Лабораторная №8

Тема: «Unit тестирование с JUnit»

Цель

Целью настоящей лабораторной работы является знакомство и освоение практических навыков unit тестирования java приложений с использованием библиотеки JUnit.

Задание

В рамках лабораторной работы требуется дополнить проект maven из прошлой лабораторной работы несколькими тестами основных методов приложения (разбор xml файла, работа с моделями, расчёт характеристик).

Содержание отчёта

Отчёт о выполнении работы должен включать в себя:

- титульный лист;
- вариант задания;
- краткое описание результатов

Теория

JUnit — библиотека для модульного тестирования программного обеспечения на языке Java.

JUnit является весьма удачным решением задач, связанных с тестированием java приложений. Растущая популярность привела к созданию подобных фреймворков для других языков. Вот некоторые из них:

1) Для C++ была реализована CPPUnit;
2) JavaScript может использоваться совместно с JSUnit;
3) Для C# разработчики создали NUnit;
4) Perl скрипты можно тестировать с помощью Test::Unit;
5) PHP код могут тестировать с помощью PHPUnit модуля;

JUnit применяется для модульного тестирования, которое позволяет проверять на правильность отдельные модули исходного кода программы. Преимущество данного подхода заключается в изолировании отдельно взятого модуля от других. При этом, цель такого метода позволяет программисту удостовериться, что модуль, сам по себе, способен работать корректно. JUnit представляет из себя библиотеку классов.

Сегодня все большую популярность приобретает test-driven development (TDD), техника разработки ПО, при которой сначала пишется тест на определённый функционал, а затем пишется реализация этого функционала. На практике все, конечно же, не настолько идеально, но

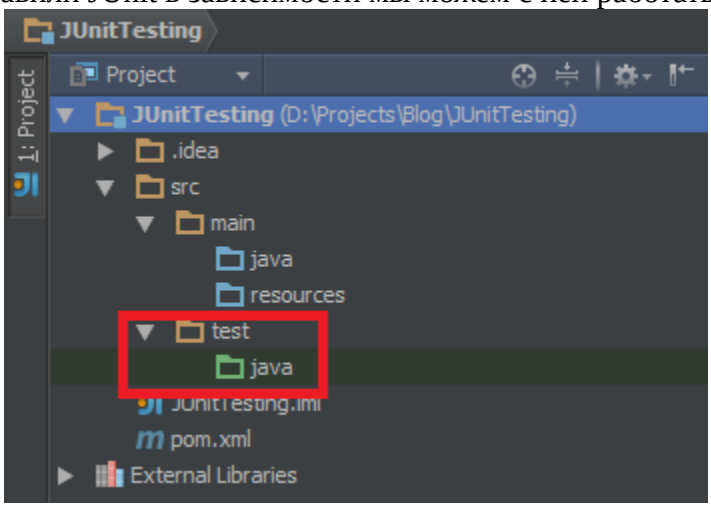
в результате код не только написан и протестирован, но тесты как бы неявно задают требования к функционалу, а также показывают пример использования этого функционала.

Пример создания теста

Для начало нам нужно подключить зависимость JUnit в pom.xml не забываем что для удобства мы используем Maven.

