

Белорусский государственный университет информатики и  
радиоэлектроники  
г. Минск, Республика Беларусь

# Разработка программного обеспечения через тестирование

*Стрельцов Г. Ю.*

*Искра Н. А. – м.т.н., старший преподаватель*

## Оглавление

1 Введение.....	3
2 Виды тестирования .....	4
1. Модульные тесты (Unit tests) .....	4
2. Интеграционные тесты (Integration tests).....	5
3. Функциональные тесты (Functional or end-to-end tests) .....	5
3 Гибкие методологии .....	7
4 Test-driven development (TDD).....	9
5 Behavior-driven development (BDD).....	11
7 In action. Используемые технологии для написания тестов .....	15
8 Модульное тестирование.....	16
8.1 Тестирование бизнес-логики .....	16
8.2 Тестирование бизнес-логики, использующей сторонние интерфейсы .	19
8.3 Тестирование REST контроллера в модели MVC .....	20
9 Интеграционное тестирование.....	22
9.1 Тестирование отправки email почты .....	22
9.2 Тестирования слоя доступа к данным из базы данных .....	23
10 Функциональное тестирование.....	25
11 Заключение .....	27
12 Используемые источники.....	28

# 1 Введение

С каждым днём в сфере разработки программного обеспечения появляется всё больше и больше различного рода задач, концепций, методологий и требований. Одними из ключевых факторов, которые влияют на успех и качество разрабатываемого программного продукта, можно с уверенностью назвать качественную разработку требований, выбор в пользу того или иного архитектурного решения, объём и качество тестирования и скорость разработки. И чем больше и сложнее задача стоит перед командами разработчиков и тестировщиков, тем больше внимания им приходится обращать на данные факторы. Цель данной работы заключается в исследовании существующих подходов и методологий в разработке ПО и поиске баланса между выбором в пользу определённого архитектурного решения, объёмами тестирования и скоростью разработки в различных ситуациях и задачах, чаще всего встречающихся в разработке ПО. Все это позволит добиться большего качества и меньшей себестоимости программного продукта. Особый акцент сделан на тестирование. В данной работе оно служит отправной точкой, позволяющей решить поставленные задачи.

## 2 Виды тестирования

Классически тесты, разрабатываемые на протяжении жизненного цикла ПО делятся на 3 основных типа по степени изолированности, сложности и стоимости:

**1. Модульные тесты (Unit tests).** Минимальные низкоуровневые тесты для проверки минимального блока кода (юнита). Это может быть функция, метод или маленький утилитарный класс. Можно сказать, что модульные тесты проверяют принцип единой обязанности на уровне методов и мелких классов без каких-либо внешних зависимостей. Позволяют достаточно быстро проверить, не привело ли очередное изменение кода к появлению ошибок в уже оттестированных фрагментах программы, а также облегчает обнаружение и устранение таких ошибок. С точки зрения объектно-ориентированного программирования, область видимости модульных тестов не должна выходить за рамки тестируемого класса. И, как следствие, тест **не является** модульным, если он:

1. напрямую взаимодействует с хранилищем данных (база данных, файлы);
2. взаимодействует с чем-то извне используя сети;
3. затрагивает файловую систему;
4. не может работать в одно и тоже время, что и остальные юнит тесты;
5. необходимо дополнительно что-то сконфигурировать, чтобы выполнить тест;

**Признаки модульных тестов:**

- все значения в памяти;
- сложные зависимости и инфраструктура имитируются с помощью заглушек;
- нет обращений к инфраструктуре, нет потоков, нет файловой системы, нет базы данных;

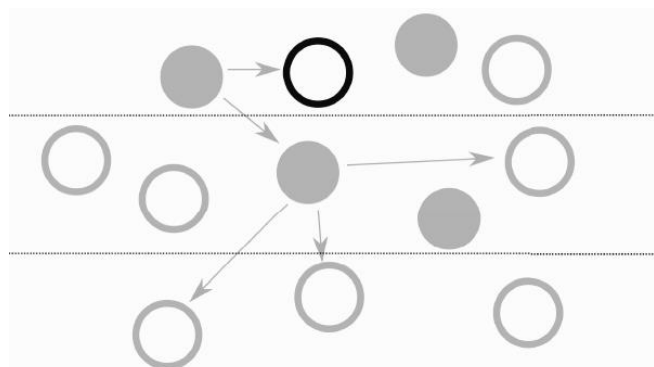


Рис. 2.1 Область видимости модульных тестов

Данный вид тестирования является самым легко реализуемым, дешевым и, как правило, самым большим по объёму в сравнении с остальными видами тестирования. Также следует отметить, что чаще всего данный вид тестов реализуется исключительно разработчиками.

**2. Интеграционные тесты (Integration tests).** Более высокий уровень абстракции, чем у модульных тестов. Это тестирование взаимодействия нескольких компонентов (С точки зрения объектно-ориентированного программирования - классов) выполняющих вместе какую-то логику или работу. Идея: у нас есть входные данные, и мы знаем, как программа должна отработать на них. Это знание, по сути, будет спецификация к тестовым данным, в которой записано, какие результаты ожидаются от программы. Тестирование же будет проверять соответствие спецификации и того, что действительно возвращает программа.

**Признаки интеграционных тестов:**

- использование реальной (не имитируемой) инфраструктуры: потоки, базы данных, файловую систему, зависимости системы;
- проверка взаимодействия компонентов;
- проверка взаимодействия с каким-либо внешним сервисом или API;

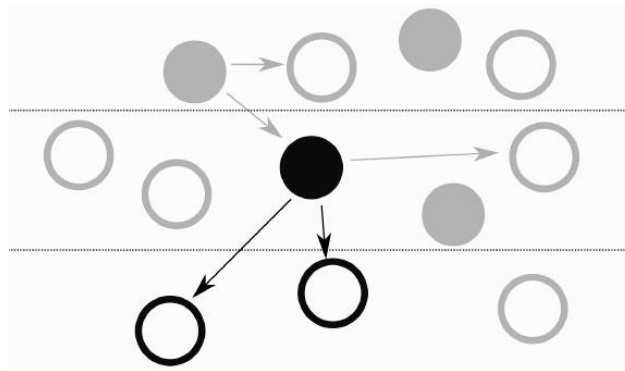


Рис. 2.2 Область видимости интеграционных тестов

Данный вид тестирования может быть реализован как разработчиками, так и тестировщиками.

**3. Функциональные тесты (Functional or end-to-end tests).** Еще более высокий уровень абстракции, чем интеграционные тесты. Тестирование производится над функциональностью, которая задана в требованиях к разрабатываемому ПО и её согласованностью со спецификацией. По большей части проверяется то, что система производит нужный ответ на заданный ввод. При этом нас не интересует внутреннее состояние компонентов системы. Можно сказать, что система в функциональных тестах — черный ящик.

