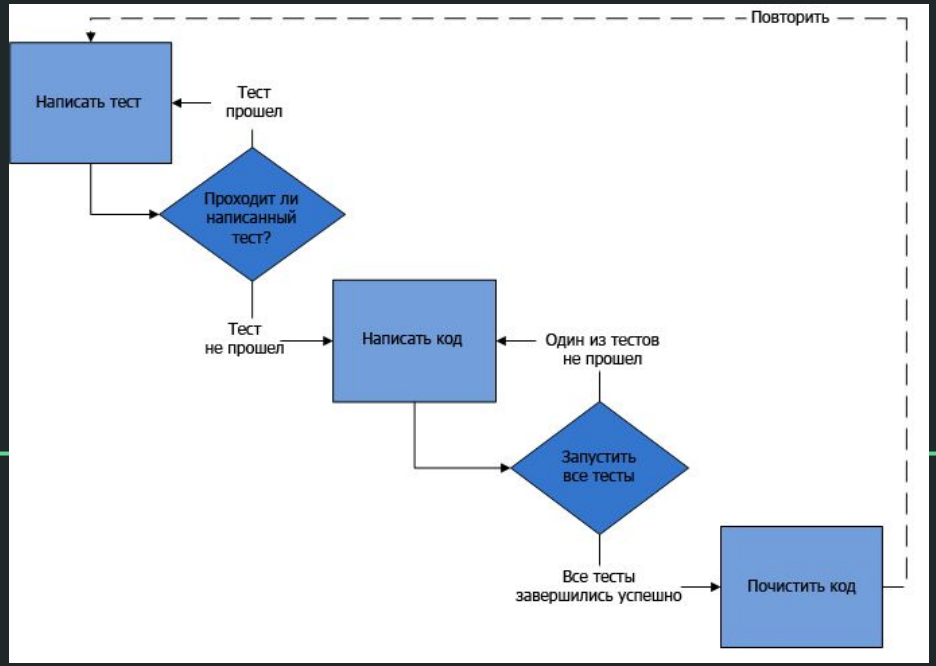
TDD в общем

- Test Driven Development
- Короткие итерации
- Написание тестов в начале каждой итерации
- Пишутся разработчиками для разработчиков
- Тесты легко пишутся
- Acceptance Test-driven development
- Unit test-driven development
- Mock / Stubs нужны для всего, что не является объектом тестирования
- Это уровень Unit тестов



- Не нужно пытаться писать тесты в TDD нотации для интеграционных тестов
- Паттерн Arrange/Act/Assert ➜ Given/When/Then
- Иногда используется паттерн Given/When/Expect/Where
- Антоним к TDD -- TLD (test last development), классический подход, когда и тестирование и написание тестов проводится после итерации разработки
- Реализуются сразу в коде

Плюсы TDD

Объективные (оцениваемые \ заметные):

- Дополнительная техническая спецификация\документация на код
- Минималистичный дизайн классов -- YAGNI (you ain’t going to need it)
- Документация всегда актуальная
- Если не получается сформулировать условия для тестов -- значит разработчику не ясны требования
- Лаконичный дизайн public API классов (т.к. паттерн тестирования AAA форсится)
- Минимум внешних зависимостей
- Легко масштабируемый код
- Отсутствие “бесполезных” тестов
- Увеличение покрытия кода тестами (автор лекции не считает процент покрытия поводом для гордости)
-

Неочевидные (о них говорят, но не понятно как оценить, либо вообще спорные):

- Уменьшение временных затрат на отладку кода
- В крупных проектах с уже внедренным TDD легче вносить новый код
- Упрощение рефакторинга (есть противоречие с минималистичным дизайном классов)
- Сокращение числа corner cases
- Безопасный рефакторинг (на самом деле даже если Unit тест был добавлен после кода приложения, то вклад в устойчивость к рефакторингу уже сделан)

Минусы TDD

- многопоточные и асинхронные сценарии тяжело описать в контексте unit test
- временные затраты: тяжело объяснить заказчику зачем тратить время на такой подход
- временные затраты: тяжело объяснить некоторым разработчикам, что читаемость их кода не менее важна, чем его оптимальность
- время на анализ и удаление\изменение delta coverage
- затягивание этапа Unit-тестирования и написания кода ведет к уменьшению времени других этапов тестирования
- использовать для нового кода проекта, написанного и поддерживаемого не по TDD, практически нереально
- сложность кода тестовой инфраструктуры пропорциональна сложности проекта

Frameworks TDD/BDD и где они обитают

Frameworks:

- Java -- SPOCK for TDD, Cucumber for BDD
- JavaScript -- Js-test-runner for TDD
- Python -- pyUnit + nose for TDD/BDD
- C# -- xUnit + LightBDD, NUnit + NBehave
- C++ -- Igloo, CBehave for BDD

