Java Programming

How to use JUnit for Java Unit Testing (in Eclipse and Ant)

Introduction to Unit Testing

Unit Testing is concerned about testing *individual* programs to ascertain that each program runs as per specification. Prior to systematic unit testing framework, programmers tends to write test expressions which print to the console or a trace file (the amount of output is sometimes controlled by a trace-level or debug-level flag). This approach is not satisfactory because it requires human judgment to analyse the results produced. Too many print statements cause the dreaded *Scroll Blindness*.

JDK 1.4 provides an assertion feature (read Assertion), which enables you to *test your assumptions* about your program logic (such as pre-conditions, post-conditions, and invariants). Nonetheless, assertion is primitive compared with the unit testing framework.

With a proper Unit Testing framework, you can automate the entire unit testing process. Your job becomes designing proper test cases to excite the program. Furthermore, the unit testing process can be integrated into the build process. In this case, the build process not only checks for syntax errors, but semantic errors as well.

Extreme programming (www.xprogramming.com) advocates "write test first, before writing codes".

JUnit

JUnit is a Java *Unit Testing Framework*. It is the *de facto* standard for Java Unit Testing. JUnit is an open-source project @ http://www.junit.org. JUnit is not included in JDK, but included in most of the IDEs such as Eclipse and NetBeans.

Installing JUnit: Goto www.junit.org ⇒ "Download JUnit" ⇒ "`junit4.x.x.zip`" (which includes executable jar files, source, and documentation), or "`junit4.x.x.jar`" (if you only want the executable) ⇒ Unzip.

Using JUnit: To use the JUnit, include JUnit jar-file "`junit4.x.x.zip`" in your `CLASSPATH`.

```
// set CLASSPATH for this command session only (set permanently via control panel)
prompt> set CLASSPATH=.;$JUNIT_HOME\junit4.x.x.jar
// You can also compile/run with classpath option
prompt> javac -classpath .;$JUNIT_HOME\junit4.x.x.jar test.java
prompt> java -classpath .;$JUNIT_HOME\junit4.x.x.jar test
```

Read: Read "Getting Started" at www.junit.org. The JUnit's "doc" directory contains the documentation, and "junit" contains samples.

JUnit 4

There are two versions of JUnit, version 3 and version 4, which are radically different. JUnit 4 uses the *annotation* feature (since JDK 1.5) to streamline the process and drop the strict naming conventions.

Let's Get Started with an Example

The following program uses `static` methods to operate on two numbers.

```
public class StaticCalculation {
```

```
   public static int add(int number1, int number2) {
      return number1 + number2;
   }

   public static int sub(int number1, int number2) {
      return number1 - number2;
   }

   public static int mul(int number1, int number2) {
      return number1 * number2;
   }

   public static int div(int number1, int number2) {
      return number1 / number2;
   }
}
```

Let's use JUnit to write test cases to test all the methods.

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

public class StaticCalculationTest {

   @Test
   public void addSubTest() {
      // assertEquals(String message, long expected, long actual)
      assertEquals("Error in add()!", 3, StaticCalculation.add(1, 2));
      assertEquals("Error in sub()!", 1, StaticCalculation.sub(2, 1));
   }

   @Test
   public void mulDivTest() {
      assertEquals("Error in mul()!", 6, StaticCalculation.mul(2, 3));
      // assertEquals(String message, double expected, double actual, double delta)
      assertEquals("Error in div()!", 0.5, StaticCalculation.div(1, 2), 1e-8);
   }
}
```

Methods marked by annotation `@org.junit.Test` are JUnit's test cases. Inside the test cases, we could use `assertEquals()`, `assertTrue()`, `assertFalse()`, `assertNull()`, `assertNotNull()`, `assertSame()`, `assertNotSame()`, `assertThