Тестирование программ

> Любой долгосрочный проект без надлежащего покрытия тестами обречен, рано или поздно, быть переписанным с нуля

Тести́рование програ́ммного обеспе́че́ния — процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определённым образом.

Тестирование ПО вообще классифицируется на несколько категорий, но мы сегодня рассмотрим только одну из них, деление по степени изолированности:

- **Компонентное или модульное тестирование** - изолированное тестирование функций каждого отдельного компонента системы.
- **Интеграционное тестирование** - проверка "соединения" отдельных компонентов системы. На этом уровне компоненты тестируются уже в целом и проверяется правильность взаимодействия между ними.
- **Приемочное тестирование** - проверка работоспособности выполняемая на уровне цельной, интегрированной, системы, для установления соответствия бизнес требованиям.
- **Ручные тесты** - ручная проверка тех частей системы, которые не возможно автоматизировать.

Кроме этих видов, также существуют другие виды тестирования, которые могут быть автоматизированными, но они проверяют другие аспекты приложения, помимо основного функционала, например:

- Тестирование удобства использования
- Тестирование пакета установки
- Тестирование (репетиция) поставки
- Нагрузочное тестирование (Load testing, performance testing)
- Тестирование безопасности системы

- и т.п.

Подход TDD: Test Driven Development.

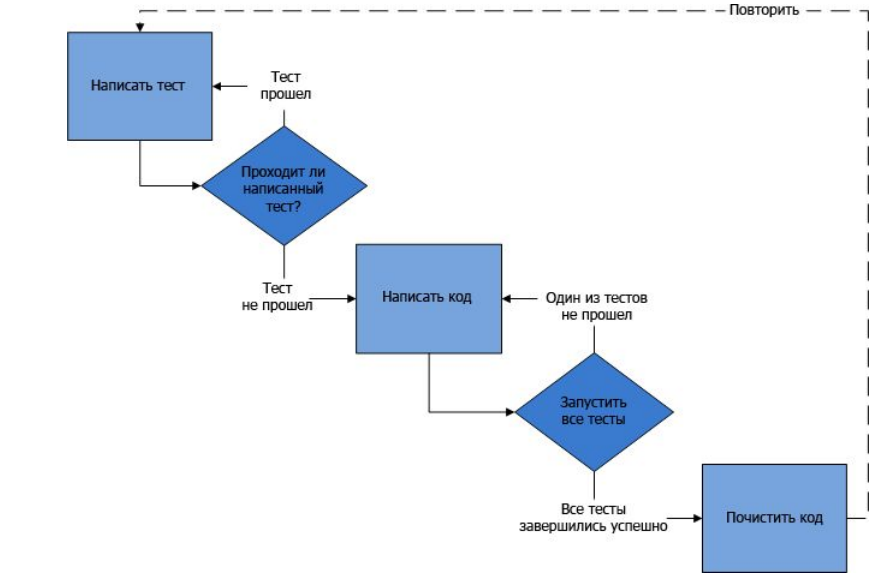
TDD или же разработка через тестирование - это техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, проверяющий желаемое изменение, затем создаётся код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.

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.

Тесты должны писаться для тестируемой функциональности. Это помогает убедиться, что приложение пригодно для тестирования, поскольку разработчику придется с самого начала обдумать то, как приложение будет тестироваться. А также способствует тому, что тестами будет покрыта вся функциональность. Когда функциональность пишется до тестов, разработчики склонны переходить к реализации следующей части системы, не протестировав существующую.

Разработка через тестирование предлагает больше, чем просто проверку корректности, она также влияет на дизайн программы. Изначально сфокусировавшись на тестах, проще представить, какая функциональность необходима пользователю. Тесты заставляют делать свой код более приспособленным для тестирования. Например, отказываться от глобальных переменных, одиночек (singletons), делать классы менее связанными и легкими для использования. Сильно связанный код или код, который требует сложной инициализации, будет значительно труднее протестировать. Модульное тестирование способствует формированию четких и небольших интерфейсов, выполняющих определенную, как правило небольшую, роль.

Цикл разработки через тестирование



При разработке какой либо части программы, цикл разработки состоит из пяти этапов.