## Признаки функциональных тестов:

- написание тестов с точки зрения клиента системы;
- ориентирование тестов на проверку ключевых требований, предъявляемых к конкретной функциональности;
- тесты проверяют соответствие ожиданий (спецификации) и реальности;

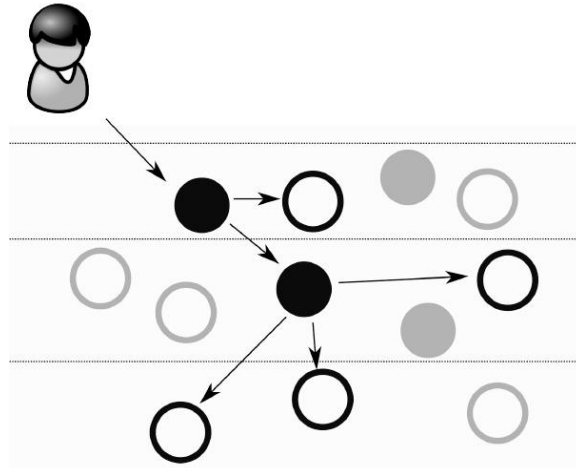


Рис. 2.3 Область функциональных тестов

Данный вид тестирования реализуется чаще всего командой тестировщиков.

Перечисленные виды тестирования не исключают друг друга. Они находятся на разных уровнях абстракции, используются на разных этапах разработки и имеют свои концептуальные особенности. С практической стороны, модульные и интеграционные тесты делают проверку с точки зрения программиста, а функциональные тесты — с точки зрения клиента. И, конечно же, на практике иногда приходится использовать смешанные типы тестов для достижения поставленных задач и удобства. К примеру, часто вместо имитации зависимости от БД или, например, какого-то внешнего сервиса (для отправки email), мы пишем гибридный вариант между модульным и интеграционным тестом. В итоге получаем «интеграционный модульный тест» (с минимальной зависимостью от БД), который в первую очередь проверяет поведение метода и, в качестве побочного эффекта, интеграцию этого поведения с БД. Это очень удобно для того, чтобы быстро проверить на начальных этапах разработки правильную работу только разработанных программных компонентов. В дальнейшем, такие гибридные тесты могут не включаться в процесс сборки в качестве автоматизированных тестов. Такую возможность часто предоставляют библиотеки и платформы для тестирования (Например, Junit и Mockito, если говорить о технологиях языка Java). Более подробно данная тема будет рассмотрена в практической части работы (In Action. Используемые технологии).

### 3 Гибкие методологии

Для эффективного применения рассмотренных видов тестирования на практике необходимо использовать гибкие методологии разработки ПО. **Методология** - систематизированный способ организации процесса или деятельности. Это повторяющийся процесс - от ранних этапов разработки (новая идея, возможность) до поддержки и сопровождения готового продукта. На рис. 3.1 изображена схема классических этапов жизненного цикла разработки ПО.

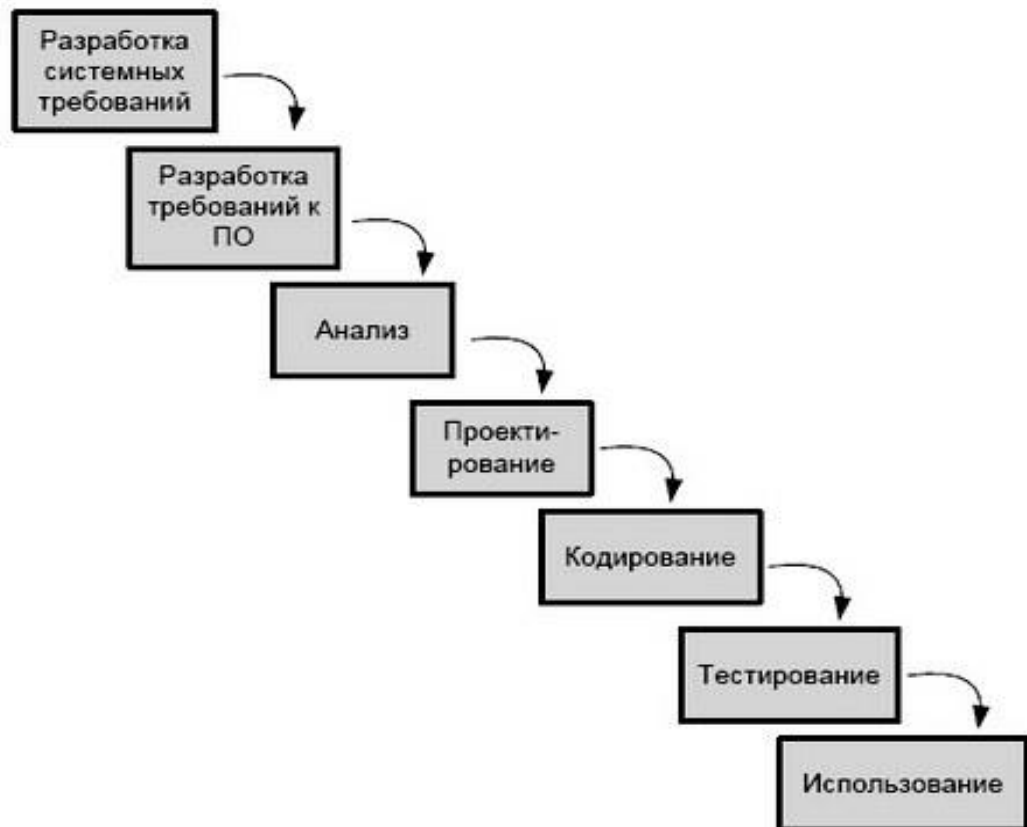


Рис 3.1 Классические этапы жизненного цикла ПО

Итак, **цель** - разрабатывать ПО быстрее, качественнее и затрачивая как можно меньше сил и ресурсов (т.е. дешевле). С точки зрения достижения этой цели, наибольший интерес представляет график на рис 3.2, демонстрирующий зависимость стоимости исправления ошибки от этапа, на котором произошёл дефект.

**Главный вывод:** чем дольше *“живёт ошибка в приложении”*, тем больше её стоимость. Идея разработки через тестирование (или Test first approach) подразумевает выполнение фазы тестирования фактически одновременно с фазами кодирования. Это позволяет на более раннем этапе разработки предупредить значительное количество ошибок, что в свою очередь приведёт нас к ускоренной разработке более дешёвого и качественного ПО.

