Лабораторная работа №5: Модульное тестирование

Цель: Познакомится с основами написания модульных тестов, а также получить основные зания о вспомогательных инструментах, необходимых при работе с проектом: системе контроля версий Git и Maven.

План занятия:

1. Изучить теоретические сведения.
2. Выполнить практическое задание.

Теоретические сведения

Перед началом работы необходимо установить следующее:

- Open JDK (https://openjdk.java.net)
- Java IDE (рекомендуемыя IDE: IntelliJ IDEA Community)
- Git (https://git-scm.com/downloads)

Опционально (в случае если используется НЕ IntelliJ IDEA):

- Maven (https://maven.apache.org)
- Git клиент (например, SourceTree https://www.sourcetreeapp.com)

Git

Git — мощная и сложная распределенная система контроля версий. С его помощью вы можете откатиться на более старую версию вашего проекта, сравнивать, анализировать, сливать изменения и многое другое. Этот процесс называется контролем версий. Существуют различные системы для контроля версий. Вы, возможно, о них слышали: SVN, Mercurial, Perforce и другие.

Основные понятия

Локальный репозиторий - репозиторий, расположенный на локальном компьютере разработчика в каталоге.

Удалённый репозиторий - репозиторий, находящийся на удалённом сервере, в который приходят все изменения и из которого забираются все обновления.

Ветка (Branch) - это параллельная версия репозитория. Она включена в этот репозиторий, но не влияет на главную версию, тем самым позволяя свободно работать в параллельной.

Коммит (Commit) - фиксация изменений или запись изменений в репозиторий. Коммит происходит на локальной машине.

Пул (Pull) - получение последних изменений с удалённого сервера репозитория.

Пуш (Push) - отправка всех неотправленных коммитов на удалённый сервер репозитория.

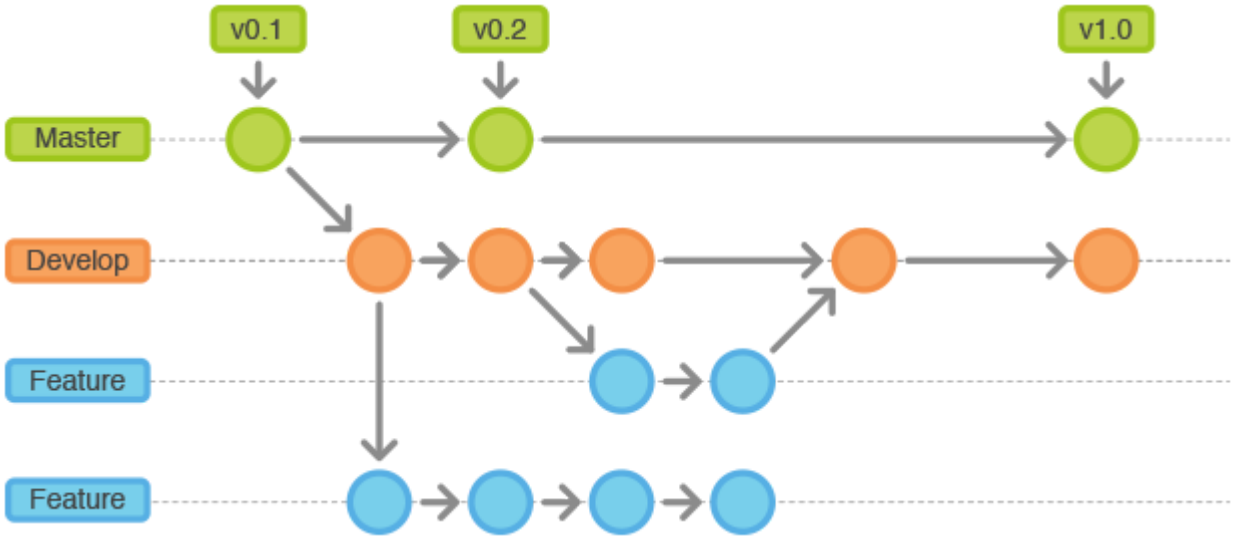
Мёрдж (Merge) - слияние изменений из какой-либо ветки репозитория с любой веткой этого же репозитория.