1. **Добавление теста.** При разработке через тестирование, добавление каждой новой функциональности в программу начинается с написания теста. Это фокусирует разработчика на определении входных требований, до того как он начнет писать код. Для этого рассматриваются возможные сценарии использования и пользовательские истории.
2. **Запуск всех тестов**. На этом этапе разработчик запускает весь набор тестов, что бы убедиться, что только что написанные тесты не проходят. Новые тесты не должны проходить проверку, так как ещё нет кода, который они тестируют. Если новый тест проходит проверку на этом этапе, это означает что он написан неправильно.
3. **Разработка кода тестируемой функциональности**. На этом этапе пишется новый код так, что тест будет проходить. Этот код не обязательно должен быть идеален, и допустимо, чтобы он проходил тест каким-то не элегантным способом. Это приемлемо, поскольку последующие этапы улучшат и отполируют его. Важно писать код, предназначенный именно для прохождения теста. Не следует добавлять лишней и, соответственно, не тестируемой функциональности.
4. **Запуск тестов**. Если все тесты проходят, программист может быть уверен, что код удовлетворяет всем тестируемым требованиям
5. **Рефакторинг или улучшение качества кода**. Когда достигнута требуемая функциональность, на этом этапе код может быть почищен. После внесения изменений в код, обязательно необходимо повторить этап запуска тестов, что бы убедиться, что внесенные изменения не нарушили логику работы функциональности. *Рефакторинг* — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы, устранить дублирование кода, облегчить внесение изменений в ближайшем будущем.

Идея проверять, что вновь написанный тест не проходит, помогает убедиться, что тест реально что-то проверяет. Только после этой проверки следует приступать к реализации новой функциональности. Этот приём, известный как «красный/зелёный/рефакторинг», называют «мантрой разработки через тестирование». Под красным здесь понимают не прошедшие тесты, а под зелёным — прошедшие.

Unit Testing: Модульное тестирование

Unit testing (юнит тестирование или модульное тестирование) — заключается в изолированной проверке каждого отдельного элемента путем запуска тестов в искусственной среде. Для этого необходимо использовать драйверы и заглушки. Поэлементное тестирование — первейшая возможность реализовать исходный код. Оценивая каждый элемент изолированно и подтверждая корректность его работы, точно установить проблему значительно проще чем, если бы элемент был частью системы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то

есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Не всякий класс легко покрыть unit тестами. При проектировании нужно учитывать возможность тестируемости и зависимости класса делать явными. Чтобы гарантировать тестируемость можно применять TDD методологию, которая предписывает сначала писать тест, а потом код реализации тестируемого метода. Тогда архитектура получается тестируемой. Распутывание зависимостей можно осуществить с помощью Dependency Injection. Тогда каждой зависимости явно сопоставляется интерфейс и явно определяется как эта зависимость встраивается в программу — в конструктор, в свойство или в метод.

Ваши тесты обязательно должны быть добавлены в контроль версий. Это позволит отслеживать изменения и возвращаться к предыдущим версиям при необходимости.

Для чего нужны модульные тесты

- Изолируют каждую часть программы и проверяют её корректность. Помогают рано обнаруживать проблемы.
- Позволяют улучшать архитектуру приложения с помощью определения ответственности модулей.
- Заставляют разработчиков мыслить в рамках входных, выходных и ошибочных условий.
- Заставляют отделять интерфейс от реализации.
- Доказывают, что код модуля работает так, как ожидалось (хотя бы математически).

Что имеет смысл тестировать.

Тесты, обычно, разрабатываются в следующих случаях:

- Для неделимых изолированных частей системы: открытых функций, методов, или классов. Их проверка обеспечивает уверенность в том, что код написан согласно требованиям по входным данным, и главное, что они дают ***ожидаемый*** результат.
- Для сложного кода без зависимостей. Это некие алгоритмы или бизнес-логика. То есть особенно важные части системы, на которых строится вся система.
- Для не очень сложного кода с зависимостями. Это такой код, который связывает между собой разные компоненты. Тесты важны, чтобы уточнить, как именно должно происходить взаимодействие между этими компонентами.
- При исправлении багов. Сначала описывается тест который воспроизводит баг, и в котором указывается требуемые, корректные, выходные условия. Затем вносятся изменения в код, и тест теперь должен пройти проверку. Если вы не проводите модульное тестирование, предлагаю заняться этим после возникновения следующего большого бага. Проверьте, с каким методом он будет связан, напишите сбойный тест с правильными аргументами и результатом, исправьте баг, снова запустите модульный тест. Если он будет

пройден, то можете быть уверены, что этот баг пришлось исправлять в последний раз (с учётом ваших определённых входных сценариев).

