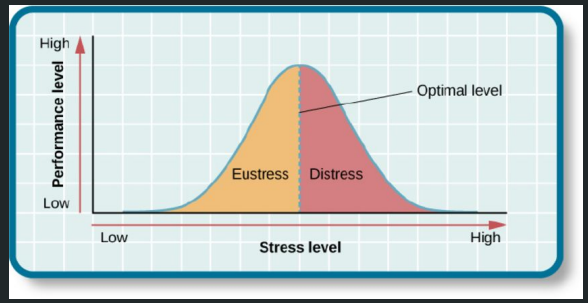
Место Unit-тестов в проекте

- Текущее состояние Unit-тестирования в Enterprise проектах
- Цель Unit-тестирования
- Последствия разработки без Unit-тестирования
- Последствия разработки с “плохим” Unit-тестированием
- Базовое представление о метриках покрытия тестами
- Атрибуты Unit-теста и набора тестов (Suite)

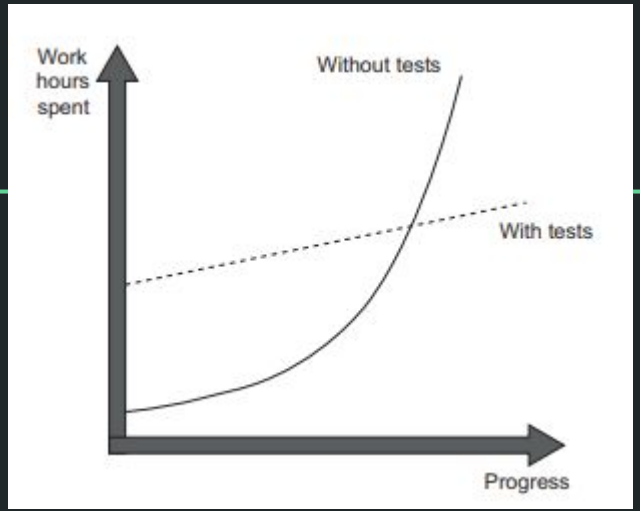
Enterprise-проект (да, по ним проще всего описать любые методики):

- Высокая сложность бизнес-логики
- Высокая продолжительность жизненного цикла ПО
- Умеренное количество данных
- Низкий или умеренные требования к производительности ПО



Место Unit-тестов в проекте

- Связь между тестированием (unit testing) и проектированием (code design)
- Не все тесты в равной степени полезны (not all tests are created equal)
- Разница между test code и production code
- Что такое регрессия?
- Так какая цель?
- Что такое “хороший” и “плохой тест” ?
- Software Entropy



Кратко об оценке покрытия кода тестами

- Как определить покрытие кода тестами?
- Source Lines of Code — SLOC (физические строки, логические строки и строки комментариев)

$$\text{Code coverage (test coverage)} = \frac{\text{Lines of code executed}}{\text{Total number of lines}}$$

- И ещё раз: дизайн кода и тестов непосредственно влияет процент покрытия, но не гарантирует проверки всех возможных случаев
- 80% 100%

```
public static bool IsStringLong(string input)
{
    if (input.Length > 5)
        return true;

    return false;
}
```