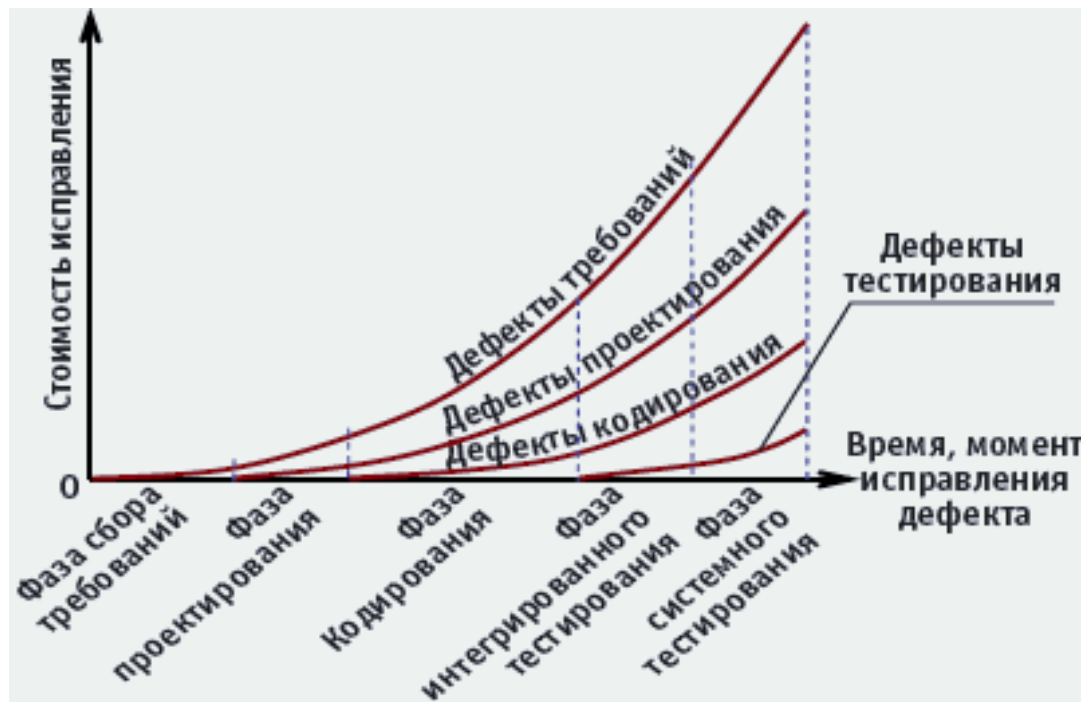


Рис. 3.2 График зависимости стоимости ошибки от момента исправления

Существуют гибкие методологии, в рамках которых возможно реализовать идею разработки через тестирование. Например, спиральная методология, экстремальное программирование или V-model. Однако существуют и более специализированные методологии, которые поощряют именно идею разработке через тестирование и именно они в данной работе представляют наибольший интерес:

1. **Test-driven development (TDD)**
2. **Behavior-driven development (BDD)**

В последующей части работы рассмотрим данные методологии более подробно.



## 4 Test-driven development (TDD)

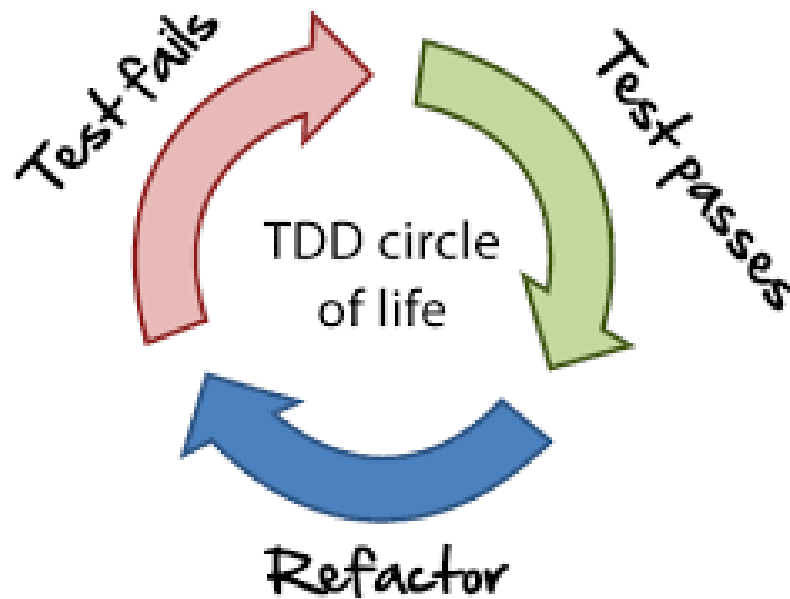


Рис. 4.1 Фазы стратегии TDD

Данная методология разработана в 2003 году Кентом Бекон, её суть заключается в цикличном выполнении трёх фаз на этапах кодирования и тестирования:

1. **Красная фаза TDD** - написание тестового случая для проверки работы некоторого функционала.
2. **Синяя фаза TDD** – изменение или доработка тестируемого кода.
3. **Зелёная фаза TDD** - успешное выполнение тестового случая.

Кроме того, выполнение данных фаз подразумевает предварительную разработку архитектуры приложения таким образом, чтобы её было возможно тестировать – и именно этот момент, по сути, является ключевым и определяющим качество разрабатываемого кода. Поскольку плохой код тестировать сложно, а порой и невозможно. И этот момент мы продумываем не после того, как весь код написан на фазе тестирования, а одновременно с кодированием.

### Достоинства TDD:

- код покрыт тестами и верифицирован;
- данные подходы заставляют разработчика заранее задумываться об архитектуре и качестве интерфейсов, иначе код будет сложно, а порой и невозможно, тестировать;
- в общем случае, затрачивается меньше времени на качественную разработку приложения;

- хорошие тесты могут легко заменить документацию, т.к. наглядно демонстрируют использование тестируемого кода;

### **Недостатки TDD:**

- высокий порог вхождения для разработчиков;
- сложность внедрения на реальном проекте среди всей команды, т. к. подход актуален только на этапах тестирования и разработки;
- ошибочный или некорректный тест приводит к написанию такого же ошибочного кода;
- много времени тратится при разработке интеграционных и функциональных тестов;
- необходимо игнорировать слишком простые/сложные ситуации для тестирования;

Следует также отметить, что данная методология применима не ко всем задачам, встречающимся в рамках разработки ПО. TDD чаще всего **эффективна** в следующих типовых задачах в процессе разработки:

- 1) разработка бизнес-логики на backend части приложения;
- 2) разработка компонентов для доступа или работы с хранилищами данных (Базы данных, файлы, облачные сервисы);
- 3) разработка сервисов для других частей приложения;

В то время как для следующих типовых задач TDD **теряет свои преимущества и трудно применима:**

- 1) разработка пользовательского интерфейса приложения;
- 2) разработка сервисов для сторонних систем и приложений;
- 3) разработка компонентов приложений, реализующих многопоточную обработку информации или логику;

Таким образом, стратегия TDD поощряет простой дизайн и внушает уверенность при разработки отдельных компонентов приложения. Однако главная проблема – на практике, если и получается применить TDD, то, как правило, высококвалифицированным и опытным разработчикам, которые хорошо знакомы с решаемой задачей. То есть высокий порог вхождения для разработчиков, сложность внедрения на уровне реального проекта и неприменимость ко всем задачам в процессе разработки – наиболее существенные проблемы стратегии TDD.

## 5 Behavior-driven development (BDD)

Из TDD методологии к разработке позже появился BDD. Формализовал и описал основные принципы данного подхода Дэн Норз (Dan North). За основу стратегии взяты основные принципы TDD, однако при создании BDD фокус сделан на то, чтобы минимизировать главные недостатки TDD, одновременно удовлетворить интересы и потребности бизнес стороны (заказчики, клиенты, бизнес-аналитики) и технической стороны (разработчики, тестировщики), тем самым вывести подход разработки через тестирование на новый уровень, который позволил бы эффективно применять его на реальных проектах. Определение BDD Дэна Норза:

**«BDD - second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high automation agile methodology»**

В этом небольшом определении Дэн Норз постарался вместить всю сущность методологии BDD. Чтобы понять его, необходимо разобраться в каждом слове.

**Second-generation** – BDD не появился из неоткуда, он строился на существующих процессах, в первую очередь таких, как экстремальное программирование, TDD, Continuous integration.

