Testing Methodology
Assignment 1
Unit testing using JUnit

Justin Pearson
Palle Raabjerg
Farshid Hassani Bijarbooneh

Deadline: 30^{th} of November

Submission Instructions

1. You are expected to work in *pairs*.
2. To pass the Lab you need to complete both *Tutorial Exercises* and the *Assignment Tasks*.
3. You can complete the Tutorial during the Lab and present it to the instructor, but if for some good reasons you can't attend the lab then you can book a time with either *Farshid* or *Palle* to present and discuss the Tutorial exercises.
4. Once you are done with the Tutorial exercises you should complete the tasks in the assignment section of the lab, all you need to do is to follow the ToDo Checklist at the end of this assignment.
5. Archive your project folder in zip format and name it `FirstName1.LastName1-FirstName2.LastName2.zip` where the first name and last name refers to your name and your partner's name participating in the assignment.
6. Login to the course manager and submit your zip archive file before the deadline.

Introduction

In this assignment, you will work with the JUnit framework to perform Java unit testing. Junit is an open source project available at www.junit.org which is already integrated into *Eclipse*[1] 3.4 and later. We will be working with JUnit version 4.

Both Eclipse and JUnit should be installed and available on the Solaris terminals.

This Lab consists of two parts. In the first part you will learn a bit about JUnit by doing a few tutorial-like mini-exercises. In the second section you work on a set of tasks with the classes *Currency*, *Money* and *Bank*.

[1]Eclipse is an open source IDE (integrated development enviroment) for Java, available from www.eclipse.org

What is JUnit?

JUnit is a framework for testing parts, or units of your code. In general, these units are considered to be the methods of each class. JUnit can help you to make sure that each of your classes work as expected. In unit testing you will usually write one test class for each of the classes that you want to test. Your test class will often include a test method for each method implemented by the class being tested. But keep in mind that this is not always feasible, or necessary. A test case will, and should often touch on more than a single method. These tests should be written to make it likely that when all the tests pass, the code is functioning as required.

Unit testing is very common when you are doing test driven development (TDD). TDD is a methodology in which you write the test code first, and then write the code you wish to test. With JUnit, if you know what classes and methods needs to be in your system, and what their functionality should be, you can start by writing a set of test cases first, and then fill in the body of your methods to pass those test cases later. And when you later need to change your code, the test cases can be a great help in making sure that your code stays robust and relatively bug-free.

Basic Information

Before you begin, we would like to call your attention to the following conventions:

- A **Test Case Class** is named `[classname]Test.java`, where *classname* is the name of the class that is tested.
- A **Test Case Method** is a method of the Test Case Class which is used to test one or more of the methods of the target class. Test Case Methods are are annotated with `@Test` to indicate them to JUnit. Cases without `@Test` will not be noticed by JUnit.

JUnit assertions are used to assert that a condition must be true at some point in the test method. JUnit has many types of assertions. The following is a selection of the most commonly used assertions:

- assertEquals(expected, actual): Assert that expected value is equal to the actual value. The expected and actual value can be of any type, for example integer, double, byte, string, char or any Java object. If the expected and actual values are of type double or float, you should add a third parameter indicating the *delta*. It represents the maximum difference between expected and actual value for which both numbers are still considered equal.
- assertTrue(condition): Asserts that the Boolean condition is True.
- assertFalse(condition): Asserts that the Boolean condition is False.
- assertNull(object): Asserts that an object is null.
- assertNotNull(object): Asserts that an object is not null.
- assertSame(expected object, actual object): Asserts that two variables refer to the same object.

- assertNotSame(expected object, actual object): Asserts that two variables do not refer to the same object.

Whenever an assertion fails, an AssertionError is thrown, which is caught by the JUnit framework and presented as a red bar, indicating test failure. Assert statements accept an extra *message* parameter before the other parameters. This parameter is a String which will be displayed if the assertion fails.

Exceptions will also be caught by JUnit, and cause the test to fail. Except when you indicate that it is expected in the annotation, like this: `@Test(expected=SomeException.class)`. If you indicate an expected exception, the test will fail if that exception is *not* thrown.

Loading the Project

Download the file `Lab1.zip` from http://www.it.uu.se/edu/course/homepage/testmetodik/ht10/Lab
In Eclipse, choose `File -> New -> Java Project`. Give it a name ("Lab1", for instance) and click `Finish`.

To load Lab1.zip into this project, choose `File -> Import -> Archive File`, `Browse` for `Lab1.zip`, click `Finish` and answer `Yes` to letting it overwrite the `.classpath` and `.project` files (this is important, otherwise you will have to mess with setting up the classpath and source libraries yourself).