Ну и вот простой тест функции на Spock, которую написал разработчик:

```
class SqrtSumAlgSpecTest extends Specification {
    Algorithm alg
    def "Sqrt sums scenarios"(){
    when:
        alg = new SqrtDecompositionSum(input.toArray(new int[0]))
    then:
        outputSumm == alg.calcSummBetween(leftIndex, rightIndex)
    where:
        input                                  | leftIndex | rightIndex| outputSumm
        [5, 10, -3, 17, 12, 1, -2, 13, -12]    |2          |5          |27
        [5, 8, 13, 5, 21, 6, 3, 7, -2, 4, 8, 12]|3         |10         |52
}
}
```

Frameworks TDD/BDD и где они обитают

Вот так выглядит описание функции для аналитика или тестировщика:

```
Feature: Sqrt Sums Algorithm Feature
In order to ensure that my algorithm works
As a Developer
I want to run a quick Cuke4Duke test

Scenario Outline: Sqrt Sums Alg Scenario
Given The input array <input array>
When The calc sum between <Left index>, <Right index>
Then The summ is <output summ>.

Examples:
|input array                          |Left index     |Right index|output summ|
|5, 10, -3, 17, 12, 1, -2, 13, -12    |2              |5          |27         |
|5, 8, 13, 5, 21, 6, 3, 7, -2, 4, 8, 12 |3           |10         |52         |
```

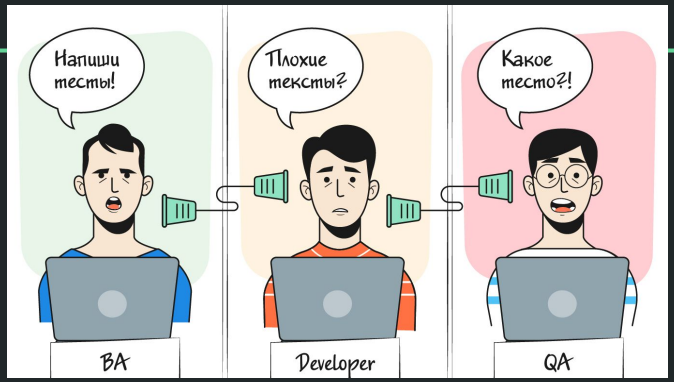
Это был пример Java + Cucumber

А вот реализацию написал разработчик:

```
public class SqrtsumsalgFeature
{
    private Algorithm alg;
    private int result;

    @Given ("^The input array ([\\d\\s\\-\\,]*)$")
    public void theInputArray(String input)
    {
        String[] split = input.split(",");
        int[] arrayInput = new int[split.length];
        for (int i = 0; i < arrayInput.length; i++)
        {
            arrayInput[i] = Integer.valueOf(split[i].trim());
        }
        alg = new SqrtDecompositionSum(arrayInput);
    }

    @When ("^The calc sum between ([\\d]*), ([\\d]*)$")
    public void theCalcSumBetween(int L, int R)
    {
        result = alg.calcSummBetween(L, R);
    }

    @Then ("^The summ is ([\\d]*).$")
    public void theSummIs(int expectedResult)
    {
        Assert.assertThat(result, is(expectedResult));
    }
}
```

BDD в общем

- Behavior Driven Development -- это расширение Test Driven Development
- Если TDD -- уровень Unit testing, то BDD -- integration, system integration и E2E
- Сначала пишем описание на человеческом языке, а потом реализацию в коде
- BDD проверяет пользовательские сценарии aka use cases aka user scenarios aka user stories
- Паттерн верхнего уровня: Title/Narrative/Scenarios/Steps
- Паттерн нижнего уровня (реализация Steps): Given/When/Then/Where
- Для описания верхнего уровня чаще всего используется язык Gherkin
- Потребности в Mock/Stubs всё так же обусловлены уровнем реализации

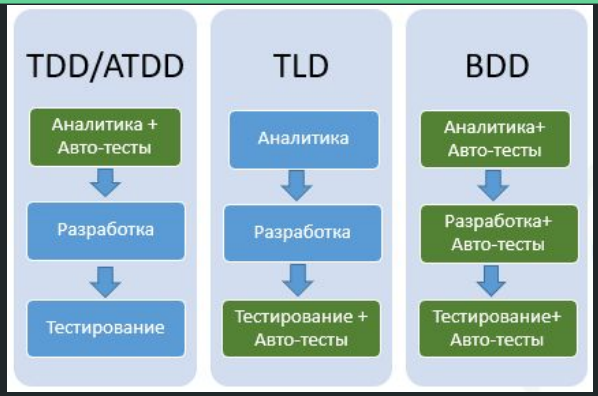


Плюсы и минусы BDD

Плюсы:

- коммуникация между разработчиками\аналитиками\заказчиками\ тестировщиками формализована
- разумный способ автоматизации E2E тестов для того чтобы сократить число ручных тестов
- спецификация и документация на уровне процессов
- спецификация и документация понятные не только разработчикам
- такие тесты хороши для проведения smock / sanity тестов
- при наличии большого числа реализаций тестов -- новые тесты могут быть добавлены без участия разработчиков

- документация новых процессов сразу на нескольких уровнях



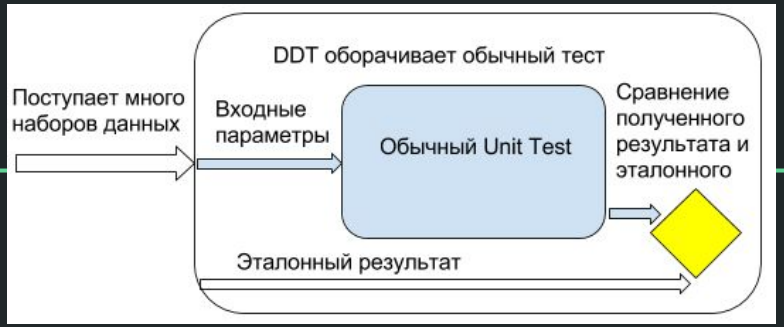
Минусы:

- тяжеловесные фреймворки ➔ увеличение затрат времени разработчиков на тесты
- дополнительный уровень абстракции (не изоляции)
- все минусы автоматизированных System Integration, UI и E2E тестов замаскированы человеческим описанием, но никуда не делись

DDD и DDT в общем

Data-Driven Design

- Позволяет быстро разработать приложение или прототип
- Удобно проектировать (кодогенерация по схеме и т.п.)
- Маленький проект -- маленькая вариативность данных -- ускоряет разработку и проектирование
- Может приводить к анти-паттернам и уходу от ООП
- На больших проектах приводит к хаосу, сложной поддержке и т.п.
- Внезапный и яркий пример такого подхода -- Sims



Data-Driven Testing

- тест умеет принимать набор входных параметров,
  и эталонный результат или эталонное состояние,
  с которым он должен сравнить результат,
  полученный в ходе прогонки входных параметров
- arrange -- входные параметры и состояния\результаты
- assert -- сравнение результата отработки метода и предоставленного ожидаемого результата
- act -- метод, который может принимать разные варианты входных данных

Подготовка данных для тестирования при DDT

Вход:

- вектора параметров для тестируемой функции\метода
- вектора ожидаемых результатов для тестируемой функции\метода
- состояние базы данных, можно брать из SQL дампа, а можно генерировать программно
- частью эталонного выходного состояния опять-таки может быть состояние базы данных
- эталонный результат может быть настолько сложным, что может представлять из себя дерево последовательности вызовов функций\операторов тестируемого объекта

Выход:

- вектора полученных результатов и их сравнения с эталоном
- состояние базы данных
- дерево последовательности вызовов функций\операторов тестируемого объекта

Форматы данных: XML, JSON, XLS(Excel), Yaml

Когда нужно DDT кроме DDD:

- процесс полностью описывается последовательностью входных\выходных данных
- накопленная сложность систем -- белый или серый ящик не работает
- от взаимосвязи входных параметров (порядок в векторе) зависит выходной результат
- много copy-paste в тестах -- значит можно перегруппировать

Подготовка данных для тестирования при DDT

Большинство BDD фреймворков имеют синтаксический сахар для DDT

```
Scenario outline: Puchase confirmations

  Purchase confirmations are the most critical payment type. We
  want to confirm payments as quickly as possible, but we must
  wait for third-party card processing. We also expect that some
  clients will experience longer delays due to network issues outside
  of our control.

  The performance is expected to degrade if the transaction volume
  exceeds current peak levels (10,000 transactions per hour).
  (Confirmed by James, 4th August 2020).

  Given the current transaction volume is <volume per hour>
  When a new purchase transaction is executed
  Then it should be confirmed within <period>, <percent> of the time.

  | volume per hour | period | percent |
  |          10,000 |     2s |     97% |
  |          10,000 |     5s |     99% |
  |          10,000 |    10s |   99.9% |
  |          20,000 |     5s |     97% |
  |          20,000 |    10s |     99% |
  |          20,000 |    15s |   99.9% |
```

Лабораторная работа №3

Задание:

- Реализовать E2E тест из лабораторной работы №2 с помощью средств TDD/BDD с передачей разных датасетов
- Провести профилирование E2E теста из лабораторной работы №2

Требования:

- Использовать паттерн Given\When\Then\Where с помощью средств выбранного фреймворка
- Вектора значений\эталонных результатов можно передавать в теле теста; передавать \ читать состояние БД или дерево исполнения вызовов не обязательно (но приветствуется)
- Профилирование по потреблению CPU, RAM, и оценка Garbage Collection Pauses (если в языке есть GC паузы)
- Результатом профилирования является файл отчета, который можно открыть в выбранной для профилирования утилите повторно