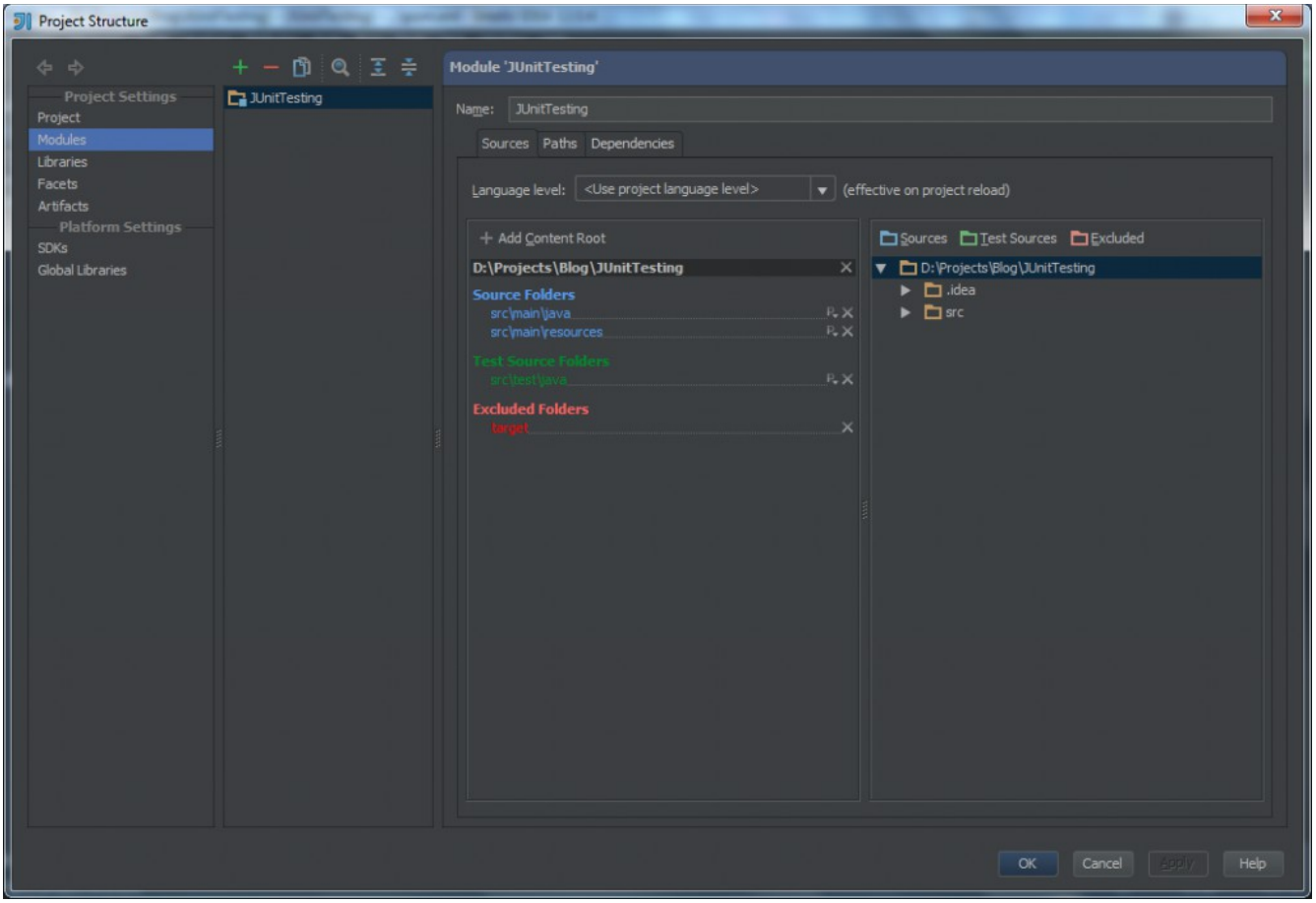
```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
```

После того как мы добавили JUnit в зависимости мы можем с ней работать.



Обязательно проверьте что папка, которая лежит в test/java должна быть зеленым цветом это будет обозначать то что в данной папке лежат тестовые классы и при сборке проекта они не будут собираться в проект.

Если же она не зеленая то заходим в Project Structure(Ctrl+Alt+Shift+S) далее выбираете слева Modules->Sources и указываете что папка test/java будет тестовым ресурсом. Пример на картинке ниже.



Допустим у нас есть класс, в котором есть метод, которые выполняет какие то действия, например суммирует какие то числа, это и будет наша логика, которую нужно протестировать.

```
public class Calculate {
    public int calA(int a, int b){
        return a+b;
    }
}
```

Как вы должны видеть этот класс когда тестируете?

1) Вы не знаете, какие манипуляции выполняют методы класса, вы видите метод и знаете что он возвращает, также вы знаете что он делает но не знаете как, а так же вы знаете что метод принимает на вход.