Пулреквест (Pull Request) - запрос на слияние веток в репозитории.

Ветвление

Во время разработки новой функциональности считается хорошей практикой работать с копией оригинального проекта, которую называют веткой. Ветви имеют свою собственную историю и изолированные друг от друга изменения до тех пор, пока вы не решаете слить изменения вместе. Это происходит по набору причин:

- Уже рабочая, стабильная версия кода сохраняется.
- Различные новые функции могут разрабатываться параллельно разными программистами.
- Разработчики могут работать с собственными ветками без риска, что кодовая база поменяется из-за чужих изменений.
- В случае сомнений, различные реализации одной и той же идеи могут быть разработаны в разных ветках и затем сравниваться.



Настройка .gitignore

В большинстве проектов есть файлы или целые директории, в которые мы не хотим (и, скорее всего, не захотим) коммитить. Мы можем удостовериться, что они случайно не попадут в коммит при помощи файла .gitignore

Создайте файл под названием .gitignore (если его еще нет) и сохраните его в директорию проекта. Внутри файла перечислите названия файлов/папок, которые нужно игнорировать, каждый с новой строки. Файл .gitignore должен быть добавлен, закоммичен и отправлен на сервер, как любой другой файл в проекте. Вот хорошие примеры файлов, которые нужно игнорировать:

- Логи
- Артефакты систем сборки (Maven)
- Папки, созданные IDE, например, Netbeans или IntelliJ

Maven

Maven - это средство для управления и сборки проектов. Он позволяет разработчикам полностью управлять жизненным циклом проекта. Благодаря этому, команда может автоматизировать процессы, связанные со сборкой, тестирование, упаковкой проекта и т.д.

В случае, когда над проектом работает несколько команд разработчиков, Maven позволяет работать в соответствии с определенными стандартами.

Преимущества Maven

- Независимость от OS. Сборка проекта происходит в любой операционной системе. Файл проекта один и тот же.
- Управление зависимостями. Редко какие проекты пишутся без использования сторонних библиотек(зависимостей). Эти сторонние библиотеки зачастую тоже в свою очередь используют библиотеки разных версий. Maven позволяет управлять такими сложными зависимостями. Что позволяет разрешать конфликты версий и в случае необходимости легко переходить на новые версии библиотек.
- Возможна сборка из командной строки. Такое часто необходимо для автоматической сборки проекта на сервере (Continuous Integration).
- Хорошая интеграция со средами разработки. Основные среды разработки на java легко открывают проекты которые собираются с помощью maven. При этом зачастую проект настраивать не нужно - он сразу готов к дальнейшей разработке.
- Как следствие - если с проектом работают в разных средах разработки, то Maven удобный способ хранения настроек. Настроечный файл среды разработки и для сборки один и тот же - меньше дублирования данных и соответственно ошибок.
- Декларативное описание проекта.

Структура и содержание проекта Maven указывается в специальном xml-файле, который называется Project Object Model (POM), который является базовым модулем всей системы.

Разработчики не обязаны указывать каждую деталь. Maven обеспечивает поведение проекта по умолчанию. Когда мы создаём проект с использованием Maven, то Maven создаёт базовую структуру проекта. Разработчики несут ответственность только за соответственное размещение файлов.

