Модульное тестирование

# С практическими рекомендациями для Java

Оглавление

[С практическими рекомендациями для Java 1](#_Toc472076784)

[Что такое модульное тестирование 3](#_Toc472076785)

[Отступление 3](#_Toc472076786)

[Назначение 3](#_Toc472076787)

[Терминология 3](#_Toc472076788)

[Роль модульного тестирования в общем процессе тестирования 5](#_Toc472076789)

[Связь модульного и интеграционного тестирования 5](#_Toc472076790)

[Связь модульного и регрессионного тестирования 5](#_Toc472076791)

[Модульное тестирование, как часть процесса разработки 7](#_Toc472076792)

[Рефакторинг 7](#_Toc472076793)

[Отладка 7](#_Toc472076794)

[Регрессионное тестирование 8](#_Toc472076795)

[Test Driven Development 8](#_Toc472076796)

[Тесты, как примеры кода 9](#_Toc472076797)

[Гибкие технологии разработки, Sbergile, DevOps 10](#_Toc472076798)

[Рекомендации по написанию тестов 12](#_Toc472076799)

[Требования к модульным тестам 12](#_Toc472076800)

[Структура тестового класса 12](#_Toc472076801)

[Объекты тестирования – все публичные методы модуля, сложные внутренние методы 13](#_Toc472076802)

[Какие методы нужно тестировать 13](#_Toc472076803)

[Какие методы не нужно тестировать 14](#_Toc472076804)

[Проектирование с учетом модульного тестирования 16](#_Toc472076805)

[Понижение связанности кода, использование шаблонов проектирования 16](#_Toc472076806)

[Создание заглушек (фреймворк Mockito) 16](#_Toc472076807)

[Общие рекомендации по написанию модульных тестов 19](#_Toc472076808)

[Чек-лист проверки нового кода на покрытие тестами 22](#_Toc472076809)

[Соглашения об использовании модульных тестов 23](#_Toc472076810)

[Используемые библиотеки (фреймворки) 23](#_Toc472076811)

[Соглашения об именовании 24](#_Toc472076812)

[Соглашения о документировании 25](#_Toc472076813)

[Ответственность за выполнение и создание модульных тестов 25](#_Toc472076814)

[Приложения 26](#_Toc472076815)

[Приложение 1. Пример методики разработки и тестирования 26](#_Toc472076816)

[Литература 27](#_Toc472076817)

# Что такое модульное тестирование

## Отступление

Многие положения этого документа касаются не только модульных тестов, но и других видов автоматического тестирования, и даже тестирования в целом. Тем не менее, в документе часто упоминается только модульное тестирование, таково назначение документа.

Часто *модульное тестирование* называется *юнит-тестированием*, поэтому в данном документе я буду использовать оба термина. В контексте этого документа, оба термина означают одно и то же.

## Назначение

Модульное тестирование применяется для проверки корректности реализации отдельных модулей исходного кода. Корректность реализации проверяется путем валидации выходных параметров методов и состояния экземпляра тестируемого класса.

Модульное тестирование используется:

* Разработчиками – для быстрой проверки корректности работы изменяемого кода, для отладки специфических ситуаций в приложении, для «защиты» кода от некорректного изменения;
* Тестировщиками – как элемент автоматизированного тестирования, как средство для быстрого контроля валидности исходного кода в системе контроля версии;
* Разработка тестов (в том числе модульных) является первым этапом разработки в модели TDD (Test Driven Development);
* Модульные тесты являются одним из средств документирования кода.

Модульные тесты представляют собой классы, написанные с использованием специальных фреймворков, предоставляющих удобный интерфейс для их выполнения. В каждом тестовом классе реализуется функциональность, необходимая для вызова методов тестируемых классов с различными аргументами. Значения, возвращаемые проверяемыми методами (либо значения полей экземпляра класса), сравниваются с эталонными значениями, рассчитанными вручную. Аналогично проверяется обработка ошибок – анализируется обработка некорректных значений входных параметров.

Тестовые фреймворки – библиотеки (наборы классов), которые предоставляют готовые механизмы для решения общих задач тестирования – интерфейс запуска, визуализация прогресса выполнения и результатов, логгирование, инициализация и очистка тестовой среды, реализацию заглушек, реализацию проверок значений.

## Терминология

**Заглушка (stub, mock)** - заменяют недостающие компоненты, которые вызываются элементом и выполняют следующие действия:

• возвращаются к элементу, не выполняя никаких других действий;

• отображают трассировочное сообщение и иногда предлагают тестеру продолжить тестирование;

• возвращают постоянное значение или предлагают тестеру самому ввести возвращаемое значение;

• осуществляют упрощенную реализацию недостающей компоненты;

• имитируют исключительные или аварийные условия.

**Покрытие кода** - мера, используемая при тестировании программного обеспечения. Она показывает процент, насколько исходный код программы был протестирован.

**Связность** кода– (или реже **прочность**) (англ. **cohesion**, англ. **module strength**) — мера силы взаимосвязанности элементов внутри модуля; способ и степень, в которой задачи, выполняемые некоторым программным модулем, связаны друг с другом.

**Тестовый случай (тест)** - артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части. Соответствует одному методу тестового класса

**Юнит-тестирование** – то же, что и модульное тестирование (от английского *unit* – *модуль*). 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.

**White-box testing (тестирование белого ящика)**. Для конструирования тестов используются внутренняя структура кода и управляющая логика. Тестировщик получает тестовые данные путём анализа логики работы программы. При этом существует вероятность, что код будет проверяться так, как он был написан, а это не гарантирует корректность логики.

**Black-box testing (тестирование черного ящика)**. Для конструирования тестов используются требования и спецификации ПО, при котором не используется знание о внутреннем устройстве тестируемого объекта.

Недостатки:

• таким способом невозможно найти ошибки, которые компенсируются другими ошибками;

• некоторые ошибки возникают достаточно редко (ошибки работы с памятью) и потому их трудно найти и воспроизвести

# Роль модульного тестирования в общем процессе тестирования

Как правило, модульное тестирование является первым этапом в общем цикле тестирования программного обеспечения. Легкость его выполнения обуславливает его активное использование на этапе написания исходного кода. Дистрибутив, передаваемый на тестирование ИФТ, должен быть уже протестирован с помощью модульных тестов (и других видов автоматических тестов, запуск которых доступен разработчикам).

