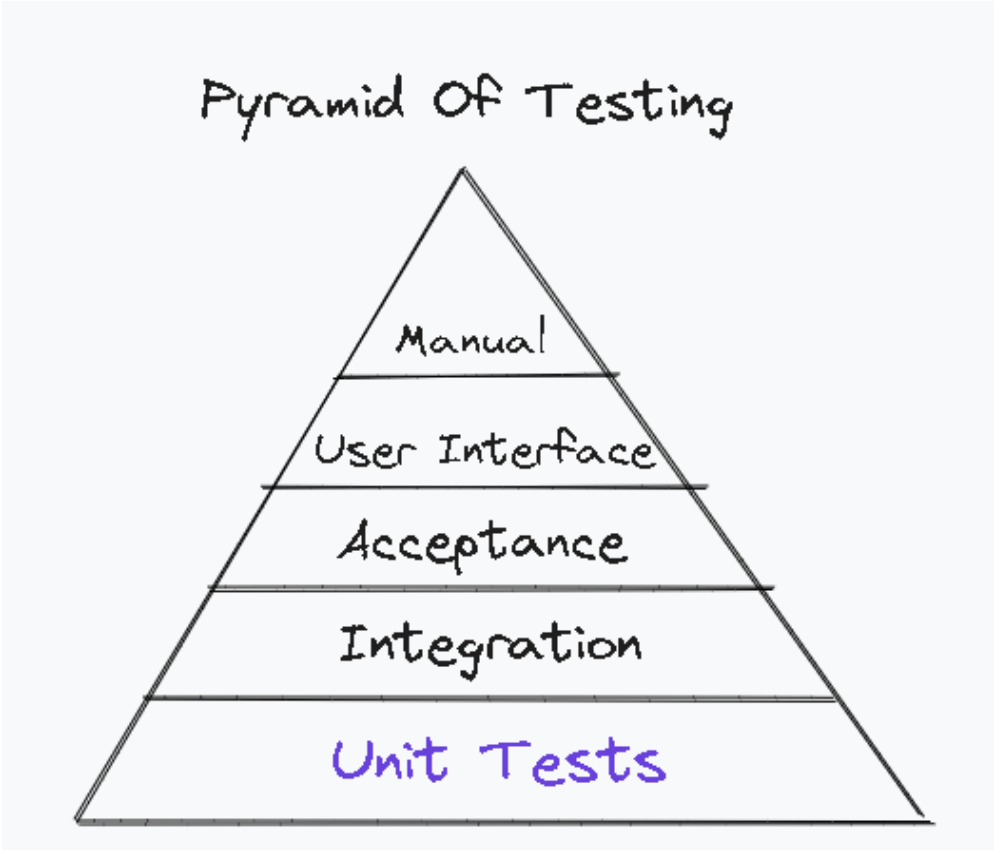
Модульное тестирование (Unit Testing)

Модульное тестирование (Unit Testing) – это тип тестирования программного обеспечения, при котором тестируются отдельные модули или компоненты программного обеспечения.