**Outside-in** – цель BDD сделать процесс разработки ПО максимально прозрачным для всех сторон. Сделать так, чтобы бизнес-люди понимали технических людей, довольно сложно. Поэтому BDD всё-таки предлагает разговаривать на языке бизнеса – сделать в процессе разработки ПО всё так, чтобы было понятно снаружи. И любой человек, который приходит в команду, даже если он не тестировщик или разработчик, он должен максимально (насколько это возможно) понимать все процессы, происходящие в проекте – в первую очередь, это касается бизнес стороны.

**Pull-based** – маркетинговая стратегия. Суть: есть человек или группа людей (stakeholders) с идеей или видением, им нужны технические специалисты с возможностями, которые могут реализовать их видение в реальный продукт, при этом не вмешиваясь ни в какие бизнес процессы и не делая больше или меньше, только то, что от них требуется.

**Multiple-stakeholder** – кроме главных заказчиков (core stakeholders), BDD подразумевает наличие и других, которым тоже необходимо давать какой-то результат. Например, enterprise architects, performance analysts, UX-дизайнеры. Они тоже участвуют в создании продукта и тоже чего-то хотят от команд разработчиков и тестировщиков, но они не говорят, что делать с

точки зрения бизнеса, однако они говорят, как это сделать лучше: более легко поддерживаемым, легко расширяемым, понятным и т.д.

**Multiple-scale** – в отличие от TDD, покрывающий только этапы разработки и тестирования ПО, или Continuous Integration, покрывающий только этапы сборки и релиза, BDD покрывает все этапы разработки ПО – начиная от появления идеи в голове заказчика до самого релиза.

**High-automation** – высокий уровень автоматизации всех процессов разработки. При этом следует отметить, что не всегда автоматизировать все процессы нужно и целесообразно, в первую очередь нужно автоматизировать такие процессы, как управление конфигурациями (Configuration Management), сборка, размещение. Только после автоматизации этих процессов, если существует реальная необходимость, следует автоматизировать другие процессы. Также в процессе автоматизации можно говорить о правиле Парето «80 / 20». Поэтому полностью исключать ручное тестирование и автоматизировать все процессы на практике не всегда следует.

**Agile методология** – то есть это всё-таки процесс разработки ПО, построенный на принципах Agile.

Чтобы рассмотреть, как подход BDD осуществляется на практике, сравним его с традиционным agile процессом (Рис. 5.1).



Рис 5.1. Традиционный Agile процесс

Выделим основные недостатки:

- владелец продукта (Product owner) пишет требования;
- разработчик пишет код так, как он понимает требования;
- тестировщик пишет тесты так, как он понимает требования;
- разработчик переделывает код после замечаний тестировщика;

- после ревью владельца продукта команда разработчиков переделывает функциональность;

Как можно заметить, главная проблема – владелец продукта, разработчик, тестировщик воспринимают требования по-своему. Что же предлагает в этом плане BDD?

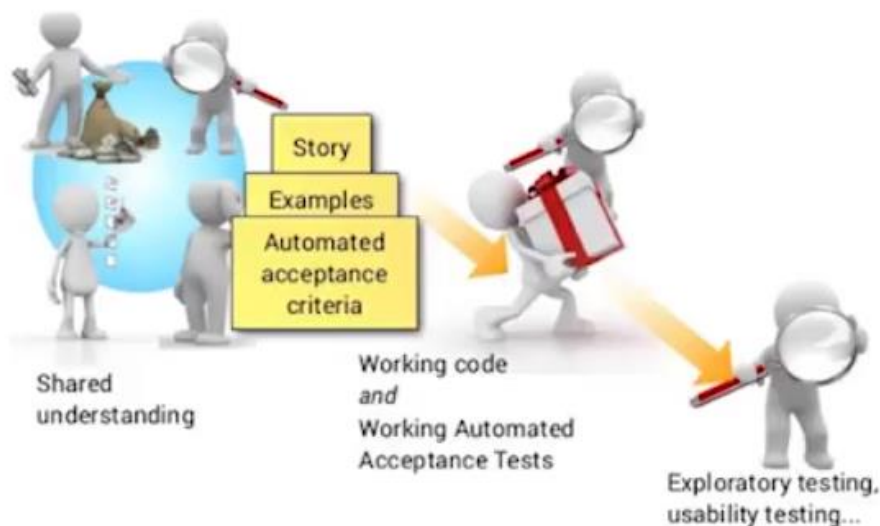


Рис. 5.2 Процесс BDD

- владелец продукта, разработчик, тестировщик **вместе** определяют примеры поведения требуемой системы;
- разработчик и тестировщик используют эти примеры для подготовки тестов (классическая TDD модель – **Красная** фаза);
- разработчик пишет код, полагаясь на разработанные тесты (классическая TDD модель – **Зеленая** и **Синяя** фаза);

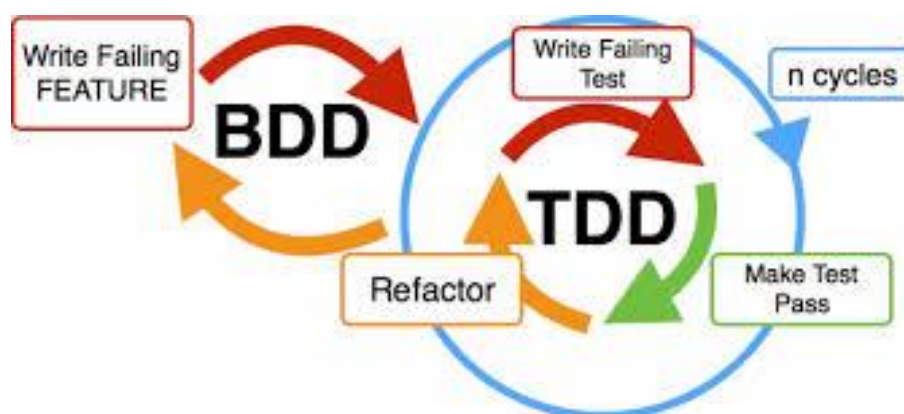


Рис 5.3 BDD в процессе разработки

Ключевой фактор, которого чаще всего не хватает, чтобы реализовать процесс BDD – недостаток взаимодействия владельца продукта, разработчика, тестировщика. Соответственно при применении данной методологии очень важно организовать взаимодействие всех 3-ех сторон. Иначе фактически это

ничем не будет отличаться от классического Agile процесса со всеми его недостатками. Из всего вышеперечисленного сделаем вывод о подходе BDD.

### **Преимущества BDD:**

- 1) ориентированность на язык бизнеса и устранения недопонимания между технической и бизнес стороной. Применимость BDD в реальных проектах;
- 2) самодостаточность BDD. Внедрение на всех этапах жизненного цикла ПО;
- 3) разработкой требований и спецификаций к требуемой функциональности занимаются и владелец продукта, и разработчик, и тестировщик, что устраняет фактор недопонимания или понимания требований по-своему с каждой стороны.
- 4) код покрыт тестами и верифицирован;
- 5) поощрение заранее хорошо спроектированной архитектуры и качестве интерфейсов

### **Недостатки BDD:**

- 1) необходимость в грамотном и высококвалифицированном управлении над проектом;
- 2) для организации процесса BDD в реальных условиях чаще всего приходится использовать специализированные инструменты и технологии (Например, SpecFlow, Cucumber);
- 3) эффективность BDD в большей степени только в продуктивном типе проектов;
- 4) неэффективность в маленьких и небольших проектах;

### **Вывод:**

На сегодняшний день методология BDD пользуется большой популярностью при разработке программных продуктов. Существует большое количество технологий и инструментов с большим сообществом пользователей, цель которых технически помочь реализовать идеи BDD. Также следует отметить, что эффективность BDD обусловлена наличием высококвалифицированного менеджмента и применимостью в большей степени на средних и больших проектах.

## 7 In action. Используемые технологии для написания тестов

Далее мы рассмотрим практическое применение модульного, интеграционного и функционального тестирования. Для этого необходимо выбрать стек технологий. Вся последующая изложенная информация носит сугубо практический характер, цель - рассмотреть на практике некоторые выше изложенные идеи и технологии, которые позволяют их реализовать.

- Язык программирования **Java 8**
- **JUnit 4.12** (<http://junit.org>) - это платформа тестирования с открытым исходным кодом для Java. Он был создан Кентом Бэком в 1997, и с тех пор это де-факто стандартное средство тестирования для разработчиков Java. Разработан для модульного тестирования, но также широко используется для других видов тестов.
- **Mockito 1.10.19** (<http://site.mockito.org>) – был разработан в 2007 г. (Szczepan Faber) и быстро созрел в высококачественный продукт. Он предлагает полный контроль над процессом имитации объектов, и позволяет писать тесты с помощью чистого и простого API.
- **PowerMock 1.6.6** (<http://powermock.github.io>) – был разработан в 2008 г. и выполняет те же задачи, что и Mockito, однако в отличие от последнего имеет возможность имитировать final классы языка Java и статические члены и др. Удобно использовать в связке с Mockito.
- **Spring framework 4.0.0** (<https://spring.io>) - универсальный фреймворк с открытым исходным кодом для Java (также есть поддержка для .NET). В данной работе наибольший интерес представляет модуль **Spring Test**.
- **Angular 4 framework** (<https://angular.io>) – платформа для разработки front-end части веб-приложений с открытым исходным кодом на основе языка TypeScript, разработанная Angular Team в Google.
- **Protractor 5** (<http://www.protractortest.org>) - представляет платформу для написания функциональных (end-to-end) тестов для приложений на Angular и AngularJS. Запускает тесты в приложении, работающем в реальном браузере, взаимодействуя с ним как пользователь.
- Система сборки **Maven 3** – система сборки проектов на языке Java. Довольно часто тесты (в особенности модульные) включаются в процесс сборки проектов, что обеспечивает автоматизацию при сборке приложений (кроме Maven, также используются Gradle, Ant и другие).

## 8 Модульное тестирование

Для того, чтобы оценить на практике методологии, в основу которых заложена идея разработки через тестирование (Test first approach) модульное тестирование отлично подходит для первого взгляда. Для демонстрации взяты наиболее популярные ситуации, встречающиеся в разработке небольших приложений. Все последующие примеры и даже дополнительные вместе с отсутствующими исходниками и необходимыми компонентами можно найти в репозитории <https://github.com/Strelts0v/testing-in-development>.

### 8.1 Тестирование бизнес-логики

Рассмотрим тестирование бизнес-логики на примере криптографического модуля используя JUnit. Итак, первый пример – мы разрабатываем криптографический модуль, который поддерживает 2 простые операции: 1) шифровки и 2) дешифровки. В соответствии с этой задачей реализован интерфейс нашего модуля – это 1-ый этап. Ничего сложного, но это именно то, что нужно для обзора подхода TDD и практики написания модульных тестов с использованием JUnit. Выполним следующие этапы:

#### 1) Разработка архитектуры криптографического модуля

Листинг 8.1 Внешний интерфейс криптографического модуля

```
/**
 * specifies contract for cryptography algorithm
 */
public interface Cryptographer {

    /**
     * encrypts input string
     * @param str - input string
     * @return encrypt string value
     */
    String encrypt (String str);

    /**
     * decrypts input string
     * @param str - input string
     * @return decrypt string value
     */
    String decrypt (String str);
}
```

#### 2) Создание сигнатуры конкретной реализации интерфейса

Реализуем разработанный интерфейс на конкретном примере – с использованием хог-шифрования. Пока у нас готова только сигнатура методов, их реализация последует после написания первых тестов.



## Листинг 8.2 Сигнатура конкретной реализации интерфейса

```
/**
 * implements Cryptographer interface according XOR crypt algorithm
 */
public class CryptographerXor implements Cryptographer {

    @Override
    public String encrypt (String str) {
        return null;
    }

    @Override
    public String decrypt (String str) {
        return null;
    }
}
```

### 3) Написание тестов (Красная фаза TDD)

Далее создаём тестовый класс. Композиционно включаем в него используемые криптографический модуль. Сам тест начинается с аннотации @Test (из JUnit), после которой следует метод с логикой тестирования. Основные моменты:

- Название метода-теста должно описывать основные действия и особенности.
- Должны чётко идентифицировать правильное и неправильное выполнение теста (Используем метод assertEquals с ожидаемым и получаемым результатом)
- Если тест упал (рис 8.1), то мы выдаём сообщение о причине.

Как видим 1-ый тест провалился (Красная фаза TDD) – значит переходим к доработке кода тестируемого класса (Синяя фаза TDD).

## Листинг 8.3 Написание первого теста

```
public class CryptographerXorTest {

    private final Cryptographer cryptographer = new CryptographerXor();

    @Test
    public void encryptAndDecryptWithShortStrTest() throws Exception {
        final String testStr = "44447777";
        final String encryptStr = cryptographer.encrypt(testStr);
        final String errorMessage =
            "Values of original and decrypt strings are different";

        Assert.assertEquals(errorMessage, testStr,
            cryptographer.decrypt(encryptStr));
    }
}
```

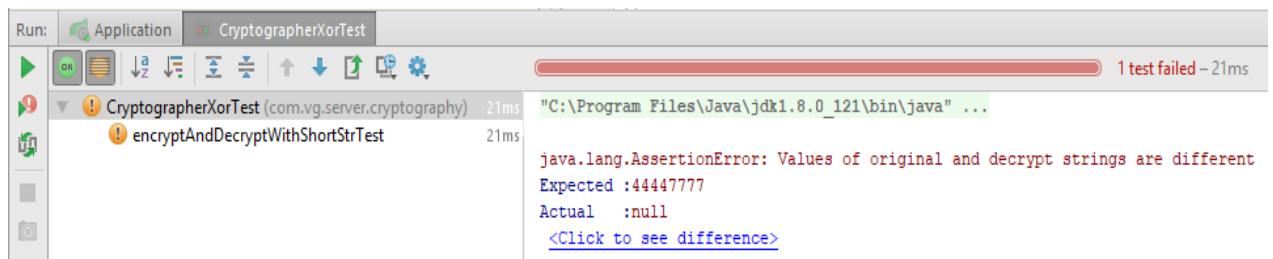


Рис 8.1 Неудачное выполнение теста в IDE IntelliJ IDEA

#### 4) Доработка методов тестируемого класса (Синяя фаза TDD)

Дорабатываем методы криптографического класса для успешной сдачи теста (Зеленая фаза TDD) Убедимся, что тесты выполняются корректно (Рис. 8.2) и перейдём к очередному циклу TDD.