You should now have a project with a `test/` and a `src/` directory, and there should be a number of errors showing up from the `Money` and `Currency` classes in `src/b_Money`. This is supposed to happen, and if Eclipse complains about it during the tutorial exercises, you may ignore it until you get to the Money part of the Lab.

Tutorial Exercises

The package `a_Introductory` contains classes in both the `test/` and the `src/` library. The unit test classes are to be found in `test/a_Introductory/` and the classes to be tested are in `src/a_Introductory`.

Fibonacci

As the first mini-exercise, we take a look at the Fibonacci class. This class is an attempt at implementing the recursive method *fib*, which should generate the n^{th} Fibonacci Number[2].

Notice that in `FibonacciTest.java`, there is a `testFib` method corresponding to the `fib` method in the Fibonacci class. If you ever wish to create test classes, Eclipse has a wizard which will create these corresponding test methods for you.

In the test class we assert that the first 7 Fibonacci numbers must be equal to 0,1,1,2,3,5,8,13 by writing the expected value and calling the `fib` method to generate the number. If the `fib` fails at generating the right number, it will throw an AssertionError with a message showing which number it failed to generate, and the JUnit framework will show the infamous red bar of a failed test.

[2]To read about Fibonacci numbers see http://en.wikipedia.org/wiki/Fibonacci_number

Running the Test Cases

When you run a test class, JUnit will run each method annotated with @Test separately and show a green bar if all of them pass, and a red bar if any of them fail. It is important that anything happening in a test method is independent from the other test methods, otherwise you risk getting weird results. But in the case of Fibonacci, we have only one test method.
Do the following:

1. To run the test, select the `FibonacciTest.java` in the project explorer and choose `Run -> Run As -> JUnit Test`. Notice the new JUnit window appears and shows a red bar indicating that the test failed. Below the listing of the running test cases you can see a stack of all failed test cases and their messages.

2. Now, find the bug in `Fibonacci.java`, correct it and run the test again. Note with satisfaction the green bar and move on to the Quadrilateral exercise.

Quadrilateral

The Quadrilateral class defines a polygon with four sides. It has two methods, `isRectangle` and `isSquare`. Furthermore, it also uses the classes `Point`, `Line` and `Vector2D`. To find if the polygon is a rectangle, we use vectors and dot products to determine if every corner forms a right angle. To find if the polygon is a square, we use `isRectangle` and check if the lengths of all sides are equal.

Do the following:

1. Look through the test classes for `Quadrilateral`, `Point`, `Line` and `Vector2D` and read the source code for the tests.

2. Run the test cases and note that not all of them succeed. The test cases are correct, but the code has been littered with a number of bugs. Examine which test cases that failed and check the failure messages. Find and correct the bugs in the code.

Assignment Tasks

Money and Currency

You have been given a template for the `Currency` and `Money` classes, with JavaDoc comments explaining what each method should do. There is also a `Bank` and `Account` class, but we will come back to that in the following section. All the methods of `Currency.java` and `Money.java` are empty.

First, write test cases for the methods of each class, and then fill in the methods with code that will make your test cases pass. The template test methods are a guideline you can use for constructing your tests. Unless you have a good idea for how to otherwise structure the tests, you should simply fill in those template test methods.

Bear in mind that the teaching assistants have their own test cases for these methods, so write some good tests yourself, to make it likely that your code will pass our tests as well.

Important note: The decimal numbers representing money are implemented as integers. The last two digits denote two decimals. So Money(200050, SEK) will mean 2000.50 SEK.

Bank

The `Bank` and `Account` classes were written by a bad programmer. When you are confident your `Money` and `Currency` classes work as intended, write test cases for the `Bank` and `Account` classes and find the bugs. Again, the specification is provided in the JavaDoc comments.

ToDo Checklist

1. Write the code for the body of the test case methods of `MoneyTest.java` and `CurrencyTest.java`.
2. Motivate your test cases by **commenting** in the test methods.
3. Complete the methods body in `Money.java` and `Currency.java` classes, and comment where necessary. Again make sure you follow the specification for each method and implement the required functionality.
4. If you run your test cases for `MoneyTest.java` and `CurrencyTest.java` at this point, they should be all pass.
5. By following the `Bank.java` and `Account.java` specification in the JavaDoc comments, write your test cases in the `BankTest.java` and `AccountTest.java` files. Note which of your test cases fail, by commenting in the corresponding test methods.
6. Find the bugs in `Bank.java` and `Account.java` based on the JUnit failures from the previous step, and note it by placing a comment in the code where the bug was spotted. Explain how you found the bug with your unit tests.
7. Fix the bugs and verify that `BankTest.java` and `AccountTest.java` passes successfully.