Юнит-тестирование

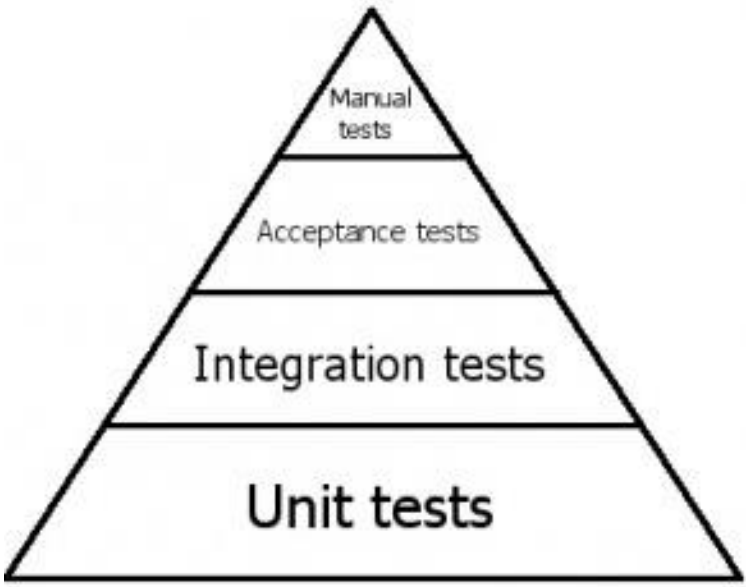
Даже если вы никогда в жизни не думали, что занимаетесь тестированием, вы это делаете. Вы собираете свое приложение, нажимаете кнопку и проверяете, соответствует ли полученный результат вашим ожиданиям

То что вы делаете, называется *интеграционным тестированием*. Современные приложения достаточно сложны и содержат множество зависимостей. Интеграционное тестирование проверяет, что несколько компонентов системы работают вместе правильно.

Оно выполняет свою задачу, но сложно для автоматизации. Как правило, тесты требуют, чтобы вся или почти вся система была развернута и сконфигурирована на машине, на которой они выполняются. Предположим, что вы разрабатываете web-приложение с UI и веб-сервисами. Минимальная комплектация, которая вам потребуется: браузер, веб-сервер, правильно настроенные веб-сервисы и база данных. На практике все еще сложнее. Разворачивать всё это на билд-сервере и всех машинах разработчиков?

We need to go deeper

Давайте сначала спустимся на предыдущий уровень и убедимся, что наши компоненты работают правильно по-отдельности.



Обратимся к википедии: **Модульное тестирование**, или **юнит-тестирование** (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Таким образом, юнит-тестирование – это первый бастион на борьбе с багами. За ним еще интеграционное, приемочное и, наконец, ручное тестирование, в том числе «свободный поиск»

Не нужно писать тесты, если

- Вы делаете простой сайт-визитку из 5 статических html-страниц и с одной формой отправки письма. На этом заказчик, скорее всего, успокоится, ничего большего ему не нужно. Здесь нет никакой особенной логики, быстрее просто все проверить «руками»
- Вы занимаетесь рекламным сайтом/простыми флеш-играми или баннерами – сложная верстка/анимация или большой объем статики. Никакой логики нет, только представление
- Вы делаете проект для выставки. Срок – от двух недель до месяца, ваша система – комбинация железа и софта, в начале проекта не до конца известно, что именно должно получиться в конце. Софт будет работать 1-2 дня на выставке
- Вы всегда пишете код без ошибок, обладаете идеальной памятью и даром предвидения. Ваш код настолько крут, что изменяет себя сам, вслед за требованиями клиента. Иногда код объясняет клиенту, что его требования — ~~гов~~ не нужно реализовывать

В первых трех случаях по объективным причинам (сжатые сроки, бюджеты, размытые цели или очень простые требования) вы не получите выигрыша от написания тестов.

Последний случай рассмотрим отдельно. Я знаю только одного такого человека, и если вы Чак Норрис, то у меня для вас плохие новости

Любой долгосрочный проект без надлежащего покрытия тестами обречен рано или поздно быть переписанным с нуля

В своей практике я много раз встречался с проектами старше года. Они делятся на три категории:

- **Без покрытия тестами.** Обычно такие системы сопровождаются спагетти-кодом и уволившимися ведущими разработчиками. Никто в компании не знает, как именно все это работает. Да и что оно в конечном итоге должно делать, сотрудники представляют весьма отдаленно.
- **С тестами, которые никто не запускает и не поддерживает.** Тесты в системе есть, но что они тестируют, и какой от них ожидается результат, неизвестно. Ситуация уже лучше. Присутствует какая-никакая архитектура, есть

понимание, что такое слабая связанность. Можно отыскать некоторые документы. Скорее всего, в компании еще работает главный разработчик системы, который держит в голове особенности и хитросплетения кода.