#### Листинг 8.4. Доработка методов тестируемого класса

```
/** property - crypto key */
private static final String CRYPTO_KEY = "qwerty";

@Override
public String encrypt(String str) {
    byte[] key = CRYPTO_KEY.getBytes();
    byte[] text = str.getBytes();
    byte[] result = new byte[str.length()];
    for (int i = 0; i < text.length; i++) {
        result[i] = (byte) (text[i] ^ key[i % key.length]);
    }
    return new String(result);
}

@Override
public String decrypt(String str) {
    return encrypt(str);
}
```

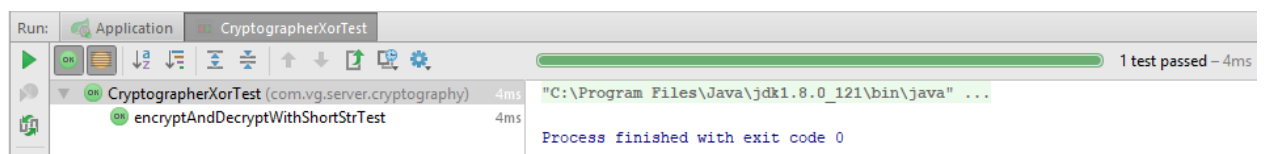


Рис. 8.2 Успешное выполнение теста в IntelliJ IDEA

#### 5) Следующий цикл TDD. Больше тестов и доработка кода

Пишем дополнительные тесты и следим за их выполнением, каждый невыполненный тест – это очередная доработка методов тестируемого класса. Конечно этот процесс можно повторять бесконечно, но стоит ограничиваемся крайними диапазонами входных данных, этого вполне достаточно для верификации работы разработанного класса.

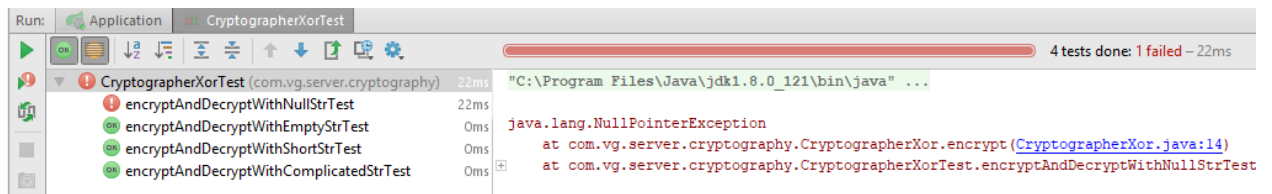


Рис 8.3 Следующие циклы TDD, написание дополнительных тестов

Ещё один важный нюанс при написании тестов. При его написании, следует не только руководствоваться тем, чтобы всё работало, как это ни странно, но и тем, чтобы найти потенциальные ошибки при тех или иных входных данных, которые могут поступить на вход криптографического модуля. При этом код при возникновении проблем, может выбросить исключение на более высокий уровень, где возможно обработать ситуацию по другому сценарию.

## 8.2 Тестирование бизнес-логики, использующей сторонние интерфейсы

Теперь рассмотрим более специфическую ситуацию для модульного тестирования на примере отправки email почты с помощью Mockito и PowerMock. Имеем класс GmailEmailService, реализующий интерфейс EmailService. Для отправки сообщения на email адрес, GmailEmailService использует javax.mail API, а именно статический метод send() класса javax.mail.Transport. В данном случае, не нужно выполнять регрессирующее тестирование для используемых внешних API. Для написания модульного теста необходимо имитировать поведение данного статического члена.

### Листинг 8.5. Тестирование сервиса отправки email

```
// set PowerMock test runner instead of JUnit's one
@RunWith(PowerMockRunner.class)
// prepare class for executing tests
@PrepareForTest({Transport.class})
public class GmailEmailServiceTest {

    private EmailService service;

    private static final int EXPECTED_INVOCATION_TIMES = 1;

    // before each test set new email service object
    @Before
    public void setUpEmailService() {
        service = new GmailEmailService();
    }

    @Test(expected = EmailSendingException.class)
    public void sendEmailMessageWithInvalidAddressTest() throws Exception {
        // mock static class Transport
        mockStatic(Transport.class);
        // do nothing when static methods of class Transport are invoked
        doNothing().when(Transport.class);
        // execute sending email
    }
}
```

```

        service.sendMessage(anyString(), anyString(),
            anyString(), anyString());
        // verify that static method of class Transport was invoked
        verifyStatic(times(EXPECTED_INVOCATION_TIMES));
    }

    @Test
    public void sendEmailMessageWithValidAddressTest() throws Exception {
        final String toEmail = "to@gmail.com";
        final String fromEmail = "from@gmail.com";
        final String subject = "test subject";
        final String body = "test body";

        mockStatic(Transport.class);
        doNothing().when(Transport.class);

        service.sendMessage(toEmail, fromEmail, subject, body);

        verifyStatic(times(EXPECTED_INVOCATION_TIMES));
    }
}

```

### 8.3 Тестирование REST контроллера в модели MVC

Так как на практике часто необходимо использовать такие тяжеловесные фреймворки и платформы, как, к примеру, Spring Framework. Рассмотрим и такую ситуацию для тестирования.

#### Листинг 8.6. Тестирование REST контроллера в Spring Framework

```

// set Spring test runner instead of JUnit
@RunWith(SpringRunner.class)
// specify that our application is based on Spring Boot
@SpringBootTest
// set auto configuration for used classes
@AutoConfigureMockMvc
public class HeroControllerTest {
    // Bean for performing mock test in Spring Framework,
    // will be bind automatically
    @Autowired
    private MockMvc mockMvc;

    // specify constants for tests
    private static final String HERO_NAME_JSON_PATH = "$.name";
    private static final String HERO_ARRAY_SIZE_JSON_PATH = "$";
    private static final String EMPTY_HERO_NAME = "none";
    private static final String HERO_ID_PARAM = "id";
    private static final String HEROES_COUNT_PARAM = "count";
    private static final String HERO_NAME_PARAM = "name";

    private static final int DEFAULT_HEROES_LIMITED_COUNT = 10;
    private static final int MAX_HEROES_LIMITED_COUNT = 16;

    @Test
    public void paramHeroShouldReturnExistHero() throws Exception {
        final String urlTemplate = "/api/hero";
        final String idParam = "11";
        final String expectedHeroName = "Dynama";
        // execute GET request with param id
        this.mockMvc.perform(get(urlTemplate).param(HERO_ID_PARAM, idParam))
    }
}

```

```

        .andDo(print())
        .andExpect(status().isOk())
        .andExpect(jsonPath(HERO_NAME_JSON_PATH)
            .value(expectedHeroName));
    }

    @Test
    public void paramHeroesShouldReturnCorrespondHeroesCount()
        throws Exception {
        final String urlTemplate = "/api/heroes";
        final String countParam = "7";
        final int expectedHeroCount = 7;

        this.mockMvc.perform(get(urlTemplate)
            .param(HEROES_COUNT_PARAM, countParam)
            .andExpect(status().isOk())
            .andExpect(jsonPath(HERO_ARRAY_SIZE_JSON_PATH,
                hasSize(expectedHeroCount))));
    }

    @Test
    public void addHeroShouldReturnCorrespondHeroWithCorrespondId()
        throws Exception {
        final String urlTemplate = "/api/add/hero";
        final String expectedName = "testName";
        final int expectedId = 17;

        String heroNameJson =
            "{ " + HERO_NAME_PARAM + " : \"" + expectedName + "\" }";
        // execute POST request to insert new Hero
        this.mockMvc.perform(post(urlTemplate)
            .content(heroNameJson)
            // check expected values
            .andExpect(status().isOk())
            .andExpect(jsonPath(HERO_NAME_JSON_PATH).value(expectedName))
            .andExpect(jsonPath(HERO_ID_PARAM).value(expectedId)));
        // and after that delete inserted Hero
        final String deleteUrlTemplate = "/api/delete/hero";
        final String addedHeroId = "17";
        this.mockMvc.perform(delete(deleteUrlTemplate)
            .param(HERO_ID_PARAM, addedHeroId)
            .andExpect(status().isOk()));
    }
}

```

Как видно из рассмотренных выше ситуаций, существует большое количество средств для того, чтобы реализовывать модульное тестирование даже в самых необычных и сложных ситуациях. Важно только к определенной ситуации применить нужный инструмент.

