ZJU Software Testing – Practice with Junit and TestNG

1. Junit Assert Methods

JUnit is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks. JUnit provides overloaded assertion methods for all primitive types and Objects and arrays (of primitives or Objects). The parameter order is expected value followed by actual value. There are a number of Junit assert methods and the examples here will illustrate them. First, create a new project, say JavaAssert, a new package called unitTests, and a new junit test class called testAssertions().

The framework code looks like

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

public class testAssertions {

	@Test
	public void test() {
		fail("Not yet implemented");
	}
}
```

Type each of these code snippets in the place marked '`fail("Not yet implemented");`'. Then run the junit test to see if it is a pass or fail as you'd expect.

int value1=5;
int value2=6;
assertEquals(value1,value2);

int value1=5;
int value2=5;
assertEquals(value1,value2);

int value1=5;
int value2=6;
assertTrue(value1<value2);

int value1=5;
int value2=6;
assertFalse(value1>value2);

int value1=5; assertNotNull(value1);
String string1=”ABC”; assertNotNull(string1);
String string1=null; assertNull(string1);

String string1=”ABC”; String string2=”ABC”; *assertSame*(string1,string2);
String string1=”ABC”; String string2=null; *assertNotSame*(string1,string2);
int[] array1={2,3,4,5}; int[] array2={2,3,4,5}; *assertArrayEquals*(array1,array2);

2. Making a Test suite to run a number of Junit tests all together

The idea here is to be able to run a number of unit tests at the same time, essentially automating the testing process, over typing in the code for each test one at a time and running it. If one fails then we’d like to be able to identify which one. We will use the Program Grade as the example for this (see below for the code).

```
public class Grades {
	public static String Grade (int exam, int course) {
		long average;
		average = Math.round((exam+course)/2);
		String result="null";
		if ((exam<0) || (exam>100) || (course<0) || (course>100))
		result="Marks out of range";
		else {
		if ((exam<50) || (course<50)) {
		result="Fail";
		}
		else if (exam < 60) {
		result="Pass,C";
		}
		else if ( average >= 70) {
		result="Pass,A";
		}
		else {
		result="Pass,B";
		}
		}
		return result;
		}
}
```

Firstly, we will create separate class files for each unit test we want to run. We'll call them JunitGradeTest1 and JunitGradeTest2. The code for these is given below.

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

public class JunitGradesTest1 {
	@Test
	public void testGrade1() {
		Grades g = new Grades();
		String mark= g.Grade(-1, 50);
		assertEquals("Marks out of range",mark);

	}
}
```

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

public class JunitGradesTest2 {
	@Test
	public void testGrade2() {
		Grades g = new Grades();
		String mark= g.Grade(50, 50);
		assertEquals("Pass,C",mark);

	}
}
```

Then, we create a test suite class that references both unit test classes that we want to carry out. They are given in the code below:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
   JunitGradesTest1.class,
   JunitGradesTest2.class,
})
public class JunitTestSuite {
}
```

Lastly, we make a testrunner class that will run the tests and report as to whether they have all been successful, or whether any of the tests have failed. Notice that this class contains a main method.

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(JunitTestSuite.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
```

```
        System.out.println(result.wasSuccessful());
    }
}
```

If we run this we should see the console display

```
true
```

Let's make a third unit test but this time we'll select the input values that do not correspond to the expected output

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

public class JunitGradesTest3 {
        @Test
        public void testGrade3() {
                Grades g = new Grades();
                String mark= g.Grade(90, 40);
                assertEquals("Pass,A",mark);
        }
}
```

The testsuite code should then be updated

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
    JunitGradesTest1.class,
    JunitGradesTest2.class,
    JunitGradesTest3.class,
})
public class JunitTestSuite {
}
```

If we run our testrunner program again then we get the output

```
testGrade3(ut2.JunitGradesTest3): expected:<[Pass,A]> but was:<[Fail]>
false
```

It notes that not all the tests passed and lists the unit test that returned the incorrect response. We can continue to add more tests to the suite then as we wish.

3. Code Coverage with EclEmma

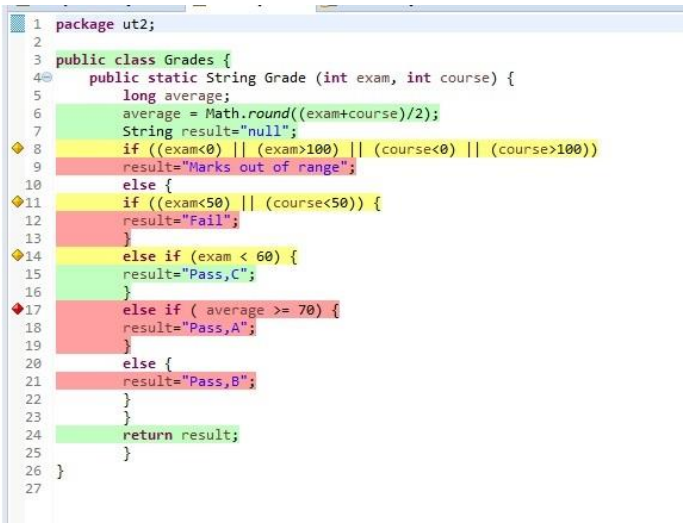
If we want to see how our code is covered by the test data we can use the EclEmma tool to help us.. Writing a simple test for the grade program as below:

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

public class TestGrade {
        @Test
```

```
        public void testGrade() {
                Grades g = new Grades();
                String mark= g.Grade(50, 50);
                assertEquals("Pass,C",mark);

}
}
```

EclEmma adds a so called launch mode to the Eclipse workbench. It is called Coverage mode and works exactly like the existing Run and Debug modes. The Coverage launch mode can be activated from the Run menu or the workbench's toolbar:

If you chose Coverage as…. and then Junit test, the code will execute and you will find the code is highlighted in various colours.



```
1  package ut2;
2
3  public class Grades {
4      public static String Grade (int exam, int course) {
5          long average;
6          average = Math.round((exam+course)/2);
7          String result="null";
8          if ((exam<0) || (exam>100) || (course<0) || (course>100))
9          result="Marks out of range";
10         else {
11         if ((exam<50) || (course<50)) {
12         result="Fail";
13         }
14         else if (exam < 60) {
15         result="Pass,C";
16         }
17         else if ( average >= 70) {
18         result="Pass,A";
19         }
20         else {
21         result="Pass,B";
22         }
23         }
24         return result;
25         }
26 }
27
```

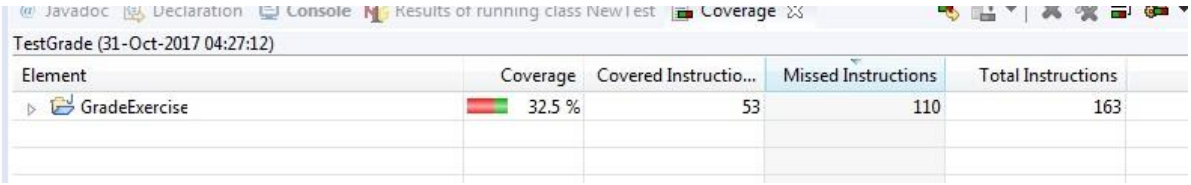
Source lines containing executable code get the following colour code:
green for fully covered lines,
yellow for partly covered lines (some instructions or branches missed) and
red for lines that have not been executed at all.

In addition coloured diamonds are shown at the left for lines containing decision branches. The colours for the diamonds have a similar semantic than the line highlighting colours:

green for fully covered branches,
yellow for partly covered branches and
red when no branches in the particular line have been executed.

The source annotations automatically disappear when you start editing a source file or delete the coverage session.

Additionally in the console it will give you a Coverage percentage as shown in the following figure.



4. Using TestNG

TestNG is similar to Junit but introduces many new, innovative, functionalities such as dependency testing, grouping concept to make testing more powerful and easier to do. The NG stands for Next Generation. A few of the features that TestNG has over JUnit 4 are:

Extra Before and After annotations such as Before/After Suite and Before/After Group
Dependency test
Grouping of test methods
Multithreaded execution
In-built reporting framework

In TestNG, suites and tests are configured or described mainly through XML files. By default, the name of the file is testng.xml. The assertion methods provided by TestNG allows the writing of tests, they are actually derived from JUnit. In order to use the assertion methods, the test class must import the library: import static org.testng.AssertJUnit.*;
The most popular methods: assertTrue, assertEquals, assertNull

To create a Java project with TestNG dependencies. We first create our Java project and then under Project->Properties->Java Build Path we add the library for TestNG

If we include our Grades class in the project then as in the previous example, when we want to write a TestNG test we select New->Other->TestNGClass and give it an appropriate name. This will create the TestNG class in the project. We can add our test code as shown below

```
import org.testng.annotations.Test;
import org.testng.annotations.BeforeMethod;
import org.testng.Assert;
import org.testng.annotations.AfterMethod;

public class NewTest {
  @Test
  public void f() {
//Test Code Added in Here
      Grades g = new Grades();
      String mark= g.Grade(50, 50);
      Assert.assertEquals(mark, "Pass,C");
  }
  @BeforeMethod
  public void beforeMethod() {
  }

  @AfterMethod
  public void afterMethod() {
  }
}
```

All we need to do then is run it. The results of the test wil be displayed in the console window. If you go to the test-output folder you will see an xml file testing-results.xml which holds a report of the results.

Exercise

To fully test the Grade program suing Boundary Value analysis the Test Cases and Test Data are given as below. Use this information to make out the proper test suite using Junit or TestNG.

Grade Program Boundary Value Analysis

Input Boundary values

Parameter	Test Case	Partition Boundary value
Exam	1*	INT_MIN
	2*	-1
	3	0
	4	49
	5	50
	6	59
	7	60
	8	100
	9*	101
	10*	INT_MAX
Course	11*	INT_MIN …
	12*	-1
	13	0
	14	49
	15	50
	16	100
	17*	101
	18*	INT_MAX

Output Boundary values

Parameter	Test Case	Partition Boundary value
Grade	19	'Marks out of range'
	20	'Fail'
	21	'Pass,C'
	22	'Pass,B'
	23	'Pass,A'

Each Boundary value is a test case

Test Data

Test No.	Test Cases/Boundaries Covered	Inputs		Expected Outputs
		exam	*course*	*Result*
1	3, 15, 20	0	50	Fail
2	4, 15, 20	49	50	Fail
3	5, 15, 21	50	50	Pass, C
4	6, 15, 21	59	50	Pass, C
5	7, 15, 22	60	50	Pass, B
6	8, 15, 23	100	50	Pass, A
7	5, 13, 20	50	0	Fail
8	5,14, 20	50	49	Fail
9	5, 15, 21	50	50	Pass, C
10	5, 16, 23	50	100	Pass, A
11*	1, 15, 19	INT_MIN	50	Marks out of Range
12*	2, 15, 19	-1	50	Marks out of Range

13*	9, 15, 19	101	50	Marks out of range
14*	10, 15, 19	INT_MAX	50	Marks out of range
15*	5, 11, 19	50	INT_MIN	Marks out of Range
16*	5, 12, 19	50	-1	Marks out of range
17*	5, 17, 19	50	101	Marks out of range
18*	5, 18, 19	50	INT_MAX	Marks out of Range