- **С серьезным покрытием. Все тесты проходят.** Если тесты в проекте действительно запускаются, то их много. Гораздо больше, чем в системах из предыдущей группы. И теперь каждый из них – атомарный: один тест проверяет только одну вещь. Тест является спецификацией метода класса, контрактом: какие входные параметры ожидает этот метод, и что остальные компоненты системы ждут от него на выходе. Таких систем гораздо меньше. В них присутствует актуальная спецификация. Текста немного: обычно пара страниц, с описанием основных фич, схем серверов и *getting started guide'ом*. В этом случае проект не зависит от людей. Разработчики могут приходить и уходить. Система надежно протестирована и сама рассказывает о себе путем тестов

Проекты первого типа – крепкий орешек, с ними работать тяжелее всего. Обычно их рефакторинг по стоимости равен или превышает переписывание с нуля

Выберите логическое расположение тестов в вашей VCS

Только так. Ваши тесты должны быть частью контроля версий. В зависимости от типа вашего решения, они могут быть организованы по-разному. Общая рекомендация: если приложение монолитное, положите все тесты в папку Tests; если у вас много разных компонентов, храните тесты в папке каждого компонента.

Выберите способ именования проектов с тестами

Одна из лучших практик: добавьте к каждому проекту его собственный тестовый проект.
У вас есть части системы <PROJECT_NAME>.Core, <PROJECT_NAME>.Bl и <PROJECT_NAME>.Web? Добавьте еще <PROJECT_NAME>.Core.Tests, <PROJECT_NAME>.Bl.Tests и <PROJECT_NAME>.Web.Tests.

У такого способа именования есть дополнительный сайд-эффект. Вы сможете использовать паттерн *.Tests.dll для запуска тестов на билд-сервере.

Используйте такой же способ именования для тестовых классов

У вас есть класс ProblemResolver? Добавьте в тестовый проект ProblemResolverTests. Каждый тестирующий класс должен тестировать только одну сущность. Иначе вы очень быстро скатитесь ~~в унылое го~~ во второй тип проектов (с тестами, которые никто не запускает).

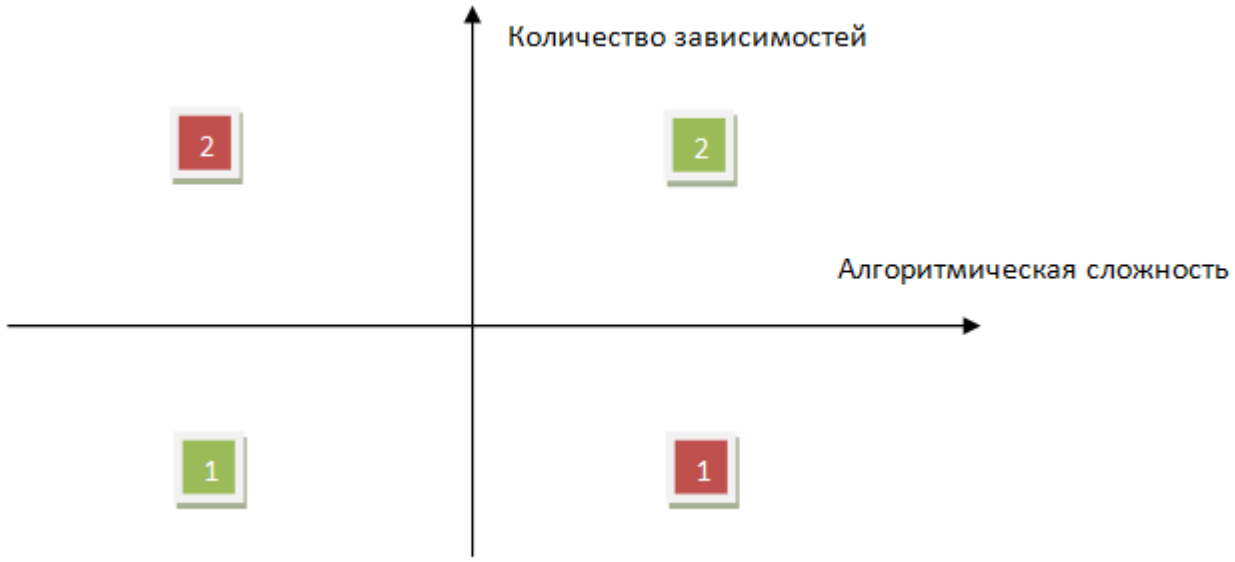
Выберите «говорящий» способ именования методов тестирующих классов

TestLogin – не самое лучшее название метода. Что именно тестируется? Каковы входные параметры? Могут ли возникать ошибки и исключительные ситуации?

На мой взгляд, лучший способ именования методов такой: *[Тестируемый метод]_[Сценарий]_[Ожидаемое поведение]*.

Что тестировать, а что – нет?