## 9 Интеграционное тестирование

В модульном тестировании мы тестировали класс для отправки email почты, однако мы протестировали только правильное выполнение самой логики, но не проверили действительно ли отправленное письмо приходит на почту. Рассмотрим самый простейший способ интеграционного теста.

### 9.1 Тестирование отправки email почты

Сразу нужно отметить, что представленные ниже тесты не являются автоматизированными и не включаются в процесс сборки проекта (тестовый класс помечается аннотацией `@Ignore`). Тест из листинга 9.1 ручной, однако у него есть одно большое преимущество. Как правило, разработчику нужно после написания подобного функционала сразу убедиться, что он действительно работает. В этой ситуации нужно совсем немного времени, чтобы верифицировать работу разработанного функционала написав ручной тест.

Листинг 9.1 Ручной интеграционный тест отправки email почты

```
@Ignore
public class GmailEmailServiceManualTest {

    private EmailService service;

    @Before
    public void setUp() throws Exception {
        service = new GmailEmailService();
    }

    @Test
    public void sendEmailMessageToReadEmailAddressTest() throws Exception {
        final String toEmail = "real.email.address@gmail.com";
        final String fromEmail = " real.email.address@gmail.com ";
        final String subject = "test subject";
        final String body = "test body";

        service.sendMessage(fromEmail, toEmail, subject, body);
    }
}
```

Тест очень простой, однако польза от него разработчику весьма существенна на начальном этапе разработки описанной функциональности. Для того, чтобы разработать автоматизированный интеграционный тест необходимо намного больше времени и сторонних средств, поэтому в небольших приложениях такой вид тестирования часто реализуется ручным способом или же только для некоторых компонентов или частей приложения.

## 9.2 Тестирования слоя доступа к данным из базы данных

Рассмотрим ситуацию доступа к данным из базы данных с помощью слоя DAO (enterprise design pattern «Data Access Object»). Один из нюансов такого тестирования заключается в том, что нельзя затрагивать реальные данные, которые хранятся в уже работающем приложении (Если мы говорим о ситуации, когда приложение уже вышло в релиз). В таком случае, удобнее всего иметь тестовую копию реальной базы данных, заполнить её некоторыми тестовыми данными и уже с ней проводить всё тестирование. В данном случае всё приложение является тестовым, поэтому такой сложности не возникает. Другой нюанс заключается в том, что состояние даже тестовой базы данных должно оставаться одним и тем же после каждого теста. Это можно реализовать с помощью специально разработанных инструментов, например, DBUnit (такой вариант является более предпочтительным) или же можно после каждого проводимого изменения базы данных возвращать её в исходное состояние удаляя внесенные изменения или заново инициализировать базу данных. В данном случае, после каждого внесенного изменения, это изменение удаляется из базы данных. Также для тестовых целей удобно использовать in-memory базу данных H2. В рассмотренной базе данных хранятся герои, которые имеют 2 параметра: уникальный идентификатор и имя.

Листинг 9.2 Тестирование слоя доступа к данным, написанного с помощью Spring Data

```
// set Spring test runner instead of JUnit
@RunWith(SpringRunner.class)
// specify that our application is based on Spring Boot
@SpringBootTest
// set auto configuration for used classes
@AutoConfigureMockMvc
public class HsqldbHeroServiceTest {

    @Autowired
    private HeroService service;

    @Test
    public void findHeroByIdTShouldReturnHeroWithCorrespondNameTest()
        throws Exception {
        final long heroId = 2L;
        final String expectedHeroName = "Superman";
        final String errorMessage =
            "Expected hero name and actual are different";

        Hero actualHero = service.findHeroById(heroId);
        Assert.assertEquals(errorMessage,
            expectedHeroName, actualHero.getName());
    }

    @Test
```

```

public void getHeroesLimitedShouldReturnCorrespondHeroListSizeTest()
    throws Exception {
    final int expectedHeroListSize = 5;
    final String errorMessage =
        "Expected hero list size and actual are different";

    List<Hero> actualHeroList = service
        .getHeroesLimited(expectedHeroListSize);
    Assert.assertEquals(errorMessage,
        expectedHeroListSize, actualHeroList.size());
}

@Test
public void addHeroShouldReturnHeroWithNonZeroIdTest()
    throws Exception {
    final String heroName = "Test";
    Hero hero = new Hero();
    hero.setName(heroName);

    final long notExpectedHeroId = 0L;
    final String errorMessage =
        "Expected not 0 generated id in added hero";
    // expect that generated hero id is not 0
    Hero addedHero = service.addHero(hero);
    Assert.assertNotEquals(errorMessage,
        notExpectedHeroId, addedHero.getId());
    // after successful test delete added hero
    service.deleteHero(addedHero.getId());
}

@Test
public void updateHeroShouldCorrespondCountOfUpdatedRowsTest()
    throws Exception {
    final String heroName = "Test";
    final String updatedHeroName = "Updated Test";
    Hero hero = new Hero();
    hero.setName(heroName);
    // expect that generated hero id is not 0
    Hero addedHero = service.addHero(hero);
    addedHero.setName(updatedHeroName);
    int updatedRows = service
        .updateHero(addedHero.getId(), addedHero);

    final int expectedUpdatedRows = 1;
    final String errorMessage =
        "Expected " + expectedUpdatedRows +
        " generated id in added hero";
    Assert.assertEquals(errorMessage,
        expectedUpdatedRows, updatedRows);
    // after successful test delete added hero
    service.deleteHero(addedHero.getId());
}
}

```

В заключении к интеграционному тестированию, следует отметить, что кроме рассмотренных средств, инструментов и библиотек разработано и огромное количество других для самых разные ситуаций. Некоторые из них для языка Java: Selenium, Arquillian, JTest, Grinder, TestNG, JWalk, EasyMock, Harmcrest и другие.



