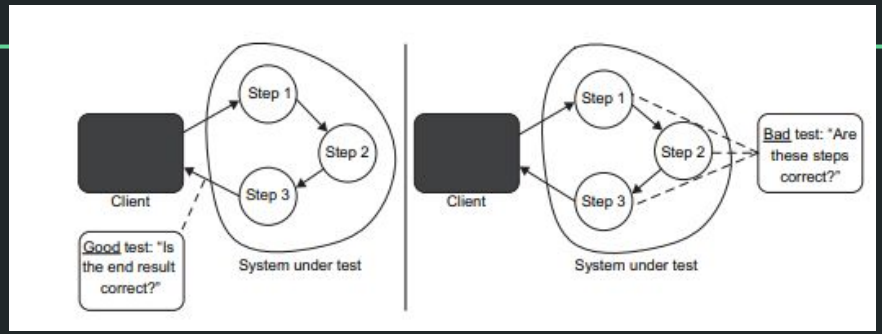
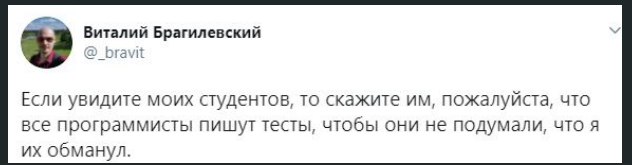
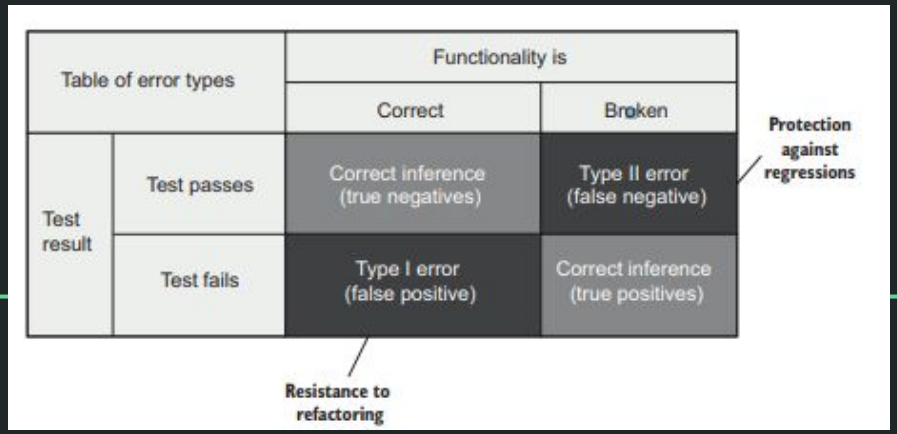
Базовые принципы ведения Unit-тестов в проекте

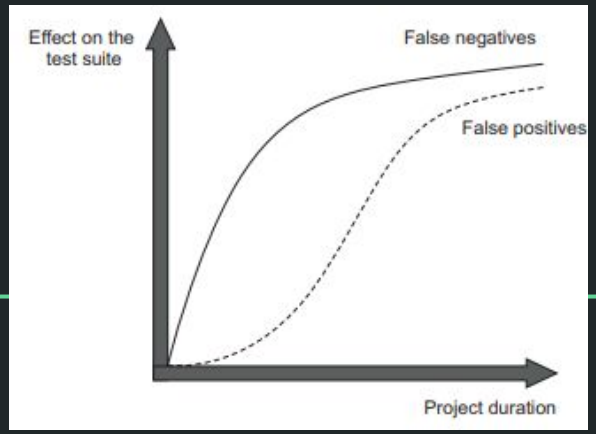
- Тесты должны предоставлять защиту от регрессий
  - Увеличиваем покрытие нашего кода
  - Уменьшаем количество транзитивных зависимостей
  - Тестируем критичный код компонентов
- Тесты должны быть устойчивы к рефакторингу
  - Сама суть рефакторинга подразумевает, что тесты сломаться не должны
  - А если сломались, то стоит задуматься, а стоило ли рефакторить
  - false positive / false negative / ошибки 1 и 2 рода