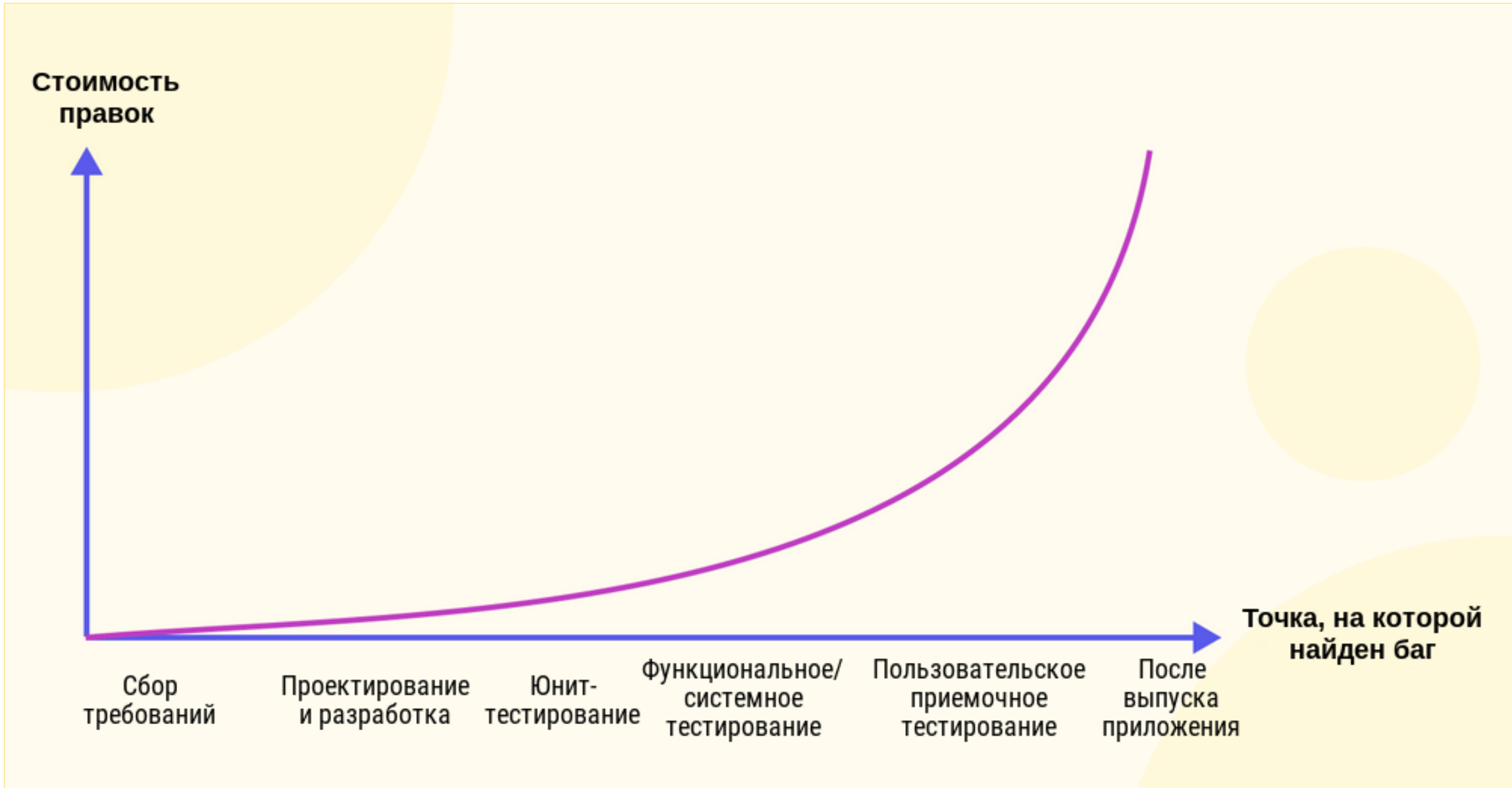
Цель модульного тестирования – проверить, что каждая единица программного кода работает должным образом.

- Модульное тестирование это метод тестирования WhiteBox, который обычно **выполняется разработчиками** на этапе кодирования приложения.
- Юнит-тесты **изолируют часть кода** и проверяют его работоспособность.
- **Единицей для измерения может служить** отдельная функция, метод, процедура, модуль или объект.
- Юнит-тесты находятся на самом базовом уровне жизненного цикла тестирования.



Модульные тесты:

- ✓ **позволяют исправить ошибки на ранних этапах разработки** и сэкономить время и усилия, которые могли бы быть потрачены на поиск и исправление проблем в уже сложившемся коде;
- ✓ **помогают предотвратить появление новых ошибок** при внесении изменений в код в будущем;
- ✓ **способствуют повышению надежности кода**. Хороший набор тестов гарантирует, что код будет работать стабильно и предсказуемо даже в сложных ситуациях;
- ✓ **служат документацией к коду**. Описание ожидаемого поведения кода в явном виде делает код более понятным и облегчает его использование и поддержку для других разработчиков;
- ✓ **юнит-тесты позволяют проводить рефакторинг кода** без опасений, что существующая функциональность будет сломана.


Стоимость правок
Точка, на которой найден баг
Сбор требований
Проектирование и разработка
Юнит-тестирование
Функциональное/ системное тестирование
Пользовательское приемочное тестирование
После выпуска приложения