## Связь модульного и интеграционного тестирования

Модульное тестирование позволяет убедиться в корректной реализации интерфейсов, предоставляемых определенным модулем приложения (классом). Обычно тестируется отдельный класс. Интеграционное тестирование проверяет корректность взаимодействия модулей системы. Четкой границы между модульным и интеграционным тестированием не существует, так как в реальном коде часто класс использует логику, реализованную и в других классах.

Если для взаимодействия с внешними классами не требуется развертывание тестовой среды, то это считается модульным тестированием. При этом все внешние модули, взаимодействующие с внешним миром – сетью, пользователями, базами данных, файловой системой, заменяются заглушками, имитирующими работу модулей для конкретных тестовых случаев. Подразумевается, что заглушки не содержат в себе ошибок, поэтому, в конечном счете, тестирование сводится к тестированию класса, интерфейсы которого вызываются в юнит-тесте.

Если же тестируется корректность результатов работы нескольких модулей, то этот вид тестирования называется интеграционным. При интеграционном тестировании необязательно тестируется вся система, чаще всего тестируется взаимодействие отдельных модулей, остальные модули могут быть заменены заглушками, имитирующими поведение реальных модулей для конкретных тестовых случаев. Часто интеграционные взаимодействия инициируются с помощью тестов, аналогичных модульным.

Обобщая, можно сказать, что модульные тесты проверяют внутреннюю логику работы модуля, интеграционные тесты проверяют корректное взаимодействие различных модулей.

## Связь модульного и регрессионного тестирования

Регрессионное тестирование используется для обнаружения ошибок в ранее протестированном исходном коде. Если в результате внесения изменения в исходный код появляется дефект в функциональности, ранее прошедшей успешное тестирование, то такой дефект считается регрессионным.

Регрессионное тестирование в общем случае основывается на полном прогоне всех предыдущих тестов. Часто используется тестирование всех ранее закрытых и верифицированных дефектов (не стали ли они воспроизводиться?). Регрессионное тестирование считается наиболее трудоемким и механически сложным («нудным»).

Использование модульного тестирования позволяет повысить качество регрессионного тестирования. Модульные тесты исполняются быстро, степень покрытия тестами функционала напрямую связана со степенью покрытия исходного кода модульными тестами. При удовлетворительном качестве покрытия модульные тесты позволяют быстро проверить работу каждого отдельного модуля системы.

Важной практикой является написание модульного теста на каждый новый дефект, регистрируемый на систему. Это позволяет избежать повторного возникновения этого дефекта по той же причине.

# Модульное тестирование, как часть процесса разработки

Написание модульных тестов – это, в первую очередь, часть процесса разработки и один из инструментов обеспечения качества исходного кода. Создание модульных тестов рекомендуется возлагать на разработчиков – как правило, автор тестируемого кода должен и писать модульные тесты к этому коду, в больших командах часть этой функции можно поручить тестировщикам, обладающим необходимыми навыками.

## Рефакторинг

Рекомендации по проведению рефакторинга были разработаны в середине 90-х годов прошлого века Кентом Беком и Мартином Фаулером. Рефакторинг был предложен как неотъемлемая часть процесса разработки, призванная к непрерывному улучшению исходного кода.

Рефакторингом называется процесс изменения внутренней структуры исходного кода модуля, не затрагивающий его внешнего поведения. Цель рефакторинга – улучшение качества существующего кода без добавления новой функциональности и изменения существующей функциональности. Изменения производятся с помощью небольших шаблонных эквивалентных преобразований.

Выполнение сложного рефакторинга несет в себе риск допущения некорректного (неэквивалентного) преобразования существующего кода. Это может привести к появлению в модуле скрытых дефектов.

Правильной практикой, рекомендованной разработчиками методики рефакторинга, является предварительное покрытие модульными тестами исходного кода, который будет рефакториться. Наличие качественных тестов позволяет убедиться, что рефакторинг не разрушил существующую функциональность. При этом подразумевается, что рефакторинг не влияет на модульные тесты, они не подвергаются изменению, так как рефакторинг не изменяет сигнатуры интерфейсов модуля.

Значение тестов для рефакторинга переоценить невозможно. Любой, даже самый небольшой рефакторинг, как известно, требует наличия написанных тестов. Эта мысль важна настолько, что Мартин Фаулер в своей книге "Refactoring" посвятил модульному тестированию целую главу. Но набор модульных тестов, покрывающих большую часть приложения, позволяет модифицировать систему значительно более агрессивно и со значительно более предсказуемыми результатами. Именно сочетание рефакторинга и тестов позволяет разработчикам быстро изменять систему в любом нужном заказчику направлении. И именно поэтому мы можем позволить себе не искать гибких решений, а использовать самые простые (цена изменений при наличии тестов не слишком высока).

## Отладка

Воспроизведение отдельных ситуаций в программе иногда очень затруднительно – условия, при которых ситуация воспроизводится не всегда легко воспроизвести. В результате отладка дефекта становится мучительной проблемой.

На помощь в этом случае так же могут прийти модульные тесты. Условия, необходимые для воспроизведения дефекта, можно жестко указать в параметрах вызываемого тестового метода. Тесты также можно отлаживать, что позволяет быстро перейти от тестового метода к тестируемому коду, в котором скрыт дефект. В дальнейшем, после исправления дефекта, тест служит надежной защитой от повторного появления этого дефекта.

## Регрессионное тестирование

При обнаружении дефекта в исходном коде рекомендуется сначала его воспроизвести с помощью модульного теста, затем исправлением дефекта заставить успешно проходить модульный тест.

В дальнейшем этот юнит-тест рекомендуется оставить, чтобы он выполнялся при каждом запуске набора модульных тестов. Таким образом, тест будет выполнять функцию регрессионного теста, защищая исходный код от повторного возникновения ранее исправленного дефекта.

## Test Driven Development

Существует широко известная методология разработки TDD – Test Driven Development, разработка через тестирование. Суть ее сводится к первоначальной реализации требований к ПО в виде пустых классов-интерфейсов, к которым пишутся модульные тесты. При первом запуске тесты «падают» - выполняются неуспешно. Затем по мере реализации логики тесты начинают выполняться корректно. Когда все тесты становятся успешно проходимыми, можно считать, что требования реализованы полностью.