Такой подход помогает легче понять модульное тестирование. Проанализируйте отдельно каждый метод. Поставщики данных могут помочь определить входные и выходные данные для любых сценариев, которые могут прийти вам в голову, поэтому что бы ни произошло, вы будете знать, чего ожидать.

Что не покрывают модульными тестами.

- Функциональность за пределами контекста модулей. Тест должен покрывать только одну функцию.
- Интеграция модулей с другими модулями. Этим занимается другая категория тестов.
- Любое неизолированное поведение (неимитируемые зависимости, настоящие БД, сеть), внешние библиотеки, фреймворки, Если мы не можем с имитировать нечто, то нам будет очень сложно это протестировать, и перенести в другую среду.
- Приватные, защищённые, статичные методы.

Характеристики хороших модульных тестов

Правильно спроектированные и написанные тесты обладают рядом полезных характеристик:

- Они достоверные. Тест является спецификацией метода класса. Контрактом, описывающим какие входные параметры ожидает этот метод, и что остальные компоненты системы ждут от него на выходе.
- Могут запускаться в любом порядке, вне зависимости от других тестов. Это позволит запускать тесты параллельно для ускорения тестирования.
- Не влияют на результат других тестов.
- Запускаются только в памяти (никаких взаимодействий с БД, чтений/записей в файловой системе). Не зависят от окружения, на котором они выполняются. Полностью управляют своими зависимостями.
- Быстрые и автоматизированные. Не требуют сложной настройки системы для запуска. Запускаются регулярно в автоматическом режиме.
- Имеют чётко определённую ЕДИНСТВЕННУЮ ЗАДАЧУ. Удобны для чтения и сопровождения.
- Соблюдают единую конвенцию именования. Хорошо именованы. Имя теста четко говорит о том, что проверяется внутри.
- Каждый тестирующий класс проверяет только одну сущность (один класс, метод, функцию).

- Их легко разрабатывать. :)

Несоблюдение этих правил приведёт к тому, что тесты будут медленными, будут требовать подключения к базе или другие ресурсы, их будет сложно разрабатывать, а с увеличением их количества у разработчиков будет всё меньше и меньше желания их поддерживать, разрабатывать, запускать.

Практическая часть: JUnit 5

JUnit - является одним из популярных фреймворков помогающим создавать модульные и интеграционные тесты.

Именование тестовых классов и методов

У вас в проекте есть бизнес логика для формирования данных для отчетов в классе ReportBuilder ? Тогда хорошим выбором имени класса с тестами будет ReportBuilderTest.

Так же важен правильный выбор имён методов тестирующих тот или иной функционал.

testReportBuilding - это не очень хороший выбор имени. Что именно тестируется? Каковы входные параметры? Что ожидаем на выходе? Могут ли возникать ошибки и исключительные ситуации? Тестируем ли мы инварианты? и т.д.

Одним из хороших шаблонов имени теста является:

```
[тестируемый метод]    [сценарий]     [ожидаемое поведение]
```

Например:

```
@Test
void makeReport   with_data      expect_BadDataException() {
  //...
}
```

Прочитав имя этого метода мы уже представляем, что у нас на входе, а что на выходе. Что мы передаём некоторые данные (вообще, по возможности, желательно указывать сами данные), и что мы ожидаем на выходе исключение. Это спецификация к вашему коду и документация к поведению тестируемого объекта. Аннотация `@Test` указывает jUnit, что это тестовый метод, который надо выполнить.

