Разработка через тестирование (TDD). Разработка на основе поведения (BDD). Системы поддержки TDD и BDD

Лаптева Юлия
ВМИ-304

TEST DRIVEN DEVELOPMENT (TDD)

Разработка через тестирование

TDD

техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.

Зачем нужны тесты?

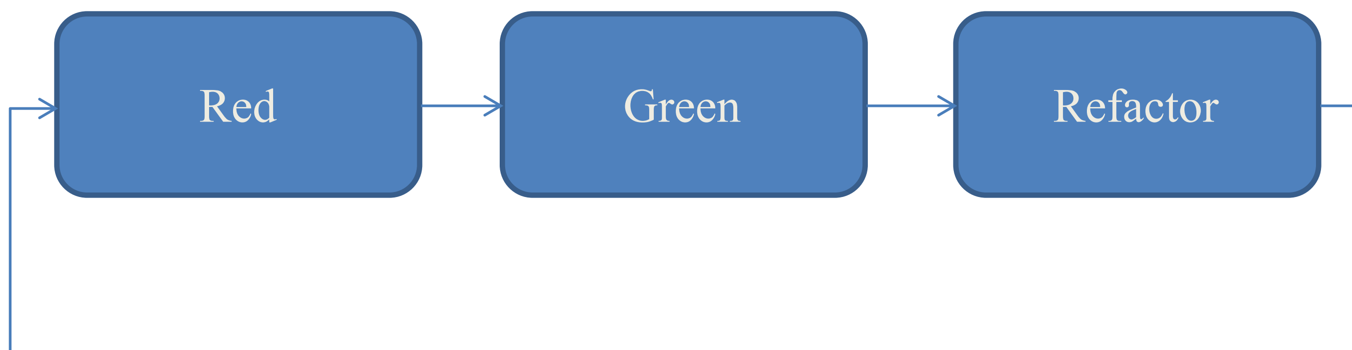
Разработка через тестирование требует от разработчика создания автоматизированных модульных тестов, определяющих требования к коду непосредственно перед написанием самого кода. Тест содержит проверки условий, которые могут либо выполняться, либо нет. Когда они выполняются, говорят, что тест пройден. Прохождение теста подтверждает поведение, предполагаемое программистом.

Testing frameworks

Разработчики часто пользуются библиотеками для тестирования (англ. testing frameworks) для создания и автоматизации запуска наборов тестов. На практике модульные тесты покрывают критические и нетривиальные участки кода. Это может быть код, который подвержен частым изменениям, код, от работы которого зависит работоспособность большого количества другого кода, или код с большим количеством зависимостей.

- Junit
- Nunit
- *Boost*

Цикл TDD



1) Написать тест для новой функциональности, которую необходимо добавить
2) Написать код, который пройдет тест
3) Провести рефакторинг нового и старого кода.

Цикл TDD

RED - В первой фазе программист пишет тест и метод-заглушку на тестируемом классе, необходимый для того, чтобы тест запустился.

- Функция-заглушка (в программировании) — функция, не выполняющая никакого осмысленного действия, возвращающая пустой результат или входные данные в неизменном виде. То же самое, что заглушка метода.

! Заглушку следует писать так, чтобы тест не выполнялся - это поможет удостовериться, что тест правильно реагирует на ошибку.

Red

```
public class ReporterTests
{
  [Fact]
  public void ReturnNumberOfSentReports()
 {
   var reportBuilder = new Mock<IReportBuilder>();
   var reportSender = new Mock<IReportSender>();

   // задаем поведение для интерфейса IReportBuilder
   // Здесь говорится: "При вызове функции CreateReports
   // вернуть List<Report> состоящий из 2х объектов"
   reportBuilder.Setup(m => m.CreateRegularReports())
      .Returns(new List<Report> {new Report(), new Report()});

   var reporter = new Reporter(reportBuilder.Object,
reportSender.Object);

   var reportCount = reporter.SendReports();

   Assert.Equal(2, reportCount);
  }
}
```

```
public class Reporter
{
  private readonly IReportBuilder reportBuilder;
  private readonly IReportSender reportSender;

  public Reporter(IReportBuilder reportBuilder,
IReportSender reportSender)
  {
    this.reportBuilder = reportBuilder;
    this.reportSender = reportSender;
  }

  public int SendReports()
  {


  }
}
```

Цикл TDD

- Метод-заглушка редактируется так, чтобы тест начал работать правильно

(при этом пишется минимальное количество кода)

GREEn

- ```
  public class Reporter
  {
    private readonly IReportBuilder reportBuilder;
    private readonly IReportSender reportSender;

    public Reporter(IReportBuilder reportBuilder, IReportSender
  reportSender)
    {
      this.reportBuilder = reportBuilder;
      this.reportSender = reportSender;
    }

    public int SendReports()
    {
      return reportBuilder.CreateRegularReports().Count;
    }
  }
  ```
  ```

Цикл TDD

- Производится рефакторинг кода
- **Рефакторинг** (*refactoring*), или **реорганизация кода** — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы
- В программировании термин *рефакторинг* означает изменение исходного кода программы без изменения его внешнего поведения. В экстремальном программировании и других гибких методологиях рефакторинг является неотъемлемой частью цикла разработки ПО: разработчики попеременно то создают новые тесты и функциональность, то выполняют рефакторинг кода для улучшения его логичности и прозрачности. Автоматическое юнит-тестирование позволяет убедиться, что рефакторинг не разрушил существующую функциональность.

Основные проблемы требующие рефакторинга

- Дублирование кода;
- длинный метод;
- большой класс;
- длинный список параметров;
- «жадные» функции — это метод, который чрезмерно обращается к данным другого объекта;
- избыточные временные переменные;
- классы данных;
- несгруппированные данные
- И т.д.

Особенности рефакторинга в TDD

При разработке через тестирования на фазе рефакторинга помимо вышеперечисленных проблем решается основная для этого этапа

привести код в наиболее полное соответствие с его назначением

Пример разработки через тестирование 1

```
public void testMultiplication()
{
        Dollar five = new Dollar(5);
        five.times(2);
         assertEqualsdO. five.amount);
}
```

Пример разработки через тестирование 2

```
class Dollar
{
     int amount;
    Dollar (int amount)
    {}
    void times (int multiplier) { }


};
```

Пример разработки через тестирование 3

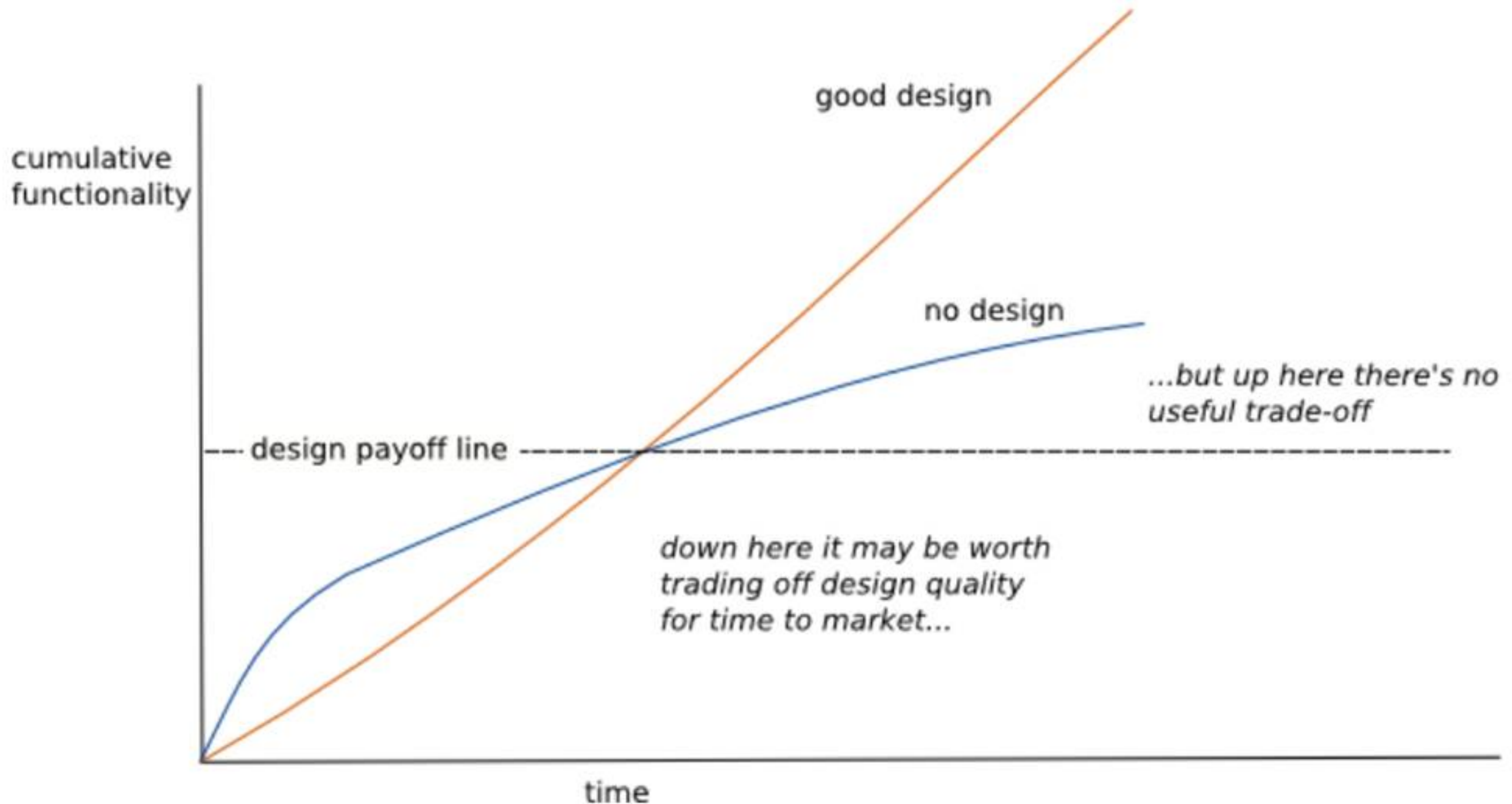
```
class Dollar
{
     int amount=10;
    Dollar (int amount) { }
    void times (int multiplier) { }
};
```

Пример разработки через тестирование 4