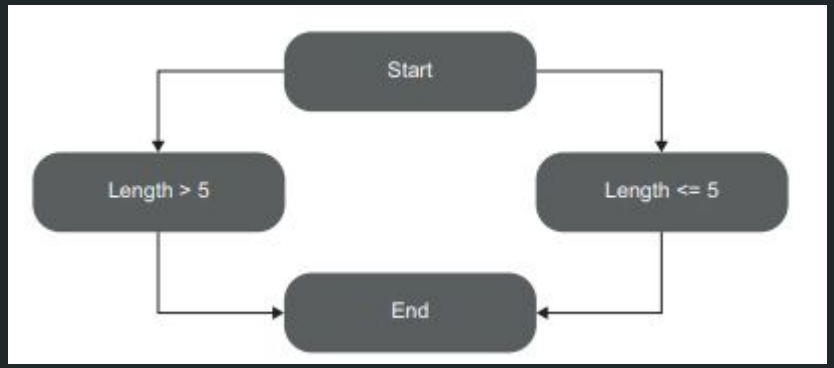
Not covered by the test

Covered by the test

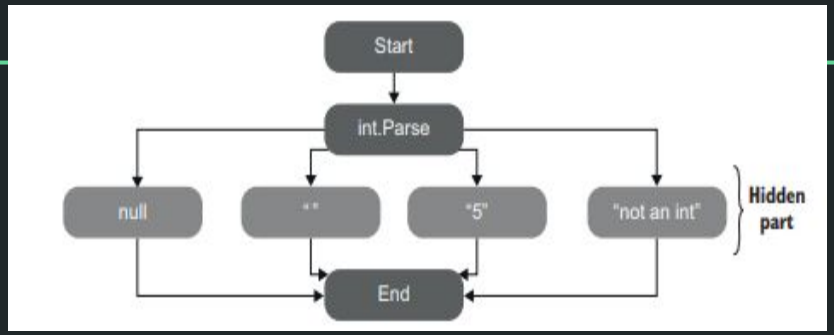
```
public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

```
public static bool IsStringLong(string input)
{
    return input.Length > 5;
}

public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

Нужно ли покрывать тестами все варианты?




```
public static bool IsStringLong(string input)
{
    return input.Length > 5;
}

public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```