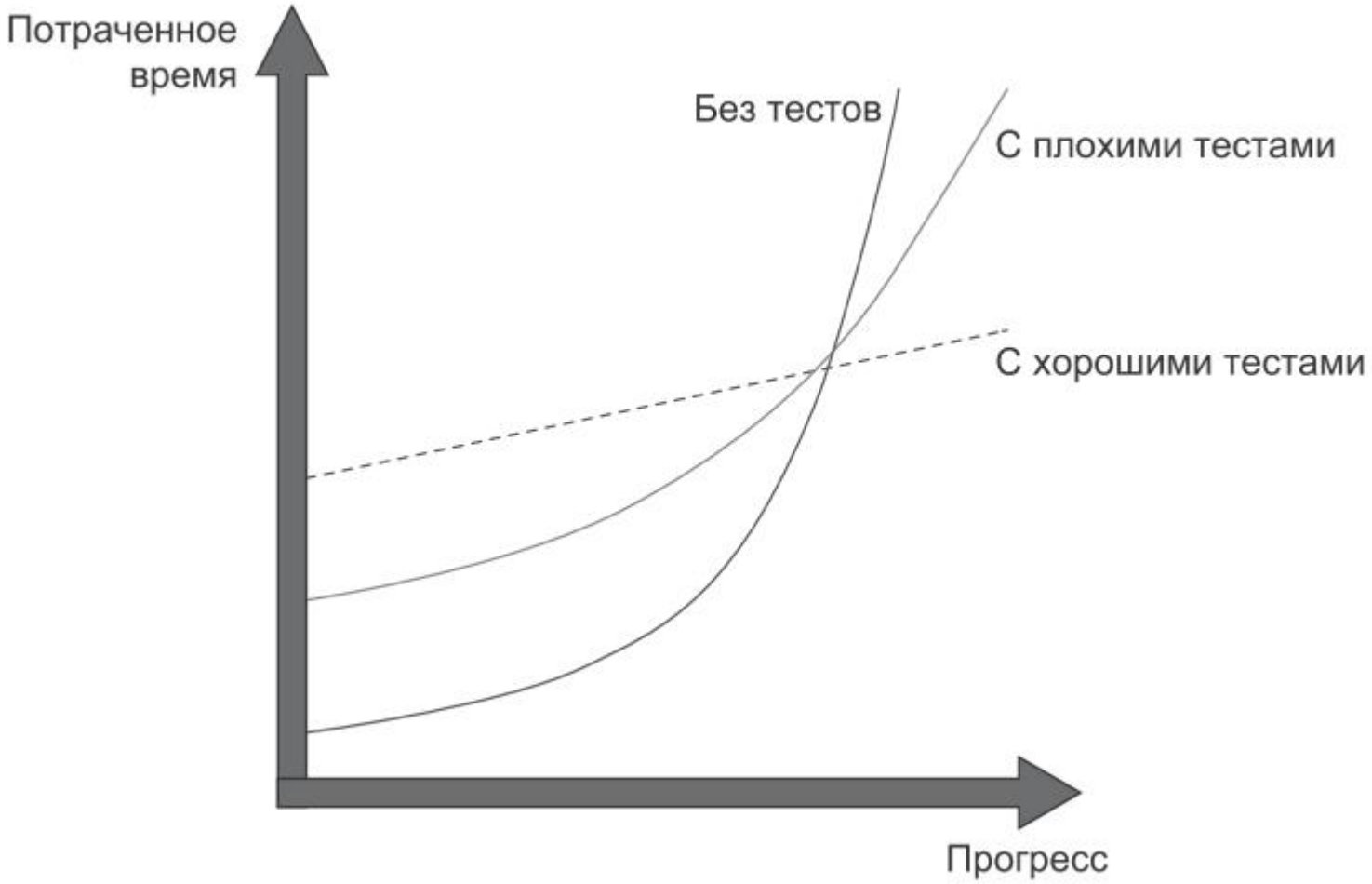
Какими качествами должен обладать юнит-тест?

Таких качеств всего три.

Юнит-тест должен проверять правильность работы небольшого фрагмента кода – юнита, должен делать это быстро и поддерживать изоляцию от другого кода.

Как тесты влияют на разрабатываемые нами продукт?

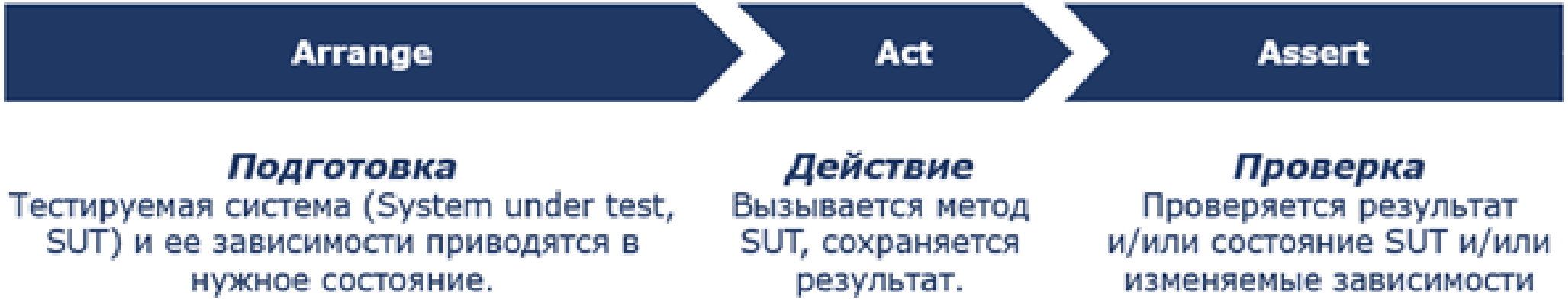
https://habr.com/ru/companies/sportmaster_lab/articles/676840/



Написание юнит-тестов включает несколько шагов:

- ✓ **Определение тестируемых «юнитов».** Выберите функции, методы или классы, которые вы хотите протестировать. Определите, какие части вашего кода должны быть покрыты тестами.
- ✓ **Разработка тестовых случаев.** Определите различные сценарии использования и состояния, которые должны быть протестированы. Напишите тестовые случаи для каждого сценария, где вы проверяете ожидаемые результаты и поведение кода.
- ✓ **Написание тестового кода.** Напишите код для каждого тестового случая, который проверяет ожидаемый результат. Используйте специальные фреймворки и библиотеки для тестирования, чтобы упростить процесс написания и выполнения тестов.
- ✓ **Запуск тестов.** Запустите все написанные тесты и проверьте их результаты. Убедитесь, что все тесты прошли успешно и код работает правильно.
- ✓ **Анализ результатов и исправление ошибок.** Изучите результаты тестов и обратите внимание на любые ошибки или неожиданное поведение. Исправьте проблемы в вашем коде и перезапустите тесты для проверки исправлений.