А если быть точней, то вот что видит тестер:

```
public int calA(int a, int b)
```

2) Вы должны передать в этот метод всевозможные данные и попытаться сделать так что бы тест завалился, это главная цель тестера.

Unit тест с технической стороны — это класс который лежит в тестовом ресурсе и который предназначен только для тестирование логики, а не для использования в production коде.

Пример JUnit теста:

```
@Test
public void testMultiply() {
   // Тестируемый класс
   MyClass tester = new MyClass();

   // Проверяемый метод
   assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));
}
```

Важно!
В JUnit предполагается, что все тестируемые методы могут быть выполнены в произвольном порядке. Поэтому тесты не должны зависеть от других тестов.

Для того чтобы указать что данные метод есть тестовым его нужно про аннотировать @Test после чего данный метод можно будет запускать в отдельном потоке для проведения тестирования.

Доступные аннотации JUnit

В следующей таблице приведен обзор имеющихся в аннотации JUnit 4.x.

Аннотация	Описание
@Test public void method()	*Аннотация **@Test** определяет что метод method() является тестовым.*
@Before public void method()	*Аннотация **@Before** указывает на то, что метод будет выполнятся перед каждым тестируемым методом **@Test**.*
@After public void method()	*Аннотация **@After** указывает на то что метод будет выполнятся после каждого тестируемого метода **@Test***
@BeforeClass public static void method()	*Аннотация **@BeforeClass** указывает на то, что метод будет выполнятся в начале всех тестов, а точней в момент запуска тестов(перед всеми тестами **@Test**).*
@AfterClass public static void method()	*Аннотация **@AfterClass** указывает на то, что метод будет выполнятся после всех тестов.*
@Ignore	*Аннотация **@Ignore** говорит, что метод будет проигнорирован в момент проведения тестирования.*
@Test (**expected** = Exception.class)	***(expected** = Exception.class) — указывает на то, что в данном тестовом методе вы преднамеренно ожидается Exception.*
@Test (**timeout**=100)	***(timeout**=100) — указывает, что тестируемый метод не должен занимать больше чем 100 миллисекунд.*

Проверяемые методы (основные)

Метод	Описание
fail(String)	*Указывает на то что бы тестовый метод завалился при этом выводя текстовое сообщение.*
assertTrue([message], boolean condition)	*Проверяет, что логическое условие истинно.*
assertsEquals([String message], expected, actual)	*Проверяет, что два значения совпадают. Примечание: для массивов проверяются ссылки, а не содержание массивов.*
assertNull([message], object)	*Проверяет, что объект является пустым **null**.*
assertNotNull([message], object)	*Проверяет, что объект не является пустым **null**.*
assertSame([String], expected, actual)	*Проверяет, что обе переменные относятся к одному объекту.*
assertNotSame([String], expected, actual)	*Проверяет, что обе переменные относятся к разным объектам.*

Для демонстрации основных возможностей этой библиотеки, можно написать примитивный класс:

```
public class Salary {

  private int value;
  private String type;

  public Salary(int v, String t){
    value = v;
    type = t;
  }

  public Salary add(Salary s){
    return new Salary(value + s.getValue(), type);
  }

  public int getValue(){
    return value;
  }
}
```

Если ранние версии jUnit (3.8 и ниже) требовали наличие класса-наследника junit.framework.TestCase, то в более поздних версиях объявление теста сводится к маркировке тестового метода аннотацией @Test. Создадим несколько тестовых методов:

```
import static org.junit.Assert.*;
```

```
import org.junit.Test;

public class TestSalary {
  @Test
  public void testAdd() {
    Salary m1 = new Salary(12, "USD");
    Salary m2 = new Salary(14, "USD");
    Salary expected = new Salary(26, "USD");
    Salary result = m1.add(m2);
    assertFalse(expected.equals(result));
  }
}
```

Метод assertFalse проверяет, является ли результат выражения в скобках неверным. При запуске теста последний пройдет успешно, т.к. результат - не равный. В классе "org.junit.Assert" предусмотрены и другие методы:

- assertEquals(int1, int2) или утверждение эквивалентности. Проверяет на равенство двух значений любого примитивного типа;
- assertFalse, assertTrue(condition) или булевые утверждения. Вместо “condition” необходимо вставить проверяемое условие;
- assertNull, assertNotNull(obj) относятся к Null утверждениям и проверяет содержимое объектной переменной на Null значение;
- assertSame(obj1, obj2) утверждение позволяет сравнивать объектные переменные.