```
public static int Parse(string input)
{
    return int.Parse(input);
}

public void Test()
{
    int result = Parse("5");
    Assert.Equal(5, result);
}
```

Что говорить разработчику, который хвастается 100% покрытием его кода тестами?

Что нужно сделать, чтобы тесты заработали?

Интегрировать тесты в процесс разработки с помощью CI\CD: запускать тесты после каждого коммита в бранч и после каждого мержа в release\master.
Приучить разработчиков проверять локально существующие тесты перед созданием pull request.

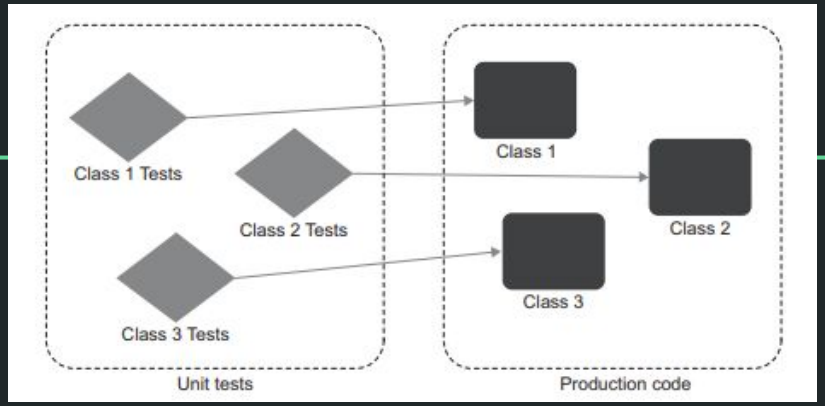
Тесты должны проверять конкретные критические секции кода, а не просто накручивать процент покрытия:

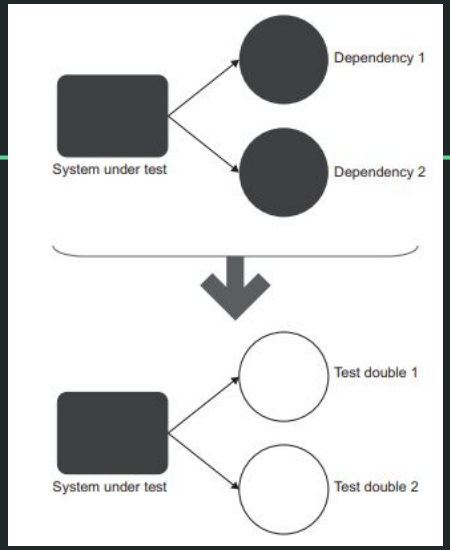
- Инфраструктурный код
- Внешние сервисы и зависимости
- Код, который связывает части проекта в единое окружение

Максимизировать ценность теста (возможность отлавливать регрессии)
Минимизировать трудозатраты на поддержку кода теста

Как определить Unit-тест

- Проверяет малую часть кода (Unit) -- модуль, класс, сценарий использования
- Время выполнения теста незначительно -- порядок десятков миллисекунд
- Модуль проверяется изолированно от других
- Классический “Детройтский” и Лондонский подход к определению изоляции
- Test Double, System Under Test (SUT), Method Under Test (MUT)

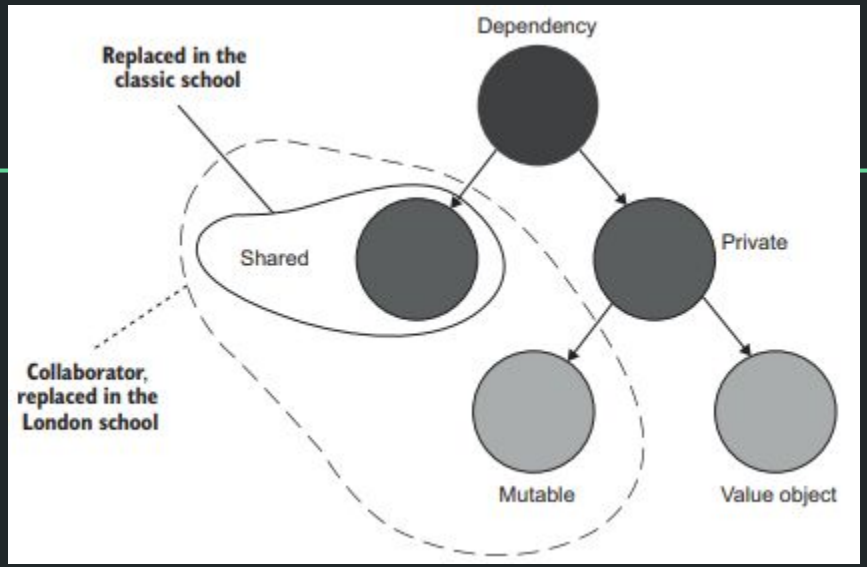