Концепция была разработана и впервые опубликована в 1999 году Кентом Беком в книге «Разработка через тестирование: на примере». Кратко он выразил основную идею методологии одной фразой – «Test first» («Сначала тест»).

Основные положения TDD:

* Код с не оттестированными участками не может быть опубликован;
* Тесты должны базироваться на спецификации;
* На каждое требование должен быть, как минимум, один тест, в том числе, возможно, модульный тест.

Первоначально разработка через тестирование рассматривалась как одна из практик *экстремального программирования*, однако позже выделилась как независимая методология. В данный момент ничто не мешает использовать TDD как для целых проектов, так и для реализации отдельных изменений.

Рекомендуется следующий алгоритм использования TDD для реализации отдельных изменений:

* 1. Запуск существующих тестов, если они не проходят, то поиск и устранение причин;
  2. Рефакторинг кода, если он необходим для реализации новых требований;
  3. Запуск существующих тестов, если они не проходят, то поиск и устранение причин (п. 2);
  4. Создание теста (возможно, нескольких тестов) на основании функциональных требований или технического задания;
  5. Проверка, что новый тест не проходит – если он уже проходит, значит, он некорректно реализован (или требуемая функциональность уже существует);
  6. Реализация требований в дорабатываемом модуле;
  7. Запуск тестов, проверяющих работу модуля;
  8. Если тесты не проходят, то продолжение доработок (п.6);
  9. Если тесты успешно проходят, значит, требования реализованы



## Тесты, как примеры кода

Модульные тесты выполняют неявную, но важную функцию документирования исходного кода. Изучая код модульных тестов, новый разработчик может быстрее разобраться с использованием тестируемого модуля – поймет, какие значения параметров являются допустимыми, какие предварительные подготовки требуются для вызова метода.

Эта функция модульных тестов тесно переплетается с концепцией разработки через тестирование. Фактически предварительное написание модульных тестов позволяет задокументировать требования к реализуемым интерфейсам. Поддержание тестов в актуальном состоянии при изменении внутренней логики тестируемых модулей также более надежно, чем обновление документации.

Наконец, «документирование через тестирование» реализуется в виде исходного кода, который однозначно понимается любым разработчиком, знакомым с синтаксисом языка. В то же время функциональные требования, излагаемые аналитиком на естественном языке, в принципе допускают неоднозначную трактовку.

## Гибкие технологии разработки, Sbergile, DevOps

Модульное тестирование возникло и является неотъемлемой частью гибких методологий разработки. Оно является одним из инструментов для успешной реализации основополагающих принципов agile-манифеста:

1. Модульные тесты, как и любые автоматические тесты в целом, способствуют *более регулярной и ранней поставке ценного программного обеспечения*. В любой момент можно запустить наборы модульных тестов и убедиться, что код в системе контроля версий рабочий. Или нерабочий, но значит, кто-то уже занимается разбором причин непрохождения тестов;
2. Модульные тесты обеспечивают поддержку *быстрых изменений требований*, благодаря возможности быстро проверить корректность работы изменяемого кода. Использование концепции TDD (test first!) позволяет проводить изменения требований очень быстро и без риска некорректной реализации;
3. Важным индикатором *работающего продукта* являются безошибочно выполняющиеся модульные тесты;
4. *Техническое совершенство и качество проектирования* обеспечиваются, в том числе, и непрерывным рефакторингом. В свою очередь качественный рефакторинг невозможен без использования юнит-тестов;
5. Наконец, *простота – искусство минимизации лишней работы* – это основное преимущество разработки через тестирование. Именно при TDD задача разработчика сводится к изменению кода таким образом, чтобы заработали модульные тесты. Не более и не менее. Реализация какой-либо функциональности, не покрытой тестами, не требуется (либо эту функциональность надо покрыть тестами).

Одна из главных практик методологии DevOps – непрерывная интеграция (continuous integration, CI). Она подразумевает регулярную (частую) автоматизированную сборку проекта из исходников, хранящихся в системе контроля версий, автоматический запуск модульных тестов и развертывание приложения на тестовых стендах. Так же многие системы поддержки непрерывной интеграции позволяют инициировать автоматический запуск модульных тестов при любом коммите в основную ветку проекта в системе контроля версий.

Непрерывная интеграция позволяет обеспечить непрерывное автоматическое отслеживание качества исходного кода. Проблемы интеграции выявляются и исправляются быстрее, в системе версионного контроля всегда лежит рабочий код, разработчики сразу видят эффект от своих изменений.

Внедряемые в Компании гибкие методологии разработки, в том числе Sbergile, декларируют поддержку ценностей, описанных в манифесте Agile. Следовательно, им не чужды и современные практики, рекомендуемые мировым сообществом ИТ-профессионалов. В том числе, в Компании придают большое значение модульному тестированию и непрерывной интеграции в целом.

# Рекомендации по написанию тестов

## Требования к модульным тестам

Требования к юнит-тестам иногда излагают с помощью принципа FIRST:

• [F]ast – (быстрые) должны быстро исполняться

• [I]solated – (изолированные) не должны влиять на исполнение других тестов

• [R]epeatable – (повторяемые) должны давать тот же результат при повторном запуске

• [S]elf-validating – (проверяемые) должны содержать четкую интерпретацию результатов выполнения рабочего кода

• [T]imely – (своевременные) должны проверять новый код с минимальным отрывом от времени его создания

Если Вы будете следовать этим пяти принципам при написании модульных тестов, Вы получите качественные тесты и, следовательно, надежный рабочий код.

## Структура тестового класса

Рассмотрим структуру тестового класса с использованием фреймворка JUnit4.

Тестовый класс нет необходимости наследовать от каких-либо предков или помечать аннотациями.