```
class Dollar
{
        int amount;
       Dollar (int amount){
       this.amount=amount }
       void times (int multiplier) {
       amount*=multiplier}

};
```

Когда это выгодно?



TDD за и против

Плюсы:

- Упрощение поддержки кода
- Четкая модульная структура программы
- Борьба со сложностью

Минусы:

- Увеличение времени разработки программного продукта
- Зависимоть от ТЗ

Разработка на основе поведения

BEHAVIOR-DRIVEN DEVELOPMENT

BDD

- BDD (Behavior-driven development, Разработка через поведение) - техника разработки, при котором рассматривается не результат выполнения какого-либо модуля, а та работка, которую он выполняет. Этот принцип можно рассматривать как продолжение TDD.

- Создателем техники считается Ден Норт

Особенности BDD

- BDD интересно тем, что тесты к нему пишутся с помощью сценариев.

- Сценарии – описание функциональности метода, написанное на естественном языке по определенному шаблону.

Отличие TDD от BDD

- **This class *should* do something**
- Используйте слово «поведение», а не «тест»
- BDD дает «доступный всем язык» для анализа

Общеупотребительный язык

- Для того, заказчик и разработчик могли составлять сценарии вместе, используется концепция общеупотребительных языков (ubiquitous language)
- Общеупотребительный язык – Набор базовых терминов предметной области. Является общим для заказчика и разработчика.

Системы для программной поддержки TDD и BDD

- JUnit – фреймворк, применяющийся для разработки на Java. В нем тестовые методы начинаются со слова test и наследуются от тест-класса TestCase.
- Nunit – открытая среда юнит-тестирования приложений для .NET. Она была портирована с языка Java (библиотека JUnit). Первые версии NUnit были написаны на J#, но затем весь код был переписан на C# с использованием таких новшеств .NET, как атрибуты.

Системы для программной поддержки TDD и BDD

- Cucumber - среда разработки на языке программирования Ruby. Разработчик описывает необходимое поведение в обычном тексте.
- Specflow

BDD-синтаксис Given, When, Then интуитивно понятен. Рассмотрим элементы синтаксиса:

- Given предоставляет контекст выполнения сценария тестирования, например, точки вызова сценария в приложении, а также любые необходимые данные.
- When определяет набор операций, инициирующих тестирование, таких как действия пользователей или подсистем.
- Then описывает ожидаемый результат тестирования.

Пример разработки системы с использованием BDD

- Начнем с того, что определим нашу функциональность.
- Feature: Show logged in user name
  In order to logged in as a user called “Username"
  I want to see page header displays the caption "Здравствуйте, Username!"
- Здесь мы задаем на понятном нам языке, что мы хотим увидеть от нашей функциональности.

Написание сценария

- Напишем сценарий, который будет основой для работы cucumber'a
- Scenario: Show logged in user name
  Given I am logged in as a user called "Username"
  When I visit the homepage
  Then the page header displays the caption "Здравствуйте, Username!"

- Scenario – имя сценария.
  Given… - Начальное условие (две категории и их описание)
  When.. – Если я на странице с категориями…
  Then – Я должен увидеть…

Написание сценария

- Для каждого действия также пишем соответствующие функции:
- Given /I am logged in as a user called "(.*)"/ do |name|
create_user(name)
sign_in_as(name)
end
- Then /the page header displays the caption "(.*)"/ do
|caption|page_header.should_contain(caption)
end
- Таким образом Cucumber или SpecFlow сможет интерпретировать каждый шаг, вычленить с помощью регулярных выражений параметры и запустить соответствующие тесты.