Базовые принципы ведения Unit-тестов в проекте

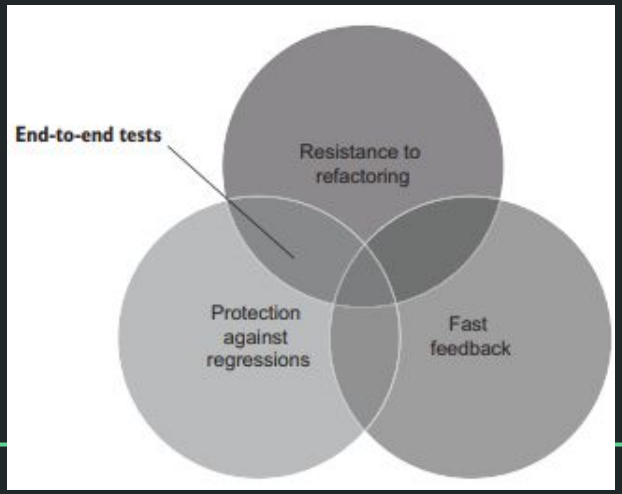


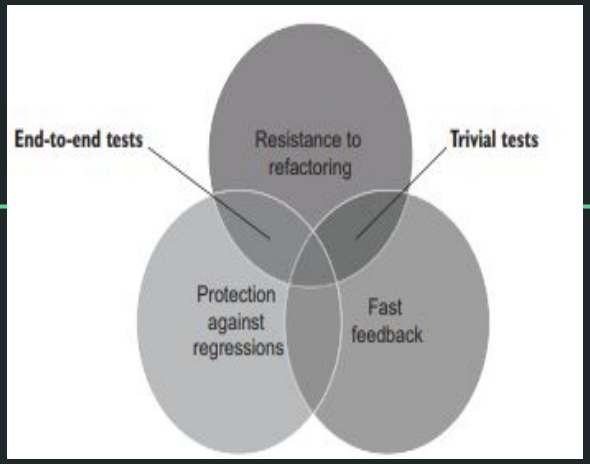


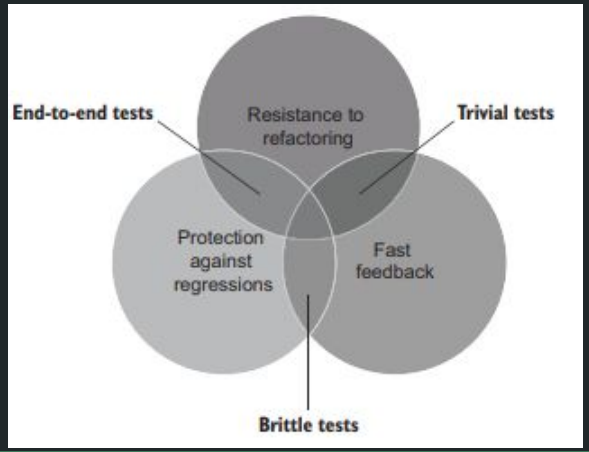
Базовые принципы ведения Unit-тестов в проекте

- Результатов тестов должен быть понятным для быстрой оценки
  - быстро проходят
  - быстро строится отчет
  - быстро можно сделать вывод по этому отчету
  - быстро можно понять -- ошибки 1 и 2 рода или корректные ошибки \ успешные тесты
- Тесты должны быть легко поддерживаемы
  - легко прочитать
  - легко понять
  - легко изменить
  - легко запустить локально
  - легко интегрировать в CI\CD