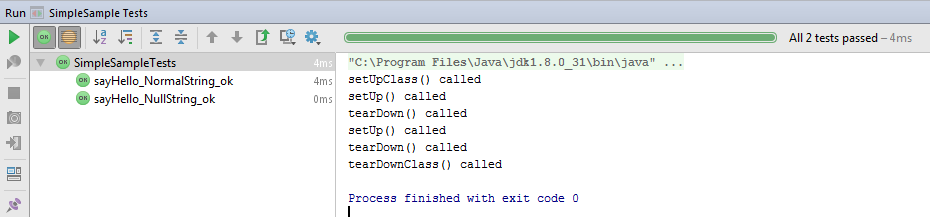
Тестируемый класс:

*/\*\*  
 \* Created by yanzin-sg on 23.08.2016.  
 \*/***public class** SimpleSample {  
  
 **public** String sayHello(**final** String name) {  
 **return "Hello, "** + name;  
 }  
}

Модульный тест для этого класса:

**import static** org.junit.Assert.\*;  
  
*/\*\*  
 \* Created by yanzin-sg on 23.08.2016.  
 \*/***public class** SimpleSampleTests {  
  
 @org.junit.BeforeClass  
 **public static void** setUpClass() **throws** Exception {  
 *// код, инициализирующий среду выполнения сценария* System.***out***.println(**"setUpClass() called"**);  
 }  
  
 @org.junit.AfterClass  
 **public static void** tearDownClass() **throws** Exception {  
 *// код, очищающий среду после выполнения сценария* System.***out***.println(**"tearDownClass() called"**);  
 }  
  
 @org.junit.Before  
 **public void** setUp() **throws** Exception {  
 *// код, инициализирующий среду перед выполнением каждого тестового случая (метода)* System.***out***.println(**"setUp() called"**);  
 }  
  
 @org.junit.After  
 **public void** tearDown() **throws** Exception {  
 *// код, восстанавливающий среду после выполнения каждого тестового случая (метода)* System.***out***.println(**"tearDown() called"**);  
 }  
  
 @org.junit.Test  
 **public void** sayHello\_NormalString\_ok() **throws** Exception {  
 *// Arrange* SimpleSample instance = **new** SimpleSample();  
 *// Act* String hello = instance.sayHello(**"Sergey"**);  
 *// Assert  
 assertEquals*(**"Hello, Sergey"**, hello);  
 }  
  
 @org.junit.Test  
 **public void** sayHello\_NullString\_ok() **throws** Exception {  
 *// Arrange* SimpleSample instance = **new** SimpleSample();  
 *// Act* String hello = instance.sayHello(**null**);  
 *// Assert  
 assertEquals*(**"Hello, null"**, hello);  
 }  
  
}

Результат выполнения тестового примера



## Объекты тестирования – все публичные методы модуля, сложные внутренние методы

Модульное тестирование в первую очередь подразумевает тестирование внешних интерфейсов модуля, то есть всех методов, доступных извне класса или пакета, содержащего тестируемый класс.

## Какие методы нужно тестировать

Обязательно тестируются все публичные методы – ими пользуются другие классы. Из методов с ограниченным доступом (private, protected, package default) тестируются только те, которые подвержены частым изменениям или которые содержат сложную логику.

Очень спорный момент – тестирование приватных методов. Прежде всего, необходимо проанализировать причины, по которым метод необходимо протестировать. Если он реализует некоторую локальную функциональность, которую имеет смысл выделить в отдельный класс – вынесите ее в отдельный класс, пусть метод станет доступным для юнит-тестов. Если же метод играет вспомогательную роль, то возможно достаточно протестировать доступные методы, его использующие. Так или иначе, открывать доступ к тестируемому методу только для целей тестирования будет нарушением инкапсуляции, и этого следует избегать.

Если нет возможности избежать написания юнит-тестов для приватных методов, то для доступа к таким методам рекомендуется использовать рефлексию, либо дополнительные фреймворки, позволяющие к ним обратиться – например, PowerMock.

Например, чтобы отладить приватный метод в модульном тесте следует получить экземпляр класса, с помощью рефлексии найти нужный метод, разрешить доступ к нему и вызвать его:

Тестируемый класс:

**public class** Main {  
  
 **public static void** main(String[] args) {  
 System.***out***.println(**new** Main().dateToJulian(Calendar.*getInstance*()));  
 }  
  
 **private double** dateToJulian(**final** Calendar date) {  
 **int** year = date.get(Calendar.***YEAR***);  
 **int** month = date.get(Calendar.***MONTH***) + 1;  
 **int** day = date.get(Calendar.***DAY\_OF\_MONTH***);  
 **int** hour = date.get(Calendar.***HOUR\_OF\_DAY***);  
 **int** minute = date.get(Calendar.***MINUTE***);  
 **int** second = date.get(Calendar.***SECOND***);  
 **double** extra = (100.0 \* year) + month - 190002.5;  
 **return** (367.0 \* year) - (Math.*floor*(7.0 \* (year + Math.*floor*((month + 9.0) / 12.0)) / 4.0))  
 + Math.*floor*((275.0 \* month) / 9.0) + day  
 + ((hour + ((minute + (second / 60.0)) / 60.0)) / 24.0)  
 + 1721013.5 - ((0.5 \* extra) / Math.*abs*(extra)) + 0.5;  
 }  
  
}

Модульный тест:

**public class** MainTest {  
  
 @Test  
 **public void** testDateToJulian() **throws** NoSuchMethodException, InvocationTargetException,  
 IllegalAccessException {  
 Main m = **new** Main();  
 Class mainClass = m.getClass();  
 Method methodDateToJulian = mainClass.getDeclaredMethod(**"dateToJulian"**, Calendar.**class**);  
 methodDateToJulian.setAccessible(**true**);  
 Calendar calendar = Calendar.*getInstance*();  
 calendar.setTime(**new** Date(116, 7, 20, 0, 0, 0));  
 Double result = (Double) methodDateToJulian.invoke(m, calendar);  
 *assertEquals*(2457620.5d, result.doubleValue(), 0.2);  
 }  
}

При изменении сигнатуры тестируемого метода тест перестанет работать, необходимо исправить строки, выделенные желтым цветом. Для облегчения чтения кода можно вынести код поиска и вызова метода в отдельный метод.