Одни говорят о необходимости покрытия кода на 100%, другие считают это лишней тратой ресурсов.
Мне нравится такой подход: расчертите лист бумаги по оси X и Y, где X – алгоритмическая сложность, а Y – количество зависимостей. Ваш код можно разделить на 4 группы.



Рассмотрим сначала экстремальные случаи: простой код без зависимостей и сложный код с большим количеством зависимостей.

1. **Простой код без зависимостей.** Скорее всего здесь и так все ясно. Его можно не тестировать.
2. **Сложный код с большим количеством зависимостей.** Хм, если у вас есть такой код, тут пахнет God Object'ом и сильной связностью. Скорее всего, неплохо будет провести рефакторинг. Мы не станем покрывать этот код юнит-тестами, потому что перепишем его, а значит, у нас изменятся сигнатуры методов и появятся новые классы. Так зачем писать тесты, которые придется выбросить? Хочу оговориться, что для проведения такого рода рефакторинга нам все же нужно тестирование, но лучше воспользоваться более высокоуровневыми *приемочными тестами*. Мы рассмотрим этот случай отдельно.

Что у нас остается:

1. **Сложный код без зависимостей.** Это некие алгоритмы или бизнес-логика. Отлично, это важные части системы, тестируем их.
2. **Не очень сложный код с зависимостями.** Этот код связывает между собой разные компоненты. Тесты важны, чтобы уточнить, как именно должно происходить взаимодействие. Причина потери Mars Climate Orbiter 23 сентября 1999 года заключалась в программно-человеческой ошибке: одно подразделение проекта считало «в дюймах», а другое – «в метрах», и прояснили это уже после потери аппарата. Результат мог быть другим, если бы команды протестировали «швы» приложения.

Придерживайтесь единого стиля написания тела теста

Отлично зарекомендовал себя подход AAA *(arrange, act, assert)* . Вернемся к примеру с калькулятором:

```
class CalculatorTests
{
        public void Sum_2Plus5_7Returned()
        {
                // arrange
                var calc = new Calculator();

                // act
                var res = calc.Sum(2,5);

                // assert
                Assert.AreEqual(7, res);
        }
}
```

Тестируйте одну вещь за один раз

Каждый тест должен проверять только одну вещь. Если процесс слишком сложен (например, покупка в интернет магазине), разделите его на несколько частей и протестируйте их отдельно.
Если вы не будете придерживаться этого правила, ваши тесты станут нечитаемыми, и вскоре вам окажется очень сложно их поддерживать.

Борьба с зависимостями

До сих пор мы тестировали калькулятор. У него совсем нет зависимостей. В современных бизнес-приложениях количество таких классов, к сожалению, не мало.

Fakes: stubs & mocks

Мы переписали класс и теперь можем подсунуть контроллеру другие реализации зависимостей, которые не станут лезть в базу, смотреть конфиги и т.д. Словом, будут делать только то, что от них требуется. Разделяем и властвуем. Настоящие реализации мы должны протестировать отдельно в своих собственных тестовых классах. Сейчас мы тестируем только контроллер.

Выделяют два типа подделок: стабы (stubs) и моки (mock).
Часто эти понятия путают. Разница в том, что стаб ничего не проверяет, а лишь имитирует заданное состояние. А мок – это объект, у которого есть ожидания. Например, что данный метод класса должен быть вызван определенное число раз. Иными словами, ваш тест никогда не сломается из-за «стаба», а вот из-за мока может. С технической точки зрения это значит, что используя стабы в Assert мы проверяем состояние тестируемого класса или результат выполненного метода. При использовании мока мы проверяем, соответствуют ли ожидания мока поведению тестируемого класса.

Тестирование состояния и тестирование поведения

Почему важно понимать, казалось бы, незначительную разницу между моками и стабами? Давайте представим, что нам нужно протестировать автоматическую систему полива. Можно подойти к этой задаче двумя способами:

Тестирование состояния

Запускаем цикл (12 часов). И через 12 часов проверяем, хорошо ли политы растения, достаточно ли воды, каково состояние почвы и т.д.

Тестирование взаимодействия

Установим датчики, которые будут засекать, когда полив начался и закончился, и сколько воды поступило из системы.
Стабы используются при тестировании состояния, а моки – взаимодействия. **Лучше использовать не более одного мока на тест**. Иначе с высокой вероятностью вы нарушите принцип «тестировать только одну вещь». При этом в одном тесте может быть сколько угодно стабов или же мок и стабы.

Принцип TDD:

1. Прочитать задание и написать тест, который заваливается
2. Написать любой код, который позволяет проходить данный тест и остальные тесты
3. Сделать рефакторинг, т.е. убрать повторяющийся код, если надо, но чтобы все тесты проходили

Установить NUnit
Так же в VS устанавливаем NUnit Test Adapter (ну чтобы запускать тесты прямо в VS)

Создадим папочку типа Solution Folder Test и в нее добавим проект