Базовые принципы ведения Unit-тестов в проекте



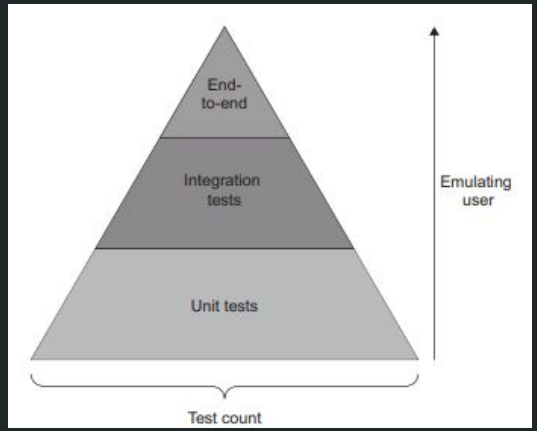




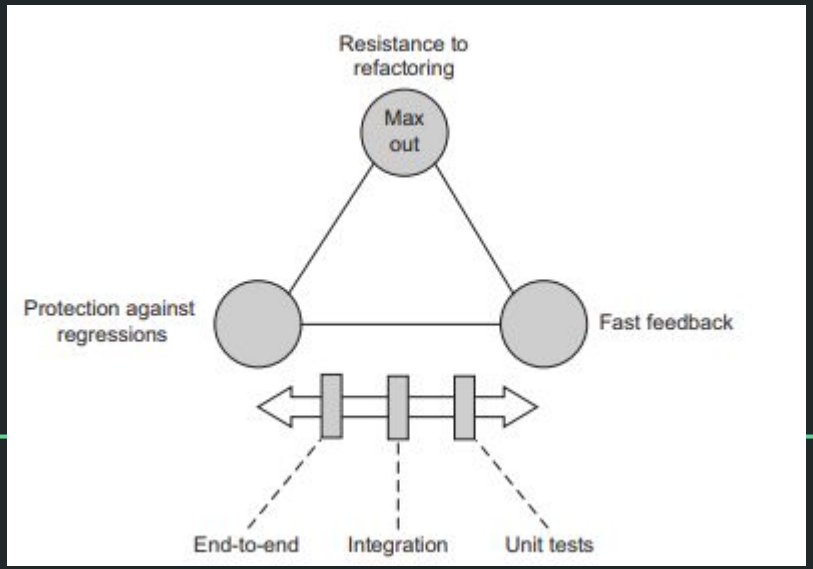
Теорема CAP и распределение тестов по уровням

В любой реализации распределённых вычислений возможно обеспечить не более двух из трёх следующих свойств:

- согласованность данных (англ. consistency) — во всех вычислительных узлах в один момент времени данные не противоречат друг другу;
- доступность (англ. availability) — любой запрос к распределённой системе завершается корректным откликом, однако без гарантии, что ответы всех узлов системы совпадают;
- устойчивость к разделению (англ. partition tolerance) — расщепление распределённой системы на несколько изолированных секций не приводит к некорректности отклика от каждой из секций.