## Какие методы не нужно тестировать

Нет смысла тестировать небольшие методы, не содержащие сложной логики и не подразумевающие частого изменения. Например, нет необходимости тестировать геттеры и сеттеры. Поскольку конструкторы используются для инициализации первоначального состояния объекта класса, то и их нет смысла тестировать. Размещение в конструкторе какой-то логики не рекомендуется. Если конструктор содержит в себе сложную логику проверки своих параметров, рассмотрите возможность вынесения этих проверок в отдельные методы (Extract Method) или использования паттернов проектирования, заменяющих прямой вызов конструкторов (Builder, Factory Method, Abstract Factory и т.п.).

# Проектирование с учетом модульного тестирования

## Понижение связанности кода, использование шаблонов проектирования

При проектировании приложения размещайте код, взаимодействующий с внешней средой – базами данных, файловой системой, сетевыми ресурсами, - в отдельные классы. Создание (получение) ссылок на экземпляры этих классов реализуйте в отдельном методе, который можно будет заменить в тестовом классе.

Вообще, при разработке следует руководствоваться принципом «одна функция – один класс». Новые функции размещайте в отдельных классах, ответственных за ее выполнение.

## Создание заглушек (фреймворк Mockito)

Если модуль, вызываемый тестируемым кодом, содержит обращения к внешней среде (базам данных, файловой системе, сетевым ресурсам), следует заменить этот модуль заглушкой. Вообще любая зависимость, точное поведение которой неизвестно и которая по этой причине может влиять на результат выполнения теста, должна быть заменена заглушкой.

В простейшем случае, заглушка – это класс, реализующий интерфейс, ожидаемый тестируемым классом. Вы можете создать наследника заменяемого класса, переопределить в нем вызываемые тестируемым классом методы, и возвращать из них некоторые фиксированные значения.

Например, для тестирования метода, который на вход получает HttpServletRequest, необходимо создать наследника этого абстрактного класса, и переопределить в нем необходимые методы. Именно в этом случае – придется переопределить все методы, так как это интерфейс.

Значительно проще использовать для создания заглушек фреймворк Mockito. Принцип его работы основан на создании обертки (mock) для заменяемого класса. Для нужных методов этой обертки для разных комбинаций аргументов можно задать возвращаемые значения. Это позволяет не реализовывать весь класс/интерфейс.

Пример, тестируемый метод:

**public boolean** isCredentialsValid(HttpServletRequest req) {  
 **if** (**"user1"**.equals(req.getParameter(**"login"**))  
 && **"pa$$word"**.equals(req.getParameter(**"password"**)))  
 **return true**;  
 **return false**;  
}

Создание заглушки в тестовом методе:

@Test  
**public void** login\_wrong() {  
 HttpServletRequest req = *mock*(HttpServletRequest.**class**);  
 *when*(req.getParameter(**"login"**)).thenReturn(**"user1"**);  
 *when*(req.getParameter(**"password"**)).thenReturn(**"password"**);  
 *assertFalse*(isCredentialsValid(req));  
}

Для проверки поведения тестируемого объекта фреймворк Mockito позволяет задавать и проверять последовательность вызовов объекта-заглушки. Используйте для этой цели метод verify() класса Mockito. Например, если мы ожидаем, что тестируемый объект вызовет поиск объекта в БД, затем его удаление, то мы можем написать такой код:

Тестируемый метод:

**public void** workWithObject(**int** id) {  
 **dataAccess**.findObject();  
 **dataAccess**.deleteObject(id);  
}

Тестовый метод:

@Test  
**public void** someMethod\_testBehavior\_findAndDelete() {  
 DataAccess dao = *mock*(DataAccess.**class**);  
  
 *when*(dao.findObject()).thenReturn(**true**);  
 *when*(dao.deleteObject(1)).thenReturn(**true**);  
  
 SimpleSample instance = **new** SimpleSample(dao);  
 instance.workWithObject(1);  
  
 *verify*(dao, *times*(1)).findObject();  
 *verify*(dao, *times*(1)).deleteObject(1);  
}

В этом примере мы создаем mock-заглушку для объекта DataAccess. Затем мы вызываем тестируемый код и проверяем, сколько раз вызывались методы заглушки.

Если нужно создать заглушку для статического метода, можно воспользоваться фреймворком PowerMock. Например, мы пишем тест на класс, который вызывает статический метод в другом классе, который в свою очередь обращается к внешним системам. Рассмотрим пример (ниже). У нас есть класс LoginManager, который обращается к статическому методу в классе, осуществляющем загрузку пользователей из БД. Создадим заглушку для этого метода средствами PowerMock:

Тестируемый код (часть кода опущена и упрощена):

**public class** UserDAO {  
 **public static** User[] loadUsers() **throws** Exception {  
 *// здесь некий код, который работает с внешними системами (например, с БД)* **throw new** Exception(**"database not available"**);  
 }  
}

**public class** LoginManager {  
 **private static** LoginManager *instance*;  
 **private** User[] **users**;  
  
 **private** LoginManager() {  
 }  
  
 **public static** LoginManager getInstance() **throws** Exception {  
 **if** (*instance* == **null**) {  
 LoginManager temp = **new** LoginManager();  
 temp.loadUsers();  
 *instance* = temp;  
 }  
 **return** *instance*;  
 }  
  
 **public boolean** validateLogin(String username, String password) {  
 **for** (User user : **users**) {  
 **if** (username.equalsIgnoreCase(user.getName()))  
 **return** password.equals(user.getPassword());  
 }  
 **return false**;  
 }  
  
 **private void** loadUsers() **throws** Exception {  
 **users** = UserDAO.*loadUsers*();  
 }  
}

Тестовый код:

**import** com.sbt.unittesting.dao.UserDAO;  
**import** com.sbt.unittesting.model.User;  
  
**import** org.junit.Test;  
**import** org.junit.runner.RunWith;  
**import** org.powermock.core.classloader.annotations.PrepareForTest;  
**import** org.powermock.modules.junit4.PowerMockRunner;  
  
**import static** org.junit.Assert.\*;  
**import static** org.powermock.api.mockito.PowerMockito.\*;  
  
@RunWith(PowerMockRunner.**class**)  
@PrepareForTest(UserDAO.**class**)  
**public class** LoginManagerTest {  
  