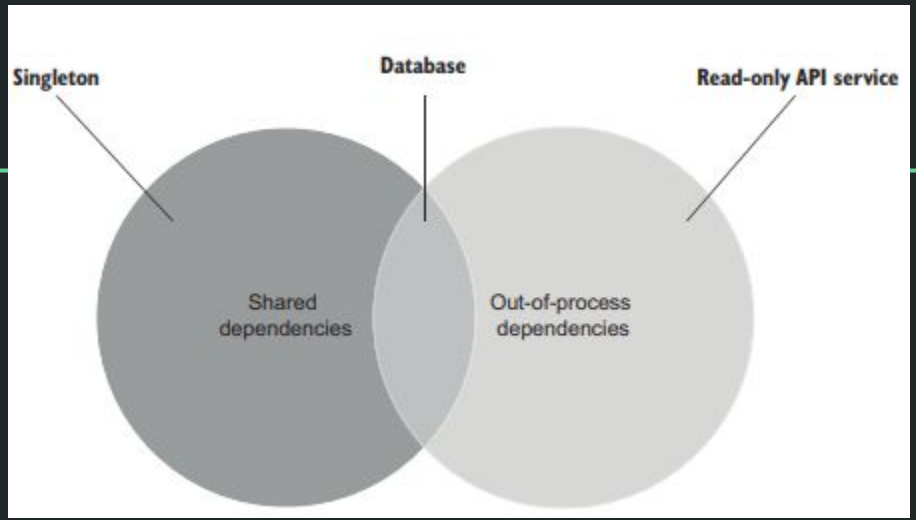


Изолирование объекта тестирования

Уровни изоляции: private, shared, out-of-process, volatile, mutable, immutable

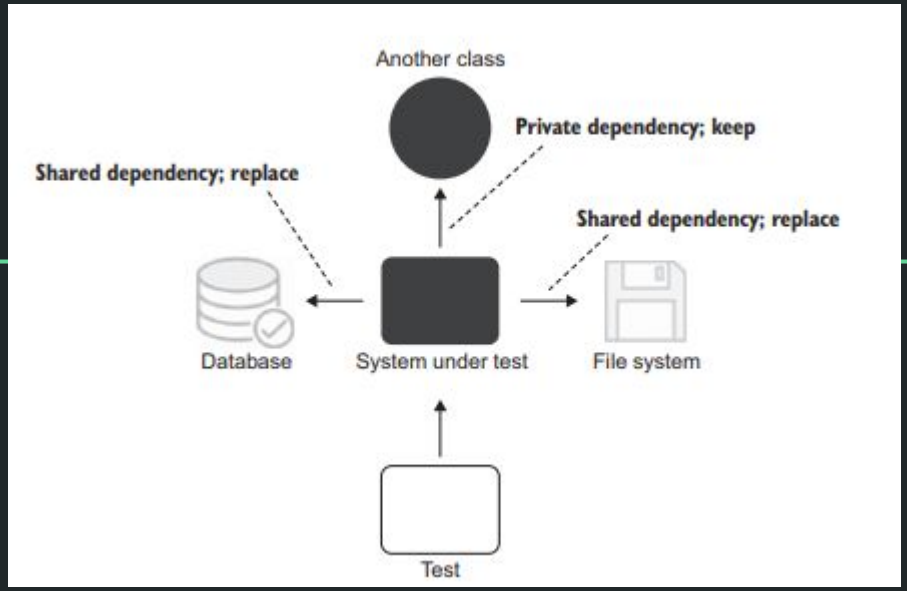
	Isolation of	A *unit* is	Uses test doubles for
London school	Units	A class	All but immutable dependencies
Classical school	Unit tests	A class or a set of classes	Shared dependencies





Изолирование объекта тестирования

Уровни изоляции: private, shared, out-of-process



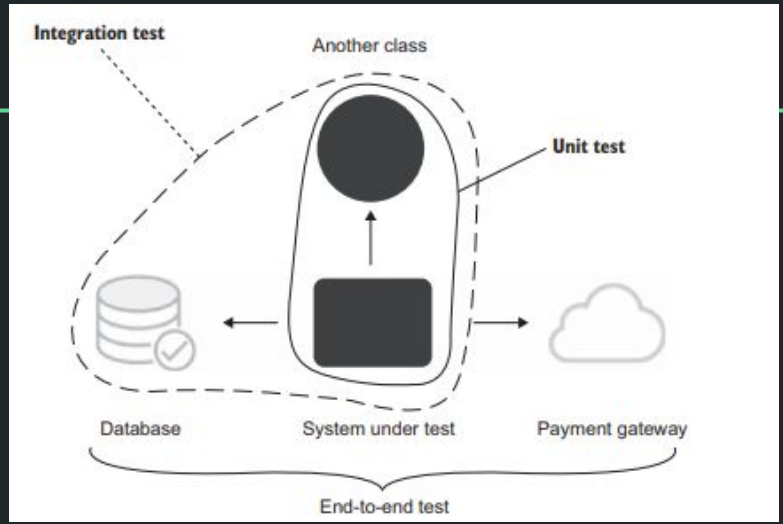
Плюсы\минусы двух подходов к тестам

Лондонский подход:

- Лучшее гранулирование, тест-класс проверяет конкретный класс и только его
- Чем больше проект, тем сложней использовать классический подход из-за транзитивных зависимостей между классами
- Если тест не пройден -- очевидно в каком именно классе ошибка
- Логично подводит к TDD и прочим современным методикам

Классический подход:

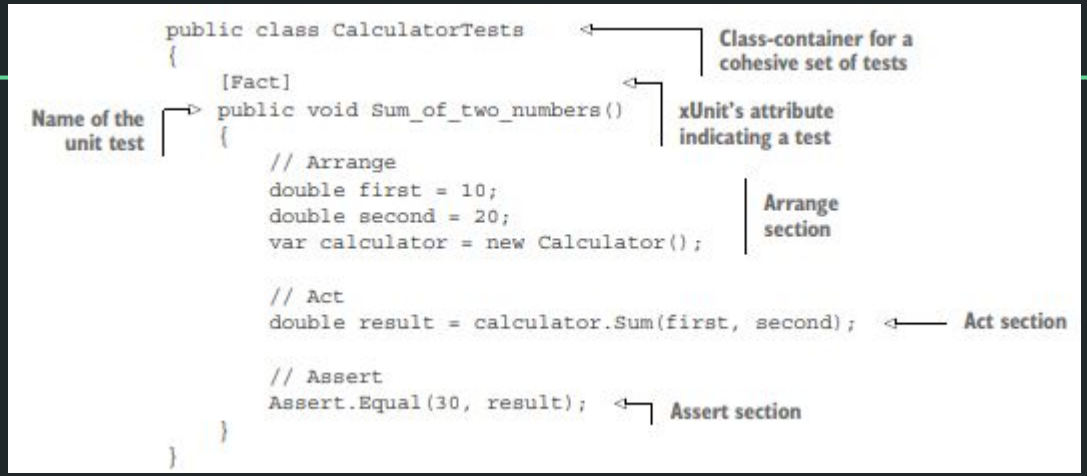
- На самом деле это естественный способ строить интеграционные тесты
- Единственный подход к End-to-end тестам (E2E)
- Но обе эти категории относятся уже не к чистому unit-тестированию



Паттерн написания Unit-тестов AAA

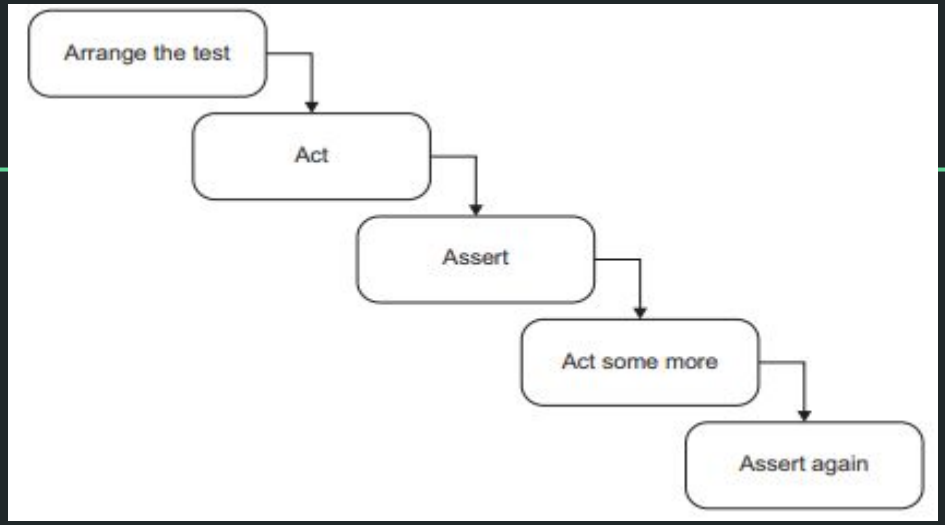
AAA -- arrange, act, and assert
Arrange -- подготовить объект тестирования, контекст выполнения методов, инициализироваться переменные среды