Стандартная схема Maven проекта:

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

- /src/main/java - исходный код приложения;
- /src/main/resources - файлы конфигурации и прочие ресурсы используемые в проекте;
- /src/test - непосредственно сами тесты и вспомогательные тестовые классы;
- /target - артефакты сборки проекта (например, JAR файл)

- /target/classes - скомпилированный байт-код.

pom файл

pom.xml - это основной файл, который описывает проект. Давайте разберём из чего состоит файл pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  .............
</project>
```

Корневой элемент <project>, схема, которая облегчает редактирование и проверку, и версия POM. Внутри тэга project содержится основная и обязательная информация о проекте:

```
 <!-- The Basics -->
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
```

В Maven каждый проект идентифицируется парой groupId artifactId. Во избежание конфликта имён, groupId - наименование организации или подразделения и обычно действуют такие же правила как и при именовании пакетов в Java - записывают доменное имя организации или сайта проекта. artifactId - название проекта. Внутри тэга version, как можно догадаться хранится версия проекта. Тройкой groupId, artifactId, version можно однозначно идентифицировать jar файл приложения или библиотеки. Если состояние кода для проекта не зафиксировано, то в конце к имени версии добавляется "-SNAPSHOT" что обозначает, что версия в разработке и результирующий jar файл может меняться. <packaging>...</packaging> определяет какого типа файл будет создаваться как результат сборки. Возможные варианты pom, jar, war, ear

Давайте лучше рассмотрим на примере проекта powermock-core groupId - org.powermock, artifactId - powermock-core , version - 1.4.6

Также добавляется информация, которая не используется самим мавеном, но нужна для программиста, чтобы понять, о чём этот проект:

- <name>powermock-core</name> название проекта для человека
- <description>PowerMock core functionality.</description> Описание проекта
- <url>http://www.powermock.org</url>; сайт проекта.

Зависимости - следующая очень важная часть pom.xml - тут хранится список всех библиотек (зависимостей) которые используюся в проекте. Каждая библиотека идентифицируется также как и сам проект - тройкой groupId, artifactId, version (GAV). Объявление зависимостей заключено в тэг <dependencies>...</dependencies>.

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.4</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Как вы могли заметить, при описании зависимости может присутствовать тэг <scope>. Scope задаёт, для чего библиотека используется. В данном примере говорится, что библиотека JUnit нужна только для выполнения тестов.

Maven репозитории

Репозитории - это место где хранятся артефакты: jar файлы, pom -файлы, javadoc, исходники. Существуют:

- Локальный репозиторий по умолчанию он расположен в <home директория>/.m2/repository - персональный для каждого пользователя.
- Центральный репозиторий который доступен на чтение для всех пользователей в интернете.
- Внутренний "корпоративный" репозиторий- дополнительный репозиторий, один на несколько пользователей.

Чтобы самому каждый раз не создавать репозиторий, сообщество для Вас поддерживает центральный репозиторий. Если для сборки вашего проекта не хватает зависимостей, то они по умолчанию автоматически скачиваются с http://repo1.maven.org/maven2. В этом репозитории лежат практически все опенсорсные фреймворки и библиотеки.

Самому в центральный репозиторий положить нельзя. Т.к. этот репозиторий используют все, то перед тем как туда попадают артефакты они проверяются, тем более что если артефакт однажды попал в репозиторий, то по правилам изменить его нельзя.

Для поиска нужной библиотеки очень удобно пользоваться сайтами http://mavenrepository.com/ и http://findjar.com/

TestNG

TestNG – это тестовый фреймворк, вдохновленный JUnit и NUnit, но представляющий некоторые новые функциональные возможности, которые делают его более мощным и простым в использовании.

Написание TestNG теста обычно состоит из трех этапов:

- Создайте метод в тестовом классе и пометьте его аннотацией @Test (метод должен быть публичным)
- Напишите бизнес-логику своего теста внутри созданного метода.
- Добавьте информацию о вашем тесте (например, имя класса, группы, которые вы хотите запустить) в файл testng.xml

Последний пункт является опциональным на этапе отладки\разработки тестов поскольку тестовые методи и тестовые классы можно запускать из IDE напрямую из кода. В этом случае временный testng.xml файл будет неявным образом сгенерирован автоматически.

Пример простейшего теста:

```
import org.testng.annotations.*;

public class SimpleTests {

    @Test
    public void simpleTest() {
    }
}
```

TestNG suite xml

Вот пример файла testng.xml:

```
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd" >

<suite name="Suite1" verbose="1" >
  <test name="Nopackage" >
    <classes>
       <class name="NoPackageTest" />
    </classes>
  </test>

  <test name="Regression1">
    <classes>
      <class name="test.sample.ParameterSample"/>
      <class name="test.sample.ParameterTest"/>
    </classes>
  </test>

  <test>
      <classes>
        <class name="test.IndividualMethodsTest">
          <methods>
            <include name="testMethod" />
          </methods>
        </class>
      </classes>
  </test>
</suite>
```

Концепции, используемые в этой документации, следующие:

- Файл XML представляет собой тестовый набор. Он может содержать один или несколько тестов и определяется тегом <suite>.
- Тест представляется тегом <test> и может содержать один или несколько классов TestNG.
- Класс TestNG - это класс Java, который содержит хотя бы один метод с аннотацией @Test. Он представлен тегом <class> и может содержать один или несколько тестовых методов.
- Тестовый метод - это метод Java, аннотированный @Test в исходном коде класса.

Основные аннотации TestNG

@Test: Это самая важная аннотация в TestNG, в которой находится основная логика теста. Все автоматизируемые функции находятся в методе с аннотацией **@Test**. Она имеет различные атрибуты, с помощью которых может быть настроен запуск метода.

@BeforeSuite: Аннотированный метод будет запускаться перед каждым тестовым набором определенным в testng.xml файлах.

@AfterSuite: Аннотированный метод будет запускаться после каждого тестового набора определенного в testng.xml файле.

@BeforeTest: Аннотированный метод будет запускаться перед тем каждым тестом, определённым <test> тегом в xml файле.

@AfterTest: Аннотированный метод будет запускаться после каждого теста, определёного <test> тегом в xml файле.

@BeforeClass: Аннотированный метод будет запускаться перед тем как начнётся выполнение первого тестового метода в текущем классе.

@AfterClass: Аннотированный метод будет запускаться после того как будут выполены все тестовые методы в текущем классе.

@BeforeMethod: Аннотированный метод будет запускаться перед каждым тестового метода с аннотацией @Test.

@AfterMethod: Аннотированный метод будет запускаться после каждого тестового метода с аннотацией @Test.

@DataProvider Отмечает метод как поставщик данных для тестового метода. Аннотированный метод должен возвращать двумерный массив объектов Object[][], где в каждой строке массива можно задать список параметров, которые будут переданы в тестовый метод. Тестовый метод который будет использовать заданный DataProvider должен в параметрах аннотации @Test ссылаться на DataProvider по имени.

Пример использования DataProvider-а:

```
//This method will provide data to any test method that declares that its Data Provider is named "test1"
@DataProvider(name = "test1")
public Object[][] createData1() {
 return new Object[][] {
   { "Cedric", new Integer(36) },
   { "Anne", new Integer(37)},
 };
}

//This test method declares that its data should be supplied by the Data Provider named "test1"
@Test(dataProvider = "test1")
public void verifyData1(String n1, Integer n2) {
 System.out.println(n1 + " " + n2);
}
```

Assert

Тест считается успешным, если во время его выполнения не было выброшено никаких исключений.

TestNG предоставляет специальный класс для выполнения проверок в тестах: **Assert**. Пример использования:

```
@Test
public void simpleTest() {
    Assert.assertEquals(1, 2, "Numbers are not equal.");
}
```

Методы assertEquals класса Assert принимают следующие аргументы:

- проверяемое значение;
- ожидаемое значение;
- сообщение об ошибке которое будет возвращено в случае если проверка завершится неудачей.

Если проверка заданная в Assert методе не прошла будет выкинуто исключение AssertException.

Основные методы Assert класса:

- assertEquals
- assertNotEquals
- assertTrue

- assertFalse
- assertNull
- assertNotNull
- assertSame
- Fail

Существует два типа Assert:

- Hard Assert
- Soft Assert

Когда проверка терпит неудачу Hard Assert немедленно генерирует AssertException, и выполнение тестовго сценария переключается на следующий тест. Особенность Hard Assert состоит в том что все остальные действия и проверки в тесте после неудавшейся проверки выполнены не будут В случае если это поведение не является желаемым, можно использовать Soft Assert.

Soft Assert не выбрасывает исключение сразу как только проверка терпит неудачу и будет продолжать выполнение теста в любом случае. Проверить упали ли какие-то проверки сделанные с помощью SOftAssert можно вызвав метод assertAll(). Чаще всего это делается в самом конце теста. Пример использования SoftAssert