 @Test  
 **public void** testValidateLogin() **throws** Exception {  
 *mockStatic*(UserDAO.**class**);  
 *when*(UserDAO.*loadUsers*()).thenReturn(  
 **new** User[] {  
 **new** User(1, **"vasya"**, **"tamara"**),  
 **new** User(2, **"user"**, **"qwerty"**),  
 **new** User(3, **"john"**, **"bart"**)  
 });  
 *assertTrue*(LoginManager.*getInstance*().validateLogin(**"user"**, **"qwerty"**));  
 }  
}

# Общие рекомендации по написанию модульных тестов

1. Каждый тестовый метод должен тестировать только один тестовый случай ([Single Responsibility Principle](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%B5%D0%B4%D0%B8%D0%BD%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B9_%D0%BE%D0%B1%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8)). Если у тестируемого метода сложный алгоритм, напишите для него несколько тестов по количеству логических веток в алгоритме;
2. Именуйте тесты понятно – по названию тестового метода должно быть понятно, что тестируется и какой результат ожидается;
3. Выносите общий код из тестовых методов в метод setUp(), который инициализирует тестовую среду для данного сценария. Аналогично используйте метод tearDown() для освобождения ресурсов, используемых тестовым сценарием. В JUnit4 setUp() может быть заменен любым методом с аннотацией @Before, tearDown() заменен методом с аннотацией @After. Тест может содержать несколько методов с такими аннотациями. Метод, помеченный аннотацией @Before, выполняется перед запуском каждого тестового метода. Для однократного запуска метода до запуска тестовых методов используйте аннотацию @BeforeClass. Для выполнения завершающих действий после всех тестовых методов используйте метод, помеченный аннотацией @AfterClass;
4. Фреймворк Mockito позволяет использовать инъекцию mock-объектов – пользуйтесь этой возможностью и не забывайте инициализировать аннотации в методе setUp() тестового сценария. В начале каждого тестового метода вызывайте у mock-объекта метод reset():  
     
   @Mock  
   TestingClass instance;  
     
   @Before  
   public void setUp() {  
    MockitoAnnotations.initMocks(this);  
   }  
     
   public void testMethod\_someCase\_ok() {  
    instance.reset();  
    // ...  
   }
5. Пишите документацию к тестам, аналогично тому, как Вы это делаете для рабочего кода –описывайте тестовый случай, который реализует метод. Возможно, Ваш тест будет сопровождать другой инженер, и документация поможет ему освоить Ваш код быстрее. Также Ваши пояснения помогут разработчику понять, что делает тестовый метод, если он перестанет корректно выполняться («проходить»). В случае если в Вашем проекте используется методология TDD, документирование тестов несет двойную нагрузку – документация должна содержать не только описание тестового случая, но и информацию о тестируемом требовании.
6. Тест не должен зависеть от среды запуска. Не привязывайтесь к базам данных. Если база данных обязательно нужна, выносите ее настройки в ресурсы приложения, храните настройки для каждого пользователя отдельно. Не храните пути к файлам в коде тестов - выносите все файлы в ресурсы приложения.
7. Делайте каждый тест независимым от других тестов. Результат выполнения тестов не должен зависеть от порядка их выполнения. Для объектов с состояниями – создавайте новый экземпляр для каждого класса в начале тестового метода. Для заглушек, создаваемых библиотекой Mockito, вызывайте метод reset(). Разные тестовые сценарии тестируйте в разных тестовых классах.  
   Если класс сохраняет свое состояние между вызовами, казалось бы, независимых методов, проверьте его реализацию, возможно, она содержит архитектурные ошибки;
8. Необходимость юнит-теста для метода не ограничивается его видимостью. Если метод сложный, или часто меняется, или содержит критически важный код, или просто активно используется другими методами – пишите для него юнит-тест. Члены класса с ограниченным доступом можно вызывать с помощью рефлексии – смотрите пример выше. Mockito на момент написания этого документа не имеет возможности создавать заглушки для приватных методов. Но можно использовать фреймворк PowerMock, который обладает такой возможностью и легко подключается к Mockito. Как вариант, можно рассмотреть возможность увеличения доступности метода, но необходимо понимать, что это нарушение инкапсуляции;
9. Используйте в тестовом методе только одну проверку. Один вызов тестового метода и одну проверка его результата (одна серия проверок полей одного результата).
10. Если ожидаемое поведение Вашего кода в данном тестовом случае – выброс исключения, то не надо его обрабатывать в коде метода – пропишите выбрасываемое исключение в аннотации @Test в поле expected – JUnit корректно его обработает: если тест выбросит заявленное исключение, то тест будет считаться успешно пройденным, если же код не выбросит это исключение (или выбросит другое исключение), то тест будет помечен, как неуспешный.  
    Отсюда вытекает другая рекомендация – не указывайте в аннотации исключение более широкого типа – это может привести к скрытию других исключений, также являющихся наследником заявленного класса, но не подразумеваемых корректной реализацией требований.  
    Пример:  
    @Test(expected = IllegalTransferState.class)  
    public void stornTransfer\_WrongState\_ThrowsException() { … }
11. Аккуратно указывайте параметры assert-методов. Первый параметр – это ожидаемое значение, второй параметр – значение, которое тестируемый метод фактически возвращает. Не надо менять их местами – с точки зрения математики разницы нет, но с точки зрения понимания исходного кода лучше порядок соблюдать. По той же причине лучше указывать более понятный текст сообщения, выдаваемого при отрицательном результате проверки – так будет понятнее для других пользователей Вашего кода. Не пишите, *что* некорректно в тесте, пишите – *почему* не проходит тест.
12. Убедитесь, что тестовый код отделен от рабочего кода. Современные среды разработки и сборки позволяют поместить код модульных тестов в отдельной ветке. Лишний код в приложении приводит к ненужному расходованию ресурсов.  
    Аналогичная рекомендация касается собственно исходного кода рабочих классов. Не стоит как-то подстраивать рабочие классы под разработку модульных тестов. Грамотно построенная архитектура и уместное использование шаблонов проектирования достаточно для разработки кода, хорошо приспособленного для написания юнит-тестов. Не надо делать приватные методы доступными извне, если это связано только с необходимостью написания тестов. Средств рефлексии достаточно для доступа к любым членам классов.
13. Техническое руководство проекта должно уделять достаточно внимания разработке юнит-тестов и поддержанию их в актуальном состоянии. Новый код необходимо в обязательном порядке покрывать модульными тестами, тесты для старого кода необходимо обновлять при изменении оного. Одним из обязательных критериев успешности любого изменения кода необходимо считать успешное прохождение всех модульных тестов. Для автоматизации запуска тестов следует внедрять средства непрерывной интеграции (continuous integration, CI). После каждого коммита в систему контроля версий должна происходить автоматическая сборка проекта и запуск всех юнит-тестов. Падение тестов может быть основанием для отказа в успешном коммите – этот критерий настраивается в каждом проекте индивидуально, как правило, в крупных проектах это мешает разработке, в небольших проектах такая настройка дисциплинирует разработчиков.
14. Не используйте конструктор теста для инициализации тестовой среды. Для этой цели уже есть метод setUp(). Для целей очистки (восстановления) тестовой среды есть метод tearDown()
15. Не скрывайте исключения, выбрасываемые тестируемым кодом, если эти исключения не должны быть в данном тестовом случае. Поймать исключение и выдать fail() или вывести стек-трейс в консоль – это практики, ничего не дающие Вам для улучшения Вашего кода;
16. Не забывайте вызывать в методах setUp() (@Before) и tearDown() (@After) соответствующие методы предков. Возможно, в классах-предках эти методы тоже участвуют в инициализации тестовой среды и если их не вызвать, то тестовая среда будет инициализирована некорректно;
17. Убедитесь, что выполнение тестов не зависит от времени их запуска. Тест должен работать одинаково в любое время суток сегодня, завтра, через месяц, через год.
18. Убедитесь, что тест будет корректно выполняться в тестовых средах с разными региональными настройками. Если Вы жестко прописываете в проверках дату в текстовом формате, убедитесь, что проверка корректно отработает на компьютере с другой локалью. Лучший вариант – не использовать вообще локалезависимых данных до того момента, когда они потребуются.
19. Добивайтесь, чтобы тесты были небольшими и работали максимально быстро. Ставьте разумные ограничения на время выполнения тестов. JUnit поддерживает ограничение времени выполнения каждого тестового метода с помощью аннотации @Test – в ней есть параметр timeout, в котором можно ограничить время выполнения теста.
20. Не используйте в тестовом классе статические члены. Если без них никак, то инициализируйте их заново в каждом тестовом методе.
21. Тестируйте все сценарии по отдельности, даже если успешное прохождение одного сценария подразумевает успешное прохождение другого сценария. В случае изменения алгоритма такая зависимость между сценариями может измениться, и второй сценарий останется непокрытым тестами.
22. Если юнит-тест перестал успешно исполняться – не отключайте его исполнение аннотацией @Ignore. Если тестовый сценарий стал неактуальным, то удалите тест совсем. Если тестовый сценарий актуален – проанализируйте, в чем причина неуспешного исполнения теста – в дефекте рабочего кода или в изменении логики работы тестируемого кода.
23. Используйте при кодировании тестового случая стиль Arrange-Act-Assert (AAA). Суть его в разделении кода тестового метода на 3 блока – блока подготовки тестовой среды (Arrange – организовать), блока вызова тестируемого кода (Act – действовать), блока проверки логических утверждений (Assert – утверждать). Такой стиль легко читается и поддерживается другими разработчиками.