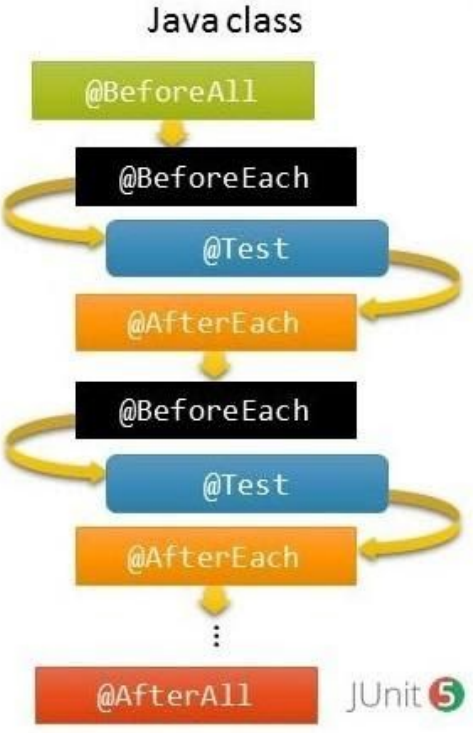
При написании тестов рекомендуется использовать единый стиль кода. Одним из рекомендуемых подходов является AAA (Arrange/Allocate, Act, Assert) или "создай, выполни, проверь результат".

```
// arrange
var builder = new ReportBuilder();

// act
var result = builder.makeReport(data);

// assert
Assertions.assertEquals("Header", result.getHeader());
```

Жизненный цикл тестов



Все тесты внутри JUnit проходят один и тот же жизненный цикл.

1. Сначала создаётся экземпляр тестового класса
2. Выполняется статический метод отмеченный аннотацией @BeforeAll. Это метод в котором можно выполнить какие-то действия общие для всех тестов в этом наборе. Например создаются какие-то общие тестовые данные, коллекции объектов, подключения к реальным или дублирующим сервисам.
3. Теперь для каждого метода отмеченного аннотацией @Test. выполняется следующий набор действий
   a) Сначала отрабатывает метод отмеченный аннотацией @BeforeEach. Это метод выполняющий так называемый SetUp, то есть подготовка входных данных. Здесь

обычно проводят чистку дублей или сервисов от данных, приводят в начальное значение входные данные и прочее. Здесь так же можно вести лог выполнения тестов. Например, записывать время начала работы каждого теста.

b) Теперь выполняется сам метод с аннотацией @Test

c) После успешного или не успешного завершения работы теста, отрабатывает метод с аннотацией @AfterEach. Это tearDown. Здесь мы тоже можем проводить очистку хранилищ от данных, проводить какие-то другие действия. Например, если мы записываем время работы теста в начале работы, то тут мы можем высчитать сколько было потрачено на выполнение.

4. После выполнения всех тестов в наборе, отрабатывает метод с аннотацией @AfterAll. Здесь обычно закрывают и уничтожают все открытые в @BeforeAll объекты. Но так же это место для каких-то других действий, которые надо выполнить после завершения всех тестов.

Запуск тестов через Maven

Если ваш проект использует Maven в качестве системы сборки, то запуск тестов производится одной командой.

```
mvn test
```

Дальнейшее чтение

- https://ru.wikipedia.org/wiki/Регрессионное_тестирование
- https://ru.wikipedia.org/wiki/Модульное_тестирование
- https://ru.wikipedia.org/wiki/Mock-объект
- https://ru.wikipedia.org/wiki/Тестирование_белого_ящика
- https://ru.wikipedia.org/wiki/Тестирование_чёрного_ящика
- https://habr.com/en/company/mailru/blog/412695/
- https://ru.wikipedia.org/wiki/Разработка_через_тестирование
- https://en.wikipedia.org/wiki/Assertion_(computing)
- https://habr.com/ru/post/275249/
- https://habr.com/ru/post/92038/

- https://habr.com/ru/post/169381/

Разработка через контракты (утверждения о истинности)

- https://ru.wikipedia.org/wiki/Контрактное_программирование
- https://en.wikipedia.org/wiki/Design_by_contract
- http://www.valid4j.org/
- https://github.com/nhatminhle/cofoja
- http://oval.sourceforge.net/

Интеграция тестирования в Spring

- https://www.baeldung.com/spring-boot-testing
- https://www.baeldung.com/integration-testing-in-spring
- https://www.baeldung.com/injecting-mocks-in-spring
- https://www.baeldung.com/restclienttest-in-spring-boot
- https://spring.io/guides/gs/testing-web/
- https://reflectoring.io/unit-testing-spring-boot/