JUnit 4 Tutorial

Noumbissi Noundou, Xavier
xnoumbis@uwaterloo.ca
Electrical and Computer Engineering
University of Waterloo

1 Introduction

The goal of this tutorial is to give an introduction to the **JUnit** [5, 1] framework that enables the reader to:

- Have a basic understanding of what JUnit is
- Implement test classes in JUnit 4
- Get coverage information from tests execution (with **Cobertura** [3])
- Test exceptions using JUnit

JUnit is a framework that helps in the automation of unit tests for Java classes. It was created by Kent Beck and Erich Gamma.

Similar tools for languages other than Java are **NUnit** for the .NET framework [6], **CppUnit** for C++ [4], and **PyUnit** for Python [7]. The work similarly to the JUnit framework.

Version 4 is the latest release of JUnit (as of writing). It is backward compatible with the previous version JUnit 3.8. Here, we will only present the new conventions usage of JUnit 4.

This new version was mainly released to take advantages of the annotations capabilities (JSR 175 [2]) introduced with Java 5.0.

JUnit plugins are available for several IDE like Eclipse, Netbeans, etc.

In this tutorial, we will not use any IDE for the demonstrations. We will run and collect test results from the command line.

2 Steps involved in performing unit tests

(i) Prepare (or set up) the test environment to fulfill conditions that must be met, according to the test plan. This means to define and set *prefix values*. E.g. initialize fields, turn on logging, etc.

(ii) Execute the test case. This means, executing (exercising) the part of the code to be tested. For that we use some test inputs (*test case values*), according to the test plan.

(iii) Evaluate the results or side effects generated by the execution of the test case to check if they match what has been defined in the test plan.

(iv) Clean up (or tear down) the test environment if needed so that further testing activities can be done, without being influenced by the previous test cases. We deal here with *postfix values*.

3 Realization of the unit testing steps in JUnit 4

Here we describe how the steps given in section 2 are to be implemented in the JUnit framework. The different steps are implemented in a Java test class. The methods of the test class receive specific annotations to make them known to the framework as such:

(i) Prepare (or set up) the test environment: the different actions to realize this are to be implemented in one or several methods, annotated with **@Before**. Those methods are then executed before each test case (test method).

```
@Before
public void setUp() {
    s = new Sample();
}
```

(ii) Execute the test case and evaluate the results (or side effects): This is done by exercising the code under test with test values within a (test) method. Each method annotated with **@Test** will be executed as a test case by JUnit. Evaluation of the expected results (or side effects) is done in the same method using assertions.

```
@Test
public void testAddition() {
    int a = 3, b = 6;
    int expectedOutput = (a+b);
    int res = s.Addition(a, b);
    assertEquals(expectedOutput, res);
}
```

(iii) Clean up (or tear down) the test environment is done in one or several methods, that are run after execution of each test method. To belong to this group, a method has to be annotated with **@After**.

```
@After
public void tearDown() {
    s = null;
}
```

4 Running test cases and collecting the results

The JUnit 4 framework provides the class **org.junit.runner.JUnitCore** (a so called *Test Runner*) in order to run the tests and collect their results. The tests can be run from a Java program using the method **org.junit.runner.JUnitCore.runClasses()**.

The tests can also be run from the command line with:

```
java org.junit.runner.JUnitCore TestSample
```

The results are printed to the console:

```
JUnit version 4.8.1
....I
Time: 0.006

OK (4 tests)
```

For a failing test, the following is printed:

```
JUnit version 4.8.1
..E..I
Time: 0.011
There was 1 failure:
1) testPrintAddition(TestSample)
org.junit.ComparisonFailure: expected:<[8]
> but was:<[9]
>
at org.junit.Assert.assertEquals(Assert.java:123)
...
at TestSample.testPrintAddition(TestSample.java:113)
...
at org.junit.runner.JUnitCore.main(JUnitCore.java:45)

FAILURES!!!
Tests run: 4,  Failures: 1
```

When using a plugin from within an IDE (such as Eclipse) results are generally printed on a GUI.

5 Getting coverage information

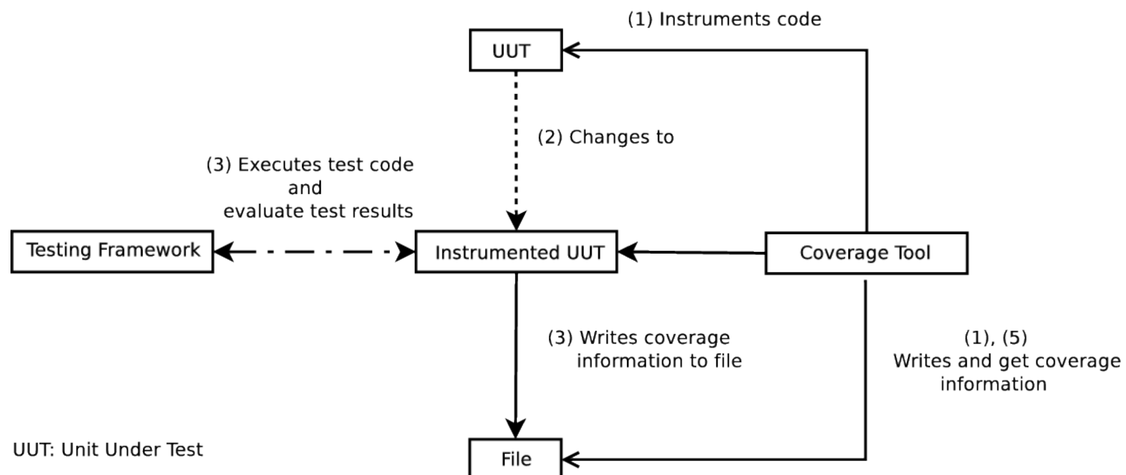

Figure 1: Coverage Tool in the Unit Testing process

Generally, one may want to gain information about parts of the Unit Under Test that was exercised. For that you may want to use a tool to get coverage information.