## 10 Функциональное тестирование

Если предыдущие виды тестов были больше для команды разработчиков, то данный вид тестирования, позволяет сравнить результат работы приложения с требованиями заказчиков. В средних и больших проектах без данного вида тестирования очень сложно обойтись, если речь идёт о верификации. Для выполнения функциональных (или end-to-end тестов) воспользуемся тестовым фреймворком Protractor. Для того чтобы установить его, в корне папки client (клиент разработанного приложения), запустим консоль и выполним команду:

```
npm install -g protractor
```

После её выполнения Protractor будет установлен на клиенте, также будет установлен webdriver-manager, который позволит легко сконфигурировать и запустить Selenium сервер. Следующая команда загрузит все необходимые бинарные файлы для Selenium:

```
webdriver-manager update
```

Теперь чтобы запустить Selenium сервер, выполним команду:

```
webdriver-manager start
```

Чтобы посмотреть статус сервера необходимо зайти в браузере по адресу <http://localhost:4444/wd/hub>. Далее приведём простейший пример теста, который будет проверить атрибут title веб-страницы (Листинг 10.1).

Листинг 10.1 Пример end-to-end теста (файл spec.js)

```
describe('Protractor Demo App', function() {  
  it('should have a title', function() {  
    browser.get('http://localhost:4200/test ');  
    expect(browser.getTitle()).toEqual('Test title');  
  });  
});
```

Далее сконфигурируем запуск теста.

Листинг 10.2 Создание конфигурационного файла для запуска тестов (файл conf.js)

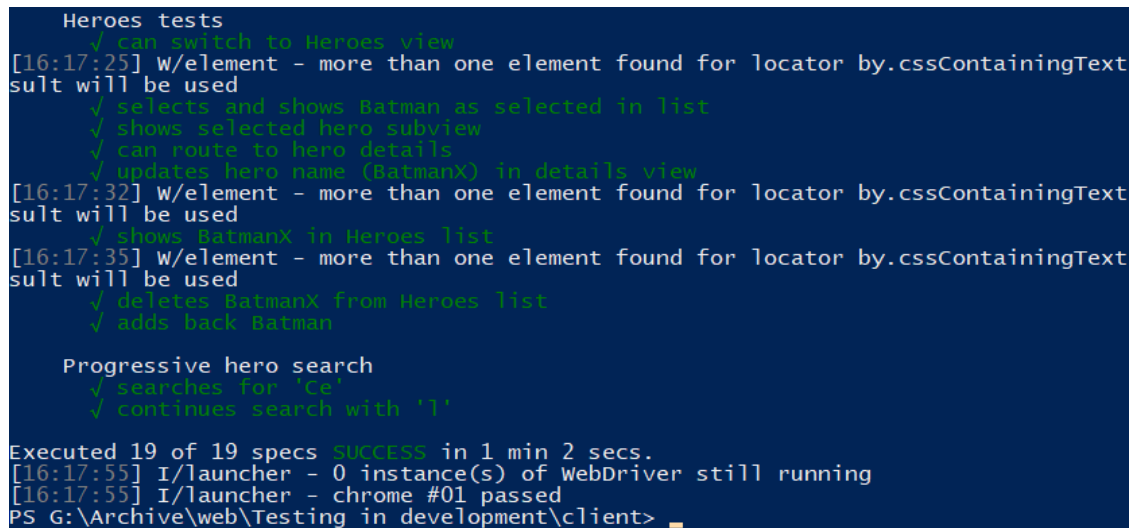
```
exports.config = {  
  framework: 'jasmine',  
  seleniumAddress: 'http://localhost:4444/wd/hub',
```

```
specs: ['spec.js']  
}
```

Всё, что нужно для запуска теста, готово. Выполним команду в корне клиентского приложения для запуска:

*protractor conf.js*

В случае разработанного тестового проекта (все исходники с тестами хранятся в репозитории по адресу <https://github.com/Strelts0v/testing-in-development>), файл с тестами лежит по пути `e2e/app.e2e-spec.ts`, а файл с конфигурациями для удобства назван `protractor.conf.js`. Запустим клиентское и серверное приложение тестового проекта. После чего выполним выше изложенный алгоритм для запуска теста. Если все шаги были выполнены верно, то в консоли будет успешное выполнение всех тестов:



```
Heroes tests  
  ✓ can switch to Heroes view  
[16:17:25] W/element - more than one element found for locator by.cssContainingText  
sult will be used  
  ✓ selects and shows Batman as selected in list  
  ✓ shows selected hero subview  
  ✓ can route to hero details  
  ✓ updates hero name (BatmanX) in details view  
[16:17:32] W/element - more than one element found for locator by.cssContainingText  
sult will be used  
  ✓ shows BatmanX in Heroes list  
[16:17:35] W/element - more than one element found for locator by.cssContainingText  
sult will be used  
  ✓ deletes BatmanX from Heroes list  
  ✓ adds back Batman  
  
Progressive hero search  
  ✓ searches for 'Ce'  
  ✓ continues search with 'l'  
  
Executed 19 of 19 specs SUCCESS in 1 min 2 secs.  
[16:17:55] I/launcher - 0 instance(s) of WebDriver still running  
[16:17:55] I/launcher - chrome #01 passed  
PS G:\Archive\web\Testing in development\client>
```

Рис 10.1 Успешное выполнение тестов в Protractor

Как вывод отметим, что, как и интеграционные, функциональные (или end-to-end) тесты требовательны к наличию различных средств автоматизации и внешних инструментов, что обусловлено их сложностью. В данном случае клиентское приложение разработано с помощью фреймворка Angular 4, поэтому в качестве тестовой платформы был использован Protractor. Соответственно, то на чем в итоге разрабатывается приложение, в частности его клиентская часть, играет ключевую роль в выборе технологий для осуществления функционального тестирования.

## 11 Заключение

После рассмотрения основных видов тестирования, обзора существующих методологий разработки программного обеспечения для реализации идеи разработки через тестирование и практической работы с различными видами тестов в некоторых ситуациях, встречающихся в разработке приложений выделим следующие основные моменты.

- 1) В зависимости от сложности разрабатываемого ПО, не все виды тестов использовать целесообразно.
- 2) Гибкие методологии разработки через тестирование действительно помогают в написании простых и понятных интерфейсов и качественного, хорошо спроектированного кода. Однако и у данных методологий есть несколько недостатков – требовательность к высококвалифицированному менеджменту проекта, использование ряда непростых технологий для построения процесса разработки, организация регулярного взаимодействия между заказчиками, разработчиками и тестировщиками.
- 3) Каждый вид тестирования нужен для определенного круга задач. Краеугольный вопрос – это автоматизация. И здесь хорошо применимо для необходимого объема автоматизации правило Парето – 80 / 20. Суть для рассматриваемого случая: на практике, только 20% автоматизированных тестов приносят 80% реальной пользы.

На данный момент, для того, чтобы продвинуться в данной работе дальше, нужно еще больше практического опыта применения всех рассмотренных идей. Цель на будущее – анализ и обобщение нового накопленного опыта и новых знаний, которые позже возможно трансформировать в доработку существующих методологий или же создание новых, снова-таки чтобы разрабатывать приложения быстрее, качественнее и дешевле.

**Ссылка на предоставленные материалы:**

<https://github.com/Strelts0v/testing-in-development>

## **12 Используемые источники**

Список использованных источников:

1. Tomek Kaczanowski Practical Unit Testing with Junit and Mockito
2. Савин Р. Г. Тестирование dot com
3. Petar Tachiev, Felipe Leme, Vincent Massol, Gary Gregory Junit in action second edition
4. Exelixis Media P. C. Mockito Programming Cookbook
5. «iTech forum 2017» Behavior-driven development