Для каждого из assert-ов вы можете добавить первым параметром строку, которая выведется, если тест провалится: assertEquals (“Test is failed”, int1, int2).

Для аннотации @Test существуют дополнительные опции. Например:

```
@Test(expected = Exception.class)
public void testDiv(){
  Salary m1 = new Salary(12, "USD");
  Salary m2 = new Salary(0, "USD");
  int result = m1.getValue()/m2.getValue();
}
```

“expected = Exception.class” означает, что мы ждем появление исключения Exception. Если исключение не будет выброшено, то такое поведение тестируемой функции будет неверным и тест провалится.

Если какой-либо тест по какой-либо серьезной причине нужно отключить (например, этот тест постоянно валится, а исправлять его пока некогда) его можно зааннотировать с помощью @Ignore:

```
import org.junit.Ignore;

import org.junit.Test;
```

```
@Ignore
@Test(timeout = 1000)
public void testAdd() {
  //код
}
```

Также, если поместить эту аннотацию на класс, то все тесты в этом классе будут отключены. Помимо этого есть также очень интересная аннотация @Test(timeout = 1000). По истечении указанного в скобках времени, если тест не пройден, он считается неудачным. Время указывается в миллисекундах.

В jUnit для задания определенных стартовых условий вам могут пригодится т.н. фикстуры. Под этим термином следует понимать состояние среды тестирования, которое требуется для успешного выполнения тестового метода. Например, это может быть набор каких-либо объектов или состояние базы данных. Фикстуры помогают многократно использовать программный код за счет правила, которое гарантирует исполнение определенной логики до или после исполнения теста. В предшествующих версиях JUnit это правило неявно подразумевалось вне зависимости от реализации фикстур разработчиком. В версии JUnit 4 фикстуры указываются через аннотации: @Before, @After, @BeforeClass, @AfterClass.

@Before используется для выполнения множества предварительных условий перед выполнением теста. Например, если есть необходимость записать данные в БД или создать пользователя перед выполнением теста. Метод, помеченный @Before будет выполняться перед выполнением каждого теста в классе.

Метод, помеченный @After запускается после выполнения каждого теста. Например, если нужно очищать переменные после выполнения каждого теста, то этой аннотацией можно маркировать метод, имеющий необходимый код. Более того, можно маркировать одновременно несколько методов аннотациями @Before и @After. Однако, следует иметь в виду, что эти методы могут запускаться в различном порядке. Для задания многократных фикстур используются аннотации @Before и @After:

```
import org.junit.After;

import org.junit.Before;

@Before
public void setup(){
  Money m1 = new Money(12, "USD");
  Money m2 = new Money(14, "USD");
}

@After
public void setup(){
  m1 = null;
  m2 = null;
}
```

Также есть такие аннотации, как @BeforeClass, @AfterClass (т.н. однократные фикстуры). Они необходимы, если вам нужно вызвать фикстуру всего один раз. Еще jUnit предоставляет функцию параметризированного тестирования. Ознакомление с этой функцией вы можете начать здесь.

Существует альтернативный фреймворк под названием TestNG. Он разрабатывается сообществом testing.org. Во многом является аналогом jUnit.

http://Junit.org – основной сайт сообщества-основателя jUnit фреймворка

http://testng.org/doc/misc.html - документация по всем возможностям TestNG фреймворка.

http://qatestlab.com/ru/pressroom/QA-Testing-Materials/Unit-Testing-Overview/ - коротко о модульном тестировании