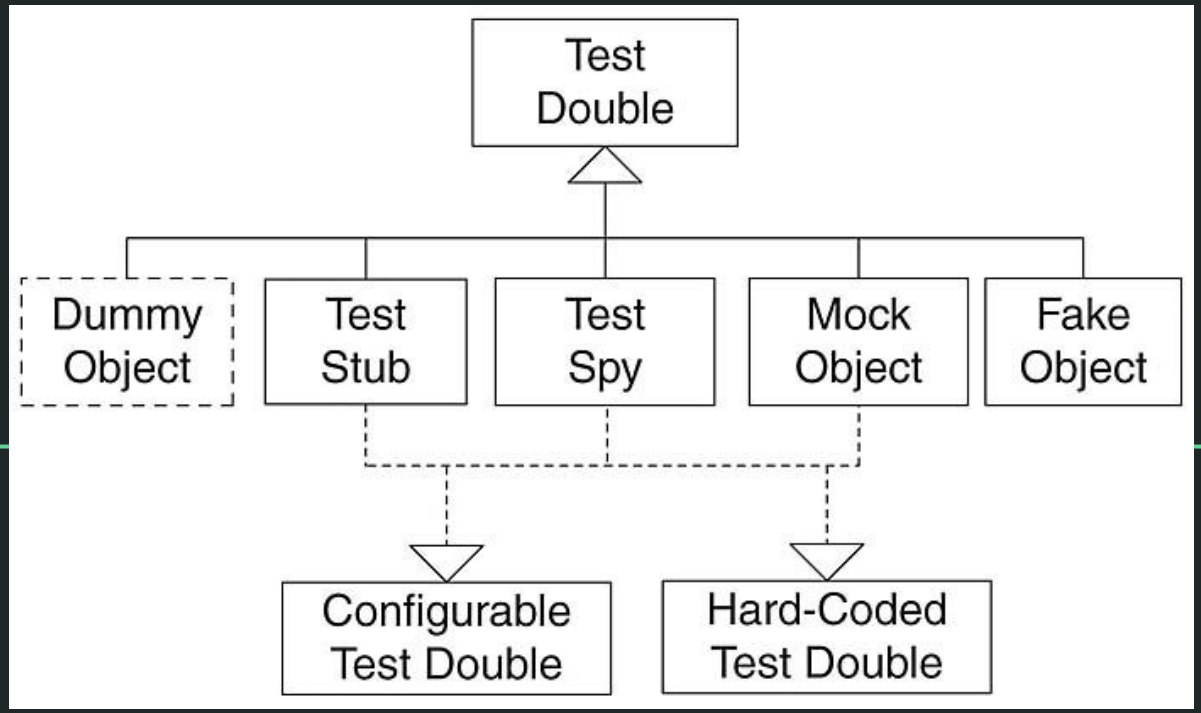


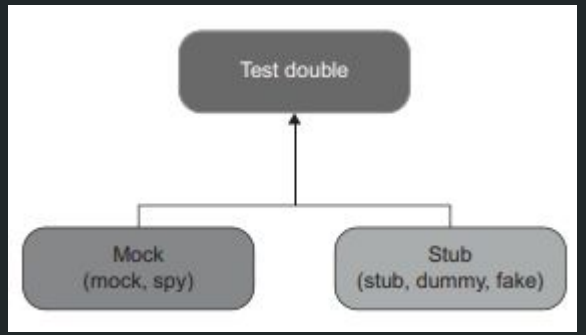
Ещё об уровнях тестов



	Protection against regressions	Resistance to refactoring
White-box testing	Good	Bad
Black-box testing	Bad	Good

Mocks / Stubs / Test Double

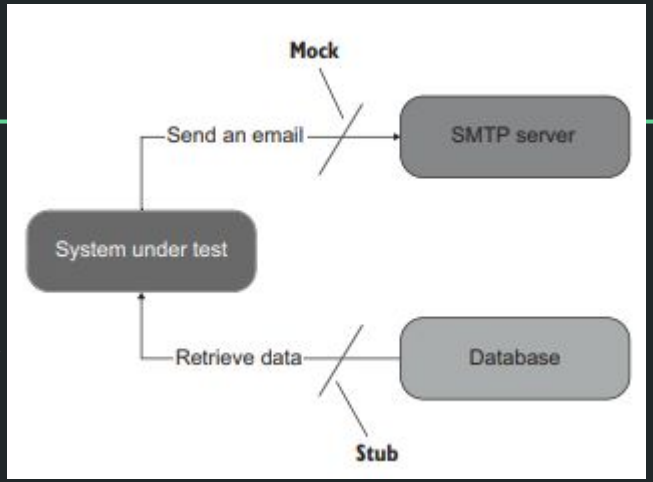




Mocks / Stubs / Test Double

Mock -- эмулировать и проверять исходящие взаимодействия. Это вызовы и взаимодействия, которые исполняются SUT к зависимым объектам, чтобы изменить их состояние

Stub -- эмулировать и входящие взаимодействия. Это вызовы и взаимодействия, которые исполняются SUT к зависимым объектам, чтобы запросить и получить данные



Любые другие отличия -- детали реализации.
Spy -- вариант Mock без использования фреймворка мокирования.
Dummy -- примитивный Stub (возвращать любое not null value).
Fake -- Stub для функционала, которого ещё нет.

Mock

Mock как использование фреймворка и создание mock test double

```
[Fact]
public void Sending_a_greetings_email()
{
    var mock = new Mock<IEmailGateway>();
    var sut = new Controller(mock.Object);

    sut.GreetUser("user@email.com");

    mock.Verify(
        x => x.SendGreetingsEmail(
            "user@email.com"),
        Times.Once);
}
```

Uses a mock (the tool) to create a mock (the test double)

Examines the call from the SUT to the test double

Mock как использование фреймворка и создание stub test double

```
[Fact]
public void Creating_a_report()
{
    var stub = new Mock<IDatabase>();
    stub.Setup(x => x.GetNumberOfUsers())
        .Returns(10);
    var sut = new Controller(stub.Object);

    Report report = sut.CreateReport();

    Assert.Equal(10, report.NumberOfUsers);
}
```

Uses a mock (the tool) to create a stub

Sets up a canned answer

А где проводить assert?