```
@Test
public void testWithSoftAssert() {
    SoftAssert sa = new SoftAssert();

    sa.assertEquals(1, 2, "Numbers are not equal.");
    sa.assertEquals(3, 5, "These numbers are not equal too.");

    sa.assertAll();
}
```

Дополнительную информацию по TestNG фреймворку можно найти в официальной документации: https://testng.org/doc/documentation-main.html

Модульное тестирование

Модульное тестирование (Unit testing) – тестирование каждой атомарной функциональности приложения отдельно, в искусственно созданной среде. Именно потребность в создании искусственной рабочей среды для определенного модуля, требует от тестировщика знаний в автоматизации тестирования программного обеспечения, некоторых навыков программирования. Данная среда для некоторого юнита создается с помощью драйверов и заглушек.

Заглушка – часть программы, которая симулирует обмен данными с тестируемым компонентом, выполняет имитацию рабочей системы.

Заглушки нужны для:

- Имитирования недостающих компонентов для работы данного элемента.
- Подачи или возвращения модулю определенного значения, возможность предоставить тестеру самому ввести нужное значение.
- Воссоздания определенных ситуаций (исключения или другие нестандартные условия работы элемента).

Преимущества модульного тестирования:

- Модульное тестирование мотивирует программистов писать код максимально оптимизированным, проводить рефакторинг (упрощение кода программы, не затрагивая ее функциональность), так как с помощью unit тестирования можно легко проверить работоспособность рассматриваемого компонента.
- Необходимость отделения реализации от интерфейса (ввиду особенностей модульного тестирования), что позволяет минимизировать зависимости в системе.
- Документация unit тестов может служить примером «живого документа» для каждого класса, тестируемого данным способом.
- Модульное тестирование помогает лучше понять роль каждого класса на фоне всей программной системы.
- Также, при «разработке через тестирование», которая активно используется в экстремальном программировании, модульное тестирования является одним из основных инструментов, позволяющий разрабатывать модули в соответствии с требованиями к данному модулю.

Практические задания

Необходимо создать Maven проект, содержащий набор модульных тестов согласно описанным в задании условиям и залить проект на персональный репозиторий на https://github.com

Задание 1 Создать новый Maven проект и подключить к нему TestNG фреймворк используя pom файл. Написать любой простейший TestNG тест и запустить его чтобы удостовериться что проект сконфигурирован правильно.

Задание 2 Создайте персональный публичный Git репозиторий на https://github.com Залейте Maven проект из предыдущего задания в созданый репозиторий в ветку master. Создайте отдельную ветку unit_tests от ветки master и используйте unit_tests ветку для задания №3.

Задание 3 Напишите модульные тесты покрывающие проверками следующие методы класса Math из стандартной библиотеки Java:

- abs(int a)
- addExact(int x, int y)
- floorDiv(int x, int y)

Примечание: покрыть необходимо ТОЛЬКО указанные выше методы, не касаясь перегруженных методов с другими типами параметров. В качестве подсказки при написании тестов можно использовать описание функционала указанных методов представленное в javadoc для каждого метода в классе Math.

Для передачи тестовых данных в тесты необходимо использовать механизм data provider-ов, предоставляемых TestNG.

Задание 4 Создайте TestNG xml файл для запуска всех реализованных модульных тестов. Запустите полный набор тестов чтобы убедиться в их успешном выполнении.

Last updated 2020-04-20 18:54:21 +0400