Act -- выполнить метод, который требуется протестироваться
Assert -- проверить выходной результат и параметры среды после выполнения метода

Given-When-Then -- аналогичный паттерн, используется в BDD\TDD фреймворках



Анти-паттерны AAA

1. (A)* вместо AAA -- неправильно определен объект тестирования, либо дизайн изначальных методов ужасен.
2. A if A1 else A2 A -- те же причины, что у (A)*. If в таких тестах усложняет поддержку кода тестов.
3. Что делать?

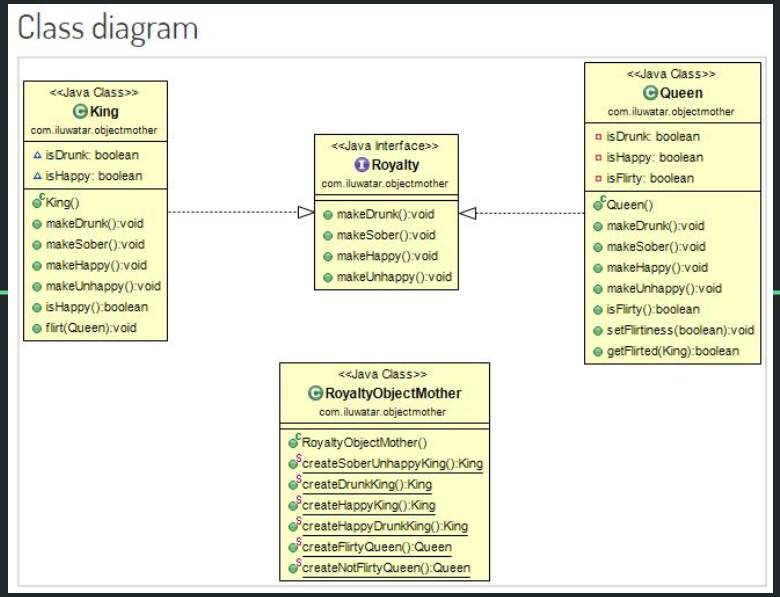


Рекомендации по наполнению AAA теста

1. Act -- один тест, одна строчка в секции, один вызов одного метода
2. Arrange -- почти половина всего кода теста будет тут:
    a. Object Mother pattern -- создание, модификация и хранение объектов, которые нужны для выполнения тестов
    b. Builder pattern -- генерация тестовых данных для избегания хардкода и обеспечения покрытия разных наборов входных данных для одного и того же теста
3. Assert -- оставшаяся почти половина кода теста будет тут. Unit -- это не только про код. Unit -- про поведение объекта тестирования.
4. Teardown-методы, Test fixture и хэлперы (требования к читаемости кода никто не отменял для тестов)
5. Модификация одного теста не должна приводить к изменению результатов другого
6. Немного о нейминге тестов

Object Mother

Фабрика immutable объектов с разделенными интерфейсами фабрика и билдера



Test Data Builder

Следуем принципам лондонского подхода:
избавляемся от зависимостей
на конкретные классы

```
public class Address
{
    public string Street { get; set; }
    public string City { get; set; }
    public PostCode PostCode { get; set; }

    public Address(string street, string city, PostCode postCode)
    {
        this.Street = street;
        this.City = city;
        this.PostCode = postCode;
    }
}
```

```
public class AddressBuilder
{
    private string street;
    private string city;
    private PostCode postCode;

    public AddressBuilder()
    {
        this.street = "";
        this.city = "";
        this.postCode = new PostCodeBuilder().Build();
    }

    public AddressBuilder WithStreet(string newStreet)
    {
        this.street = newStreet;
        return this;
    }

    public AddressBuilder WithCity(string newCity)
    {
        this.city = newCity;
        return this;
    }

    public AddressBuilder WithPostCode(PostCode newPostCode)
    {
        this.postCode = newPostCode;
        return this;
    }

    public AddressBuilder WithNoPostcode()
    {
        this.postCode = new PostCode();
        return this;
    }

    public Address Build()
    {
        return new Address(this.street, this.city, this.postCode);
    }
}
```