## Чек-лист проверки нового кода на покрытие тестами

1. Для каждого нового класса, содержащего логику, следует создавать новый тестовый сценарий (или несколько тестовых сценариев, если это применимо);
2. Каждый тестовый случай, реализуемый классом, должен покрываться как минимум одним тестовым методом. В общем случае, количество тестовых методов зависит от цикломатической сложности метода;
3. Нет необходимости тестировать геттеры, сеттеры, конструкторы, методы для реализации типовых шаблонов проектирования – синглтонов, фабрик и т.п.;
4. Для оценки покрытия кода тестами следует использовать различные инструменты оценки покрытия кода модульными тестами, например, JaCoCo.

# Соглашения об использовании модульных тестов

## Используемые библиотеки (фреймворки)

В Компании для реализации модульных тестов в проектах на Java рекомендуются следующие фреймворки:

* JUnit 4 – для создания и запуска модульных тестов
  1. Сайт - <http://junit.org/junit4/>
  2. Getting started - <https://github.com/junit-team/junit4/wiki/Getting-started>
  3. JavaDoc - <http://junit.org/junit4/javadoc/latest/index.html>
  4. Downloads - <https://github.com/junit-team/junit4/wiki/Download-and-Install>
  5. Maven dependency:  
     <dependency>  
      <groupId>junit</groupId>  
      <artifactId>junit</artifactId>  
      <version>${junit.version}</version>  
      <scope>test</scope>  
     </dependency>
* TestNG – для создания и запуска модульных тестов
  1. Сайт – [http://testng.org](http://testng.org/)
  2. Getting started – <http://testng.org>
  3. JavaDoc - <http://testng.org/javadocs/index.html>
  4. Downloads - <http://testng.org/doc/download.html>
  5. Maven dependency:  
     <dependency>  
      <groupId>org.testng</groupId>  
      <artifactId>testng</artifactId>  
      <version>${testing.version}</version>  
      <scope>test</scope>  
     </dependency>
* Mockito – создание заглушек (mock-объектов)
  1. Сайт - <http://mockito.org/>
  2. JavaDoc - [http://site.mockito.org/mockito/docs/current/org/mockito/Mockito.html](http://site.mockito.org/mockito/docs/current/org/mockito/Mockito.html%20)
  3. Downloads - <https://github.com/mockito/mockito>
  4. Maven dependency:  
     <dependency>  
      <groupId>org.mockito</groupId>  
      <artifactId>mockito-core</artifactId>  
      <version>${mockito.version}</version>  
      <scope>test</scope>  
     </dependency>
* PowerMock – создание заглушек (mock-объектов)
  1. Сайт - <https://github.com/jayway/powermock/>
  2. Wiki - <https://github.com/jayway/powermock/wiki>
  3. Downloads - <https://github.com/jayway/powermock/wiki/Downloads>
  4. Maven dependency (для интеграции с JUnit4):  
     <dependency>  
      <groupId>org.powermock</groupId>  
      <artifactId>powermock-module-junit4</artifactId>  
      <version>${powermock.version}</version>  
      <scope>test</scope>  
     </dependency>  
     <dependency>  
      <groupId>org.powermock</groupId>  
      <artifactId>powermock-api-mockito</artifactId>  
      <version>${powermock.version}</version>  
      <scope>test</scope>  
     </dependency>

Для автоматического анализа покрытия кода тестами рекомендуется использовать различные плагины для сред разработки. Например, для IntelliJ IDEA можно использовать встроенный анализатор, либо внешний – например, JaCoCo. Анализатор покрытия позволяет при запуске тестов проанализировать, по каким веткам исходного кода проходят тестовые методы, вычисляют покрытые тестами методы, определяет процент покрытия и позволяет сформировать html-отчет.