Структура юнит теста. Паттерн AAA



Arrange (Настройка): Создать тестовое окружение. Подготовить все необходимое для выполнения теста.

Act (Выполнение): Выполнить тестовый код. Самый маленький блок по объему, чаще всего одна строка кода – вызов метода.

Assert (Проверка): В этом блоке выполняется серия проверок – например, результат сравнивается с ожидаемым. Или проверяется, какие зависимости вызывались, в какой последовательности и какими параметрами. Этот блок отвечает на вопрос, правильно ли работает тестируемая система и фактически заключает в себе ценность теста.

```
void test_function(void)
{
  /* Arrange */
  int expected = 10;
  int actual = 0;
  /* Act */
  actual = add(1,9);
  /* Assert */
  assert(actual, expected);
}
```

Структура юнит-теста

1. Тестовые фикстуры

https://qarocks.ru/unit-testing-tutorial/

Тестовые фикстуры – это компоненты модульного теста, отвечающие за подготовку среды, необходимой для выполнения тест-кейса. Они создают начальные условия для тестируемого блока, чтобы лучше контролировать выполнение теста. Тестовые фикстуры очень важны для автоматизированных модульных тестов, поскольку они обеспечивают постоянную тестовую среду для всего процесса тестирования.

Допустим, у нас есть приложение для ведения блога, и мы хотим протестировать модуль создания поста. Тестовые фикстуры должны включать:

- Подключение к базе данных
- Образец поста с заголовком, содержанием, информацией об авторе и т. д.
- Временное хранилище для обработки почтовых вложений
- Настройки конфигурации (видимость сообщений по умолчанию, параметры форматирования и т. д.)
- Тестовую учетную запись пользователя
- Песочницу (Sandbox environment) – среду для изоляции тестирования от продакшен среды и предотвращения вмешательства в реальные данные блога.

Структура юнит-теста

2. Тест-кейс

Тест-кейс – это часть кода, которая проверяет поведение другого участка кода. При помощи тест-кейсов мы можем убедиться, что тестируемый модуль работает так, как ожидается, и дает желаемые результаты. Разработчики также должны писать *asserts*, чтобы конкретно определить, какие именно результаты ожидаются.

3. Test Runner

Test Runner – это программа, которая управляет запуском модульных тестов и предоставляет отчеты о результатах тестирования. Очень важно, что test runner может запускать тесты по приоритету, а также управлять тестовой средой и настройками для выполнения тестов. Он также помогает изолировать тестируемый модуль от внешних зависимостей.

Структура юнит-теста

4. Тестовые данные

Тестовые данные должны быть тщательно подобраны, чтобы охватить как можно больше сценариев для выбранного блока кода и обеспечить высокое тестовое покрытие. Как правило, предполагается подготовить данные для:

- **Обычных случаев**: ожидаемые входные данные для тестируемого блока кода.
- **Граничных случаев**: входные значения находятся на границе допустимого предела.
- **Ошибочных случаев**: недействительные входные данные, чтобы увидеть, как блок кода реагирует на ошибки (посредством сообщений об ошибках или определенного поведения).
- **Угловых случаев**: входные данные представляют собой крайние случаи, которые оказывают значительное влияние на блок кода или систему.

Структура юнит-теста

5. Моки

Моки – это, по сути, заменители реальных зависимостей тестируемого модуля. В модульном тестировании разработчики стараются проверить работу отдельных частей кода независимо друг от друга. Однако иногда для проверки одной части кода может понадобиться использовать другие компоненты программы.

Например, у нас есть класс User, который зависит от класса EmailSender для отправки уведомлений по электронной почте. В классе User есть метод sendWelcomeEmail(), который вызывает EmailSender для отправки приветственного письма только что зарегистрированному пользователю. Чтобы протестировать метод sendWelcomeEmail() без фактической отправки электронных писем, мы можем создать **мок-объект класса** EmailSender.

