;
    }

    public bool IsEmpty() => head == null;

    private StackElement head = null;
}
```

Тыкаем правой кнопкой на солюшн, говорим Create New Project, выбираем C# -> Test -> Unit Test Project. Называем его так же, как и исходный проект, только в конце .Tests (это

не принципиально, но является хорошей практикой). Например, если у вас был проект *Stack*, то добавляете новый проект *Stack.Tests*.

Там уже сгенерился такой примерно кусок кода:

```
[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
    }
}
```

Что тут написано: есть тестовый класс (помеченный атрибутом *[TestClass]*), он состоит из одного пока тестового метода (помеченного атрибутом *[TestMethod]*). Атрибут — это такая штука (на самом деле, объект некоторого класса) которая хранится в дереве разбора программы вместе с аннотированным узлом и может быть использована самим компилятором или разными инструментами, например, системой юнит-тестирования. Собственно, всё интуитивно понятно (надеюсь). Это можно запустить (Test -> Run -> All tests), оно скажет, что тест пройден (ну, неудивительно). Теперь — что с этим делать дальше.

Новому проекту добавляем ссылку на исходный проект (правой кнопкой по References, там Add, там Solution, там ставим галочку напротив нашего проекта, жмём ок — references определяют, какой проект знает о каком и, следовательно, может использовать классы из другого проекта). Переименовываем класс с убогим названием *UnitTest1* в *StackTest* (выделяем файл в Solution Explorer-е, жмём F2, вводим новое имя, соглашаемся на предложение переименовать содержимое). Метод *TestMethod1* переименовываем в PushTest, пишем там такое:

```
[TestMethod]
public void PushTest()
{
    var stack = new Stack();
    Assert.IsTrue(stack.IsEmpty());
    stack.Push(1);
    Assert.IsFalse(stack.IsEmpty());
}
```

Запускаем, тесты всё ещё проходят. Грустно, значит, мы впустую потратили время (помним, тесты не доказывают отсутствие ошибок, они помогают обнаружить их наличие). Пишем ещё один тест:

```
[TestMethod]
public void PopTest()
{
    var stack = new Stack();
    stack.Push(1);
    var result = stack.Pop();
```

```
    Assert.AreEqual(1, result);
}
```

Тоже проходит. Но такой тест будет проходить даже если стек будет не стеком, а будет просто помнить последнее положенное в него значение. Так что придётся написать ещё один тест:

```
[TestMethod]
public void PopTestWithTwoElements()
{
    var stack = new Stack();
    stack.Push(1);
    stack.Push(2);
    Assert.AreEqual(2, stack.Pop());
    Assert.AreEqual(1, stack.Pop());
}
```

Теперь попробуем, что будет, если сделать Pop до Push:

```
[TestMethod]
public void PopTestWithNoPush()
{
    var stack = new Stack();
    stack.Pop();
}
```

Тест наконец-то не прошёл, так мы таки нашли баг. Баг легко поправить (особенно, пользуясь Test -> Debug), так что смотрим на тесты ещё раз, и видим мерзостную копипасту: *stack* создаётся заново в каждом тесте. Побороть это можно, дописав в тестовом классе метод

```
[TestInitialize]
public void Initialize()
{
    stack = new Stack();
}
```

и объявив поле *stack*. Метод, помеченный как *[TestInitialize]* вызывается перед каждым тестом, *[TestCleanup]* — после каждого теста, и используется для подготовки тестовых данных/восстановления всего как было.

6. Best practices

Пожалуй, самое важное в юнит-тестах — это общие правила их использования. Их не так много, но их часто по неопытности нарушают. Например, важно, чтобы тесты были независимыми друг от друга, так что если не прошёл один, это не должно приводить к тому, что валится ещё куча остальных (или ещё хуже, когда тест прошёл и модифицировал

данные так, что другой тест тоже прошёл, хотя должен был упасть, и честно падает, если запускать его в одиночку). Тесты вообще должны тестировать что-то конкретное, так, чтобы несработавший тест сразу локализовал бы проблему. А если вы тестируете, скажем, стек, и делаете в одном тесте кучу всяких push и pop — то если тест не пройдёт, что нам это скажет?

Тесты должны работать быстро — если каждый запуск тестов останавливает разработку на полчаса, никто их не будет запускать и пользы от них не будет. Если вы работаете с каким-то куском кода, имейте на него тесты, которые проходят за единицы секунд, и имейте больший набор тестов, запускающийся после каждого коммита в git и работающий не больше, чем за 10 минут (скоро будет про то, как это организовать — можете пока погуглить GitHub Actions).

Тестов должно быть много. Есть такое понятие, как тестовое покрытие — это множество строк, которые исполнялись в ходе прогона всех тестов. Измеряется в процентах от общего количества содержательных строк кода в программе. Оно должно быть большим, где-то 90% — разумный минимум (просто потому что всякие ошибочные ситуации типа «что будет, если выдернуть жёсткий диск прямо в процессе записи») тестировать тяжело. Но и 100% покрытия — это на самом деле мало, потому что на самом деле надо проверять все пути исполнения, а не просто строчки кода, а путей исполнения могут быть миллиарды. По объёму кода тестов обычно оказывается несколько больше, чем кода, который они тестируют, но они существенно проще. И да, тесты тоже надо сопровождать, фиксить в них баги, рефакторить, и менять, если в тестируемом коде что-то поменялось. тем не менее, юнит-тесты используются нынче практически во всех программных проектах, не пользоваться ими ужасно.

Ещё есть такой подход к написанию программ, в котором сначала пишутся тесты к классу, а уж затем — сам класс. Называется он test-driven development — разработка через тестирование. Это даже круче, чем просто модульное тестирование, потому как тест заставляет задумываться о том, что мы такое собираемся писать и как его использовать. К тому же, мы получаем чёткий критерий того, что мы закончили — тест прошёл. Такой подход используется в экстремальном программировании в совокупности с некоторыми другими методиками.