Figure 1 illustrates this process.

Coverage tools instrument (add instructions to the unit under test to gather coverage information) your code before you execute the test cases. Then, when the test cases are run, information about what part of the code was executed is written to a file (created by the tool).

Then you can call the coverage tool to visualize parts of the code that were executed. These tools usually have a graphic user interface.

In the following we present the coverage tool **Cobertura**.

5.1 Cobertura

Cobertura is a coverage tool for Java. Here are the features from Cobertura from the tool website (***cobertura.sourceforge.net***) :

- Can be executed from ant or from the command line
- Instruments Java bytecode after it has been compiled

- Can generate reports in HTML or XML
- Shows the percentage of lines and branches covered for each class, each package, and for the overall project
- Shows the McCabe cyclomatic code complexity of each class, and the average cyclomatic code complexity for each package, and for the overall product
- Can sort HTML results by class name, percent of lines covered, percent of branches covered, etc. It can sort in ascending or decending order

Steps in using Cobertura with JUnit from the command line:

(i) Compile your java classes:

```
javac Sample.java TestSample.java
```

(ii) Instrument the class files of the code you want to gather coverage information:

```
cobertura-instrument.sh Sample.class
```

(iii) Execute your unit tests:

```
java org.junit.runner.JUnitCore TestSample
```

(iv) Generate coverage information:

```
cobertura-report.sh --destination <report directory>
```

This action generates HTML files in the folder <report directory>, that contains tests coverage information.

5.2 Installing Cobertura

To use Cobertura, download its latest distribution file from ***cobertura.sourceforge.net*** . After unpacking the archive, the location of the folder has to be included in the executable path.

For example on Linux based machines using a bourne shell:

```
export PATH=<Cobertura directory>:$PATH
```

For how to use other features and commands of Cobertura, please refer to the tool website.

6 Testing Exception

For testing Exceptions thrown by a method, we present here two methods. The first one was created with JUnit 4. Using it requires use of the annotation **@Test** with the

attribute **expected**:

```
@Test(expected=ArithmeticException.class)
public void testDivideByZero() {
    s.divideByZero();
}
```

However, if the developer wants to access details information about the exception instance, it is more suitable to use the old method from JUnit 3. In that case the method **org.junit.Assert.fail()** can be used to trigger failure of a test. This can be for instance desirable in cases where the expected exception was not thrown.

```
@Test
public void testAnotherException() {
    try{
        s.anotherException(0);
        //With the method fail, we can signalize to
        //the framwork that the test failed.
        fail("My Exception was not thrown!");
    }
    catch(MyException e) {
        //Here we have access to more info
        assertNotNull(e.getMessage());
    }
}
```

7 Ignoring a Test

JUnit gives also the possibility not to execute a test method. For that, the method has to be annotated with **@Ignore**. This may be desirable for instance in cases where a test method is incomplete.

```
@Ignore("Not Ready, to be changed")
@Test
public void doNothing() {
}
```

8 Building Test Suites

To group test cases into test suites, a suite class has to be defined. The annotation **@Suite.SuiteClasses** is used to specify the test classes to be incorporated to the suite.

```java
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({TestSample.class, TestOther.class})
public class MySuite {
}
```

9 Installing JUnit

For running JUnit, you need to download the latest jar file from ***junit.org*** and include it in your classpath. As of of writing, this file is **junit-4-8-1.jar**.

JUnit can also be installed from most Linux distributions package manager. For instance, on Debian based distributions you can use the command:

`apt-get install junit4`

to get JUnit 4.

You should always be careful which version the package manager is actually installing. Using for example junit instead of junit4 with Debian (squeeze) will install version 3 of the JUnit framework.

References

[1] Elliotte Rusty Harold. An early look at junit4. http://www.ibm.com/developerworks/java/library/j-junit4.html, call on Jan. 14 2010.

[2] Java Community Process. Java community process website. http://www.jcp.org/en/jsr/detail?id=175, call on Jan. 14 2010.

[3] Cobertura team. Cobertura website. http://cobertura.sourceforge.net, call on Jan. 14 2010.

[4] CppUnit team. Cppunit website. http://cppunit.sourceforge.net, call on Jan. 14 2010.

[5] JUnit team. Junit website. http://www.junit.org, call on Jan. 14 2010.

[6] Nunit team. Nunit website. http://www.nunit.org, call on Jan. 14 2010.

[7] PyUnit team. Pyunit website. http://pyunit.sourceforge.net, call on Jan. 14 2010.