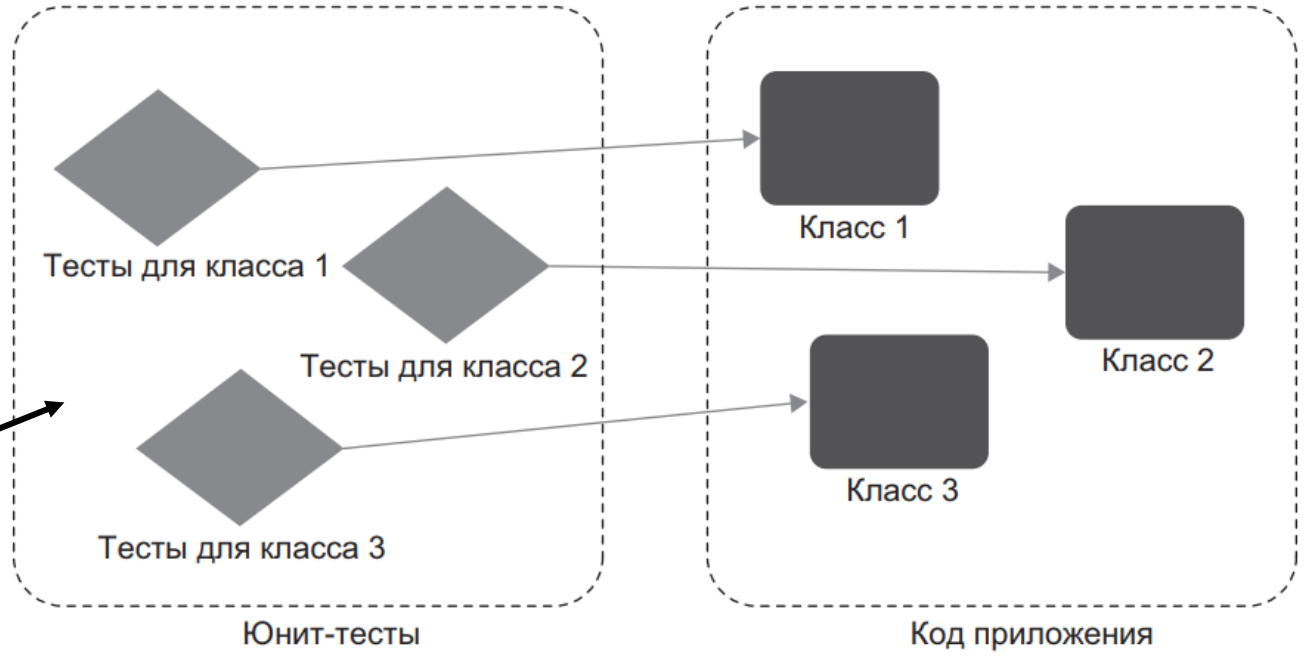
Что означает «изоляция кода» в юнит-тестировании?

Разработчик может изолировать единицу кода для более качественного тестирования. Эта практика подразумевает копирование кода в собственную среду тестирования. *Изоляция кода помогает выявить ненужные зависимости между тестируемым кодом и другими модулями или пространствами данных в продукте.*

Если класс имеет зависимость от другого класса или нескольких классов, все такие зависимости должны быть заменены на тестовые заглушки (test doubles). Это позволяет сосредоточиться исключительно на тестируемом классе, изолировав его поведение от внешнего влияния.

Тестовая заглушка (test double) — объект, который выглядит и ведет себя как его рабочий аналог, но в действительности представляет собой упрощенную версию, более удобную для тестирования.

Изоляции юнит-тестов позволяет установить простые правила тестирования рабочего код: одному классу соответствует один тест.



Тестовые заглушки (тестовые двойники)