Assert данных, которые получены через stub -- анти-паттерн.
Необходимо комбинировать mock и stub

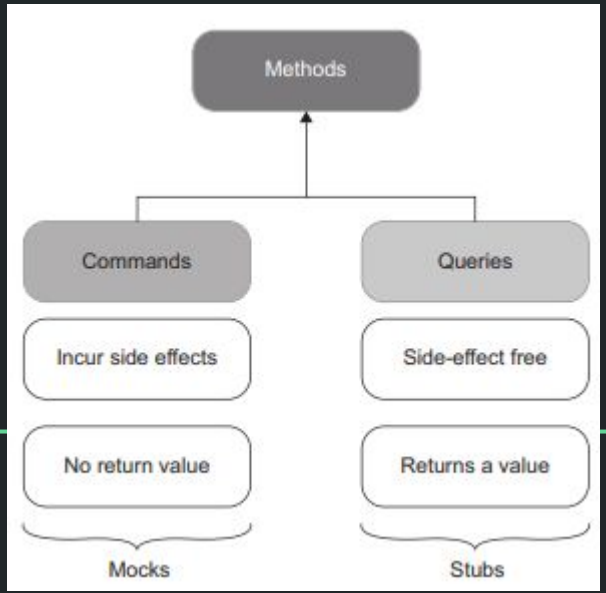
```
[Fact]
public void Purchase_fails_when_not_enough_inventory()
{
    var storeMock = new Mock<IStore>();
    storeMock
        .Setup(x => x.HasEnoughInventory(
            Product.Shampoo, 5))
        .Returns(false);
    var sut = new Customer();

    bool success = sut.Purchase(
        storeMock.Object, Product.Shampoo, 5);

    Assert.False(success);
    storeMock.Verify(
        x => x.RemoveInventory(Product.Shampoo, 5),
        Times.Never);
}
```

Sets up a canned answer

Examines a call from the SUT

Command-query separation \ Принцип сегрегации команд и запросов

Принцип гласит, что метод должен быть либо командой, выполняющей какое-то действие, либо запросом, возвращающим данные, но не одновременно. Другими словами, задавание вопроса не должно менять ответ. Более формально, возвращать значение можно только чистым (т.е. детерминированным и не имеющим побочных эффектов) методом. Следует отметить, что строгое соблюдение этого принципа делает невозможным отслеживание количества вызовов запросов.

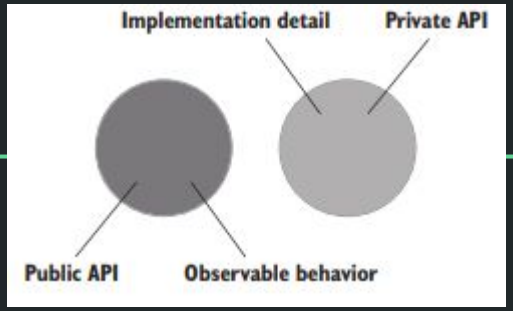


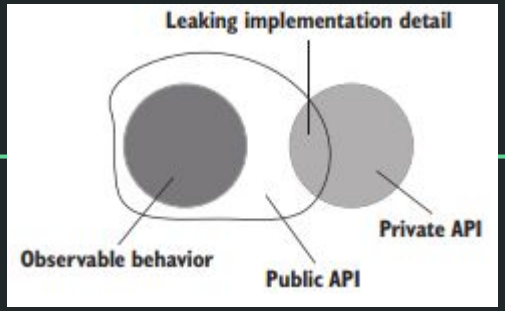
Проверять поведение ПО (Observable Behavior), а не детали реализации (Implementation Details)

Что проверять:

- операции, которые осуществляют вычисления или вызывают изменение в порядке этих вычислений
- операции, которые выполняют конечную задачу клиента \ программного интерфейса
- операции, которые приводят к изменению состояния системы

Всё, что не относится к этим категориям -- это детали реализации





	Observable behavior	Implementation detail
Public	Good	Bad
Private	N/A	Good

Mock\stub с точки зрения архитектуры