Плагины JaCoCo также можно подключить к Eclipse, Maven, Ant, Gradle, NetBeans. Существует возможность подключить анализ покрытия к инструментам непрерывной интеграции. Например, к Jenkins можно подключить упомянутую JaCoCo.

Код, покрытие которого Unit-тестами не требуется, может быть исключён из расчёта покрытия, для этого в профиль sonar-codecoverage нужно добавить секцию для конфигурирования плагина JaCoCo (org.jacoco:jacoco-maven-plugin):

    <configuration>

        <excludes>

            <exclude>\*\*/\*.class</exclude>

        </excludes>

    </configuration>

Где вместо \*\*/\*.class необходимо указать шаблон для исключения классов, покрытие по которым рассчитывать не нужно.

## Соглашения об именовании

Название тестовых классов и тестовых методов должно отражать название соответствующих им тестовых сценариев и тестовых случаев.

Для одного класса может быть несколько тестовых классов – по количеству тестовых сценариев. Каждый тестовый класс в свою очередь может содержать несколько тестовых методов – по количеству тестируемых методов и тестовых случаев.

Тестовые классы следует называть по имени тестируемого класса с суффиксом <Tests>. Если у класса несколько тестовых сценариев, то именовать тестовые классы следует по шаблону <TestScenarioName><Tests>.

Например, для класса MoonPhaseCalculator, реализующего расчет фаз Луны, тестовый класс следует называть MoonPhaseCalculatorTests. Если класс может использоваться в нескольких тестовых сценариях, то их названия будут такими MoonPhaseCalculatorScenario1Tests, MoonPhaseCalculatorScenario2Tests и т.д.

Рекомендуется каждый тестовый случай выносить в отдельный метод. Например, если пишется тест для метода, определяющего фазу Луны для даты, необходимо отдельно протестировать вызов метода для дат, соответствующих фазам роста и убывания Луны, и отдельно – для пограничных дат, соответствующих полнолунию и новолунию (фактически, полнолуние – это момент времени, целиком сутки называются полнолунием условно. То же касается новолуния).

Именовать методы при таком подходе необходимо по шаблону <MethodName>\_<StateUnderTest>\_<ExpectedBehavior>, где

* MethodName – название тестируемого метода;
* StateUnderTest – тестируемое утверждение;
* ExpectedBehavior – ожидаемый результат (поведение).

Например,

calculateMoonPhase\_ContinuosDate\_WanningCrescent() – тестирование случая для убывающей Луны;

calculateMoonPhase\_BoundaryDate\_FullMoon() – тестирование случая для полнолуния.

Техническое замечание - тесты следует создавать в тех же пакетах, что и тестируемые классы. Это облегчает тестирование методов с уровнем доступа, ограниченным классами, принадлежащими тому же пакету.

## Соглашения о документировании

Документирование тестов регламентируется общим регламентом документирования исходного кода Java. Необходимо использовать JavaDoc и комментарии для описания тестовых сценариев, методов и важных моментов алгоритма.

## Ответственность за выполнение и создание модульных тестов

Модульные тесты создаются разработчиками, изменяющими рабочий код. При создании новой функциональности разработчик обязуется одновременно реализовать юнит-тесты, покрывающие новую функциональность. В какой момент писать модульные тесты – до, после или во время создания рабочего кода – это дело вкуса разработчика или традиций, культивируемых в проекте/команде. Помимо юнит-тестов, разработчик разрабатывает прочие автоматические тесты, вспомогательные классы-заглушки и тестовые данные для новой функциональности.

Общее правило должно быть такое – в системе контроля версий не должно быть кода, не покрытого модульными тестами. В общий репозиторий рабочий код должен попадать вместе с соответствующими модульными тестами. Единственное исключение – наличие других видов автоматических тестов, если реализация модульных тестов невозможна для данного случая.

Перед выполнением коммита должен выполняться прогон всех тестов. Если в проекте не настроена непрерывная интеграция, то запуск тестов осуществляется разработчиком. Коммит кода, не прошедшего тесты успешно, запрещен. При наличии возможности инструментарий непрерывной интеграции должен выполнять рассылку, оповещающую о наличии тестов, выполняющихся неуспешно.

Если при изменении рабочего кода «ломаются» существующие тесты, разбором причин их невыполнения занимается разработчик, выполнивший изменение рабочего кода, приведшее к их неисправности. При необходимости к исправлению кода/теста привлекается автор юнит-теста, аналитик проекта, руководитель разработки.

Ответственность за автоматические запуски модульных тестов, за поддержание работы тестовой среды и системы непрерывной сборки возложена на тестировщиков проекта и администраторов тестовых стендов.

# Приложения

## Приложение 1. Пример методики разработки и тестирования

Во вложении приведен пример методики разработки и тестирования, базирующийся на реальной методике, используемой в проектах ЕФС. Методика описывает рабочий процесс разработки и тестирования программного обеспечения – от воркфлоу тикета в Atlassian Jira до ответственности участников команд. Приведены рекомендации по разработке автоматических и модульных тестов, по использованию фреймворков.



# Литература

* 1. Бек Кент. - Экстремальное программирование. Разработка через тестирование
  2. Mark Ethan Trotler – Testable Javascript
  3. Фаулер Мартин. - Рефакторинг. Улучшение существующего кода
  4. Бек Кент. – Экстремальное программирование
  5. Манифест Agile (http://agilemanifesto.org/iso/ru/manifesto.html)