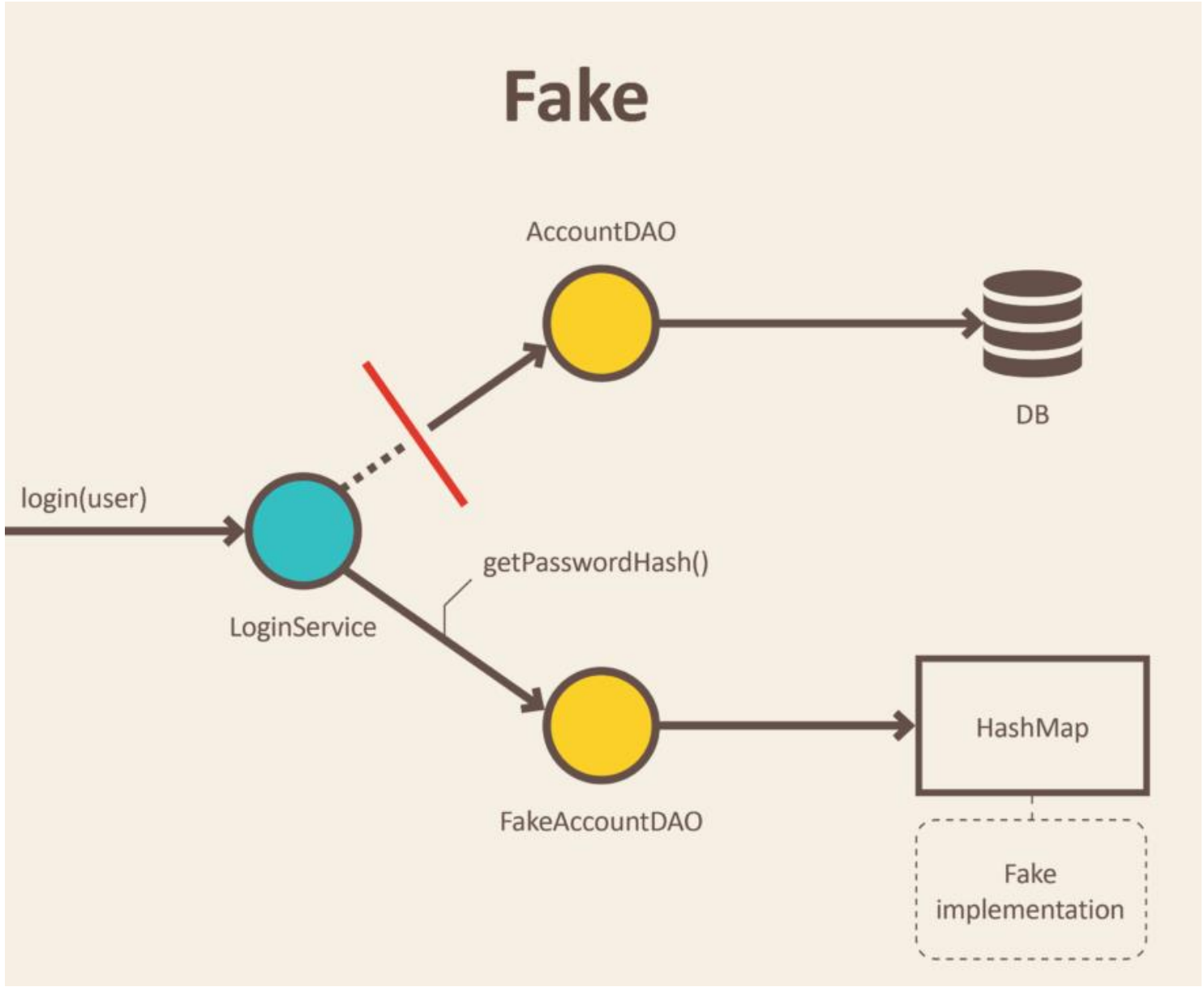
Тестовый двойник – это объект, который может заменить реальную часть кода для тестирования, примерно как дублер заменяет актера в фильме. Существует пять основных видов тестовых двойников: пустышка (dummy), фейк (fake), заглушка (stub), мок (mock), шпионы (spies).

- ✓ **Пустышки.** Например, пустышки используются для заполнения списка параметров, чтобы код компилировался и компилятор “не ругался”.
- ✓ **Фейки.** Например, объект базы данных, который мы можем использовать исключительно в тестовых сценариях, в то время как в продакшене используются реальные данные из БД.
- ✓ **Заглушки.** Допустим, у нас есть объект, который при вызове метода должен получить данные из базы данных для ответа. Вместо реального модуля мы можем использовать заглушку , которая вернет статичные данные.
- ✓ **Моки.** Моки очень похожи на заглушки, но они более ориентированы на “взаимодействие”. В качестве примера отлично подойдет функциональность, которая вызывает службу отправки сообщений по электронной почте. Нас не интересует отправлено ли нужное письмо. Нам нужно убедиться, что сервис отправки сообщений по электронной почте в принципе вызывался.
- ✓ **Шпионы.** Шпионы – это те же заглушки, но записывающие информацию о том, кто, как и когда их вызвал. Для примера снова подойдет сервис отправки сообщений по электронной почты, который фиксирует количество отправленных сообщений.

Fake

Имитация (или поддельные объекты) — это объекты, имеющие рабочие реализации, но не такие, как у настоящих рабочих объектов. Обычно они имеют упрощённую версию реального объекта.

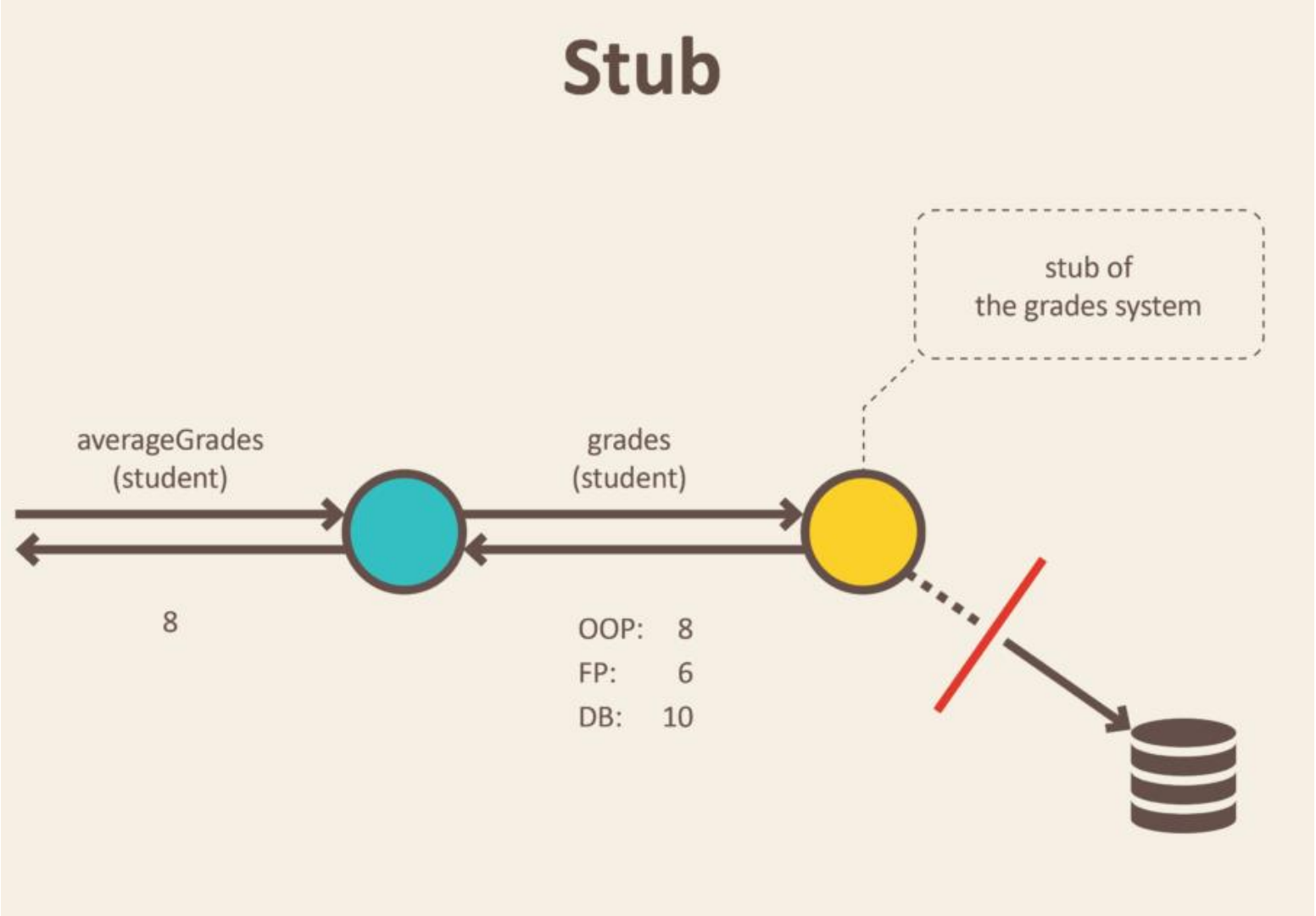
В качестве примера может быть реализация в оперативной памяти объектов доступа к данным (Data Access Object) или репозиторий (Repository). Реализация поддельных объектов не будет привлекать базу данных, но будет использовать простую коллекцию для хранения данных. Это позволяет нам выполнять интеграционный тест сервисов без участия базы данных и выполнения тем самым трудоёмких запросов.



Stub (заглушка)

Заглушка— объект, содержащий предопределённые данные и использует их для ответа на вызовы во время тестов. Она используется, когда мы не можем или не хотим привлекать объекты, которые бы отвечали реальными данными или имели бы нежелательные побочные эффекты.

В качестве примера может быть объект, который должен получить некоторые данные из базы данных в качестве результата при вызове метода. Вместо реального объекта, мы вводим заглушку и определяем в ней, какие данные она должна вернуть.



✓ **Избегайте команд if в тестах**

Наличие в тестах команды if также является антипаттерном.
Тест (неважно, юнит- или интеграционный) **должен представлять собой простую последовательность шагов без ветвлений**.
Присутствие команды if означает, что тест проверяет слишком много всего. Следовательно, такой тест должен быть разбит на несколько тестов. Ветвление затрудняют чтение и понимание тестов.

✓ **Снижайте количество тест-кейсов для юнита**

В среднем на юнит должно приходиться **от 1 до 9 тест-кейсов**. Это очень хороший индикатор качества юнита - если тест-кейсов больше или хочется их сгруппировать, нам точно нужно разделить его на два и больше независимых юнитов.

✓ **Избегайте секций Act больше, чем из одного вызова**

Если для теста требуется вызвать сначала один метод тестируемой системы, потом другой, высока вероятность того, что API тестируемой системы спроектировано неправильно.

Для упрощения написания и запуска тестов существуют специальные фреймворки для разных языков программирования, например:

- ✓ **unittest**: встроенный модуль для тестирования в языке программирования Python. Он предоставляет широкие возможности и включает в себя возможности для создания фейковых объектов (mock objects).
- ✓ **pytest:** популярный фреймворк для тестирования в Python. Обладает простым и понятным синтаксисом и предоставляет множество функций для удобного написания и выполнения тестов.
- ✓ **JUnit:** фреймворк для тестирования в языке Java. Предоставляет средства для написания и запуска юнит-тестов и широко используется в разработке на Java.
- ✓ **NUnit:** фреймворк для тестирования в языке C#. Предоставляет возможности для создания и запуска тестов в C# и обладает богатым функционалом.
- ✓ **Mocha:** фреймворк для тестирования в JavaScript. Позволяет писать тесты с использованием синтаксиса, близкого к естественному языку, и обладает широкой поддержкой различных типов тестов.

Пример тестирования приложения с использованием unittest Python

Модуль calc.py

```python
def add(a, b):
    return a + b

def sub(a, b):
    return a-b

def mul(a, b):
    return a * b

def div(a, b):
    return a / b
```

```python
import unittest
import calc

class CalcTest(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        self.assertEqual(calc.div(8, 4), 2)

if __name__ == '__main__':
    unittest.main()
```

Как измерить покрытие кода тестами?

Для этого есть две чаще всего используемые метрики – процент покрытия строк (code coverage) и процент покрытия логических ветвей (branch coverage) кода.

$$\textbf{Code coverage} = \frac{\textit{Количество выполненных тестами строк кода}}{\textit{Общее количество строк кода}}$$

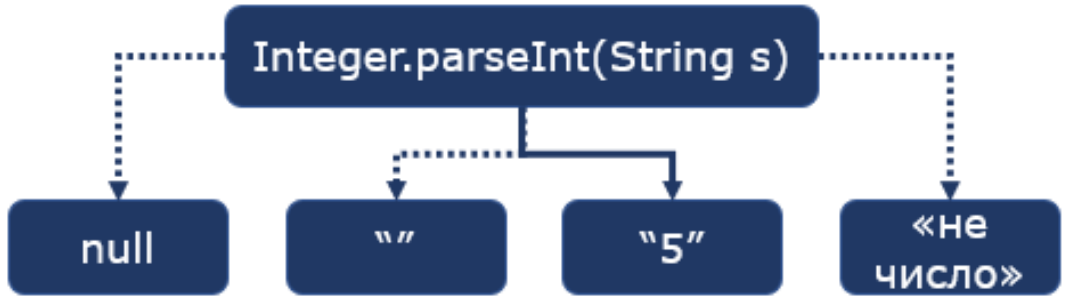
$$\textbf{Branch coverage} = \frac{\textit{Количество выполненных тестами ветвей}}{\textit{Общее количество ветвей}}$$

Но даже стопроцентное покрытие тестами не гарантирует хорошего качества тестов. И вот почему. Эти метрики не отражают некоторые важные детали, такие как наличие и полноту проверок (asserts).

Пример

```
public static Integer parse(String numberAsString){
      return Integer.parseInt(numberAsString);
   }
```

```
@Test
   void parseTest() {
      assertEquals(Integer.valueOf(5), parse("5"));
   }
```



Есть метод, который принимает строку и превращает её в число. Очень простой метод. На него написан простой тест, который передаёт строку со значением «5» и сравнивает число 5 с тем, какое число возвращает метод.

Если мы посмотрим на этот тест и попробуем оценить метрики покрытия, мы увидим, что для этих метрик покрытие будет 100%. Но, к сожалению, если мы заглянем в библиотечный метод **parseInt**, мы увидим, что на самом деле мы проверили только один из четырех вариантов. Поэтому с одной стороны у нас покрытие 100%, но с другой стороны мы покрыли по факту только 25% кода.

Пример использования моков Python

```
def get_joke():
    url='https://api.chucknorris.io/jokes/random'
    response=requests.get(url)

    if response.status_code==200:
        joke=response.json()['value']
    else:
        joke="No jokes"
    return joke

def len_joke():
    joke=get_joke()
    return len(joke)
```

В модуле реализованы две функции

Первая **get_joke()** – посылает запрос по url-ссылке и из полученного ответа сохраняет шутку в переменной joke.

Для данной функции нужно написать минимум два юнит теста, для покрытия каждой ветви функции

Вторая функция **len_joke()** возвращает длину шутки из первой функции.

Здесь достаточно одного юнит теста, который проверит как правильно функция считает длину строки

Для проверки работы функции **len_joke()** нужно знать какую строку вернет функция **get_joke(),** но **get_joke()** возвращает рандомную строку, поэтому вместо вызова **get_joke()** мы обратимся к тестовой заглушке с заранее известной строкой, длина которой будет нам известна.

Юнит-тест для функции len_joke()

```
class TestJoke(unittest.TestCase):
    # указываем путь к объекту который хотим замокать
    # декоратор patch создает фиктивный объект класса MagicMock()
    # и передает ссылку на него в декорируемый объект
    @patch(‘main.get_joke')
    def test_len_joke(self, mock_get_joke):
        mock_get_joke.return_value='ahaha‘   # возвращаем фейковую шутку
        self.assertEqual(len_joke(),5)    # проверяем как функция len_joke() вычисляет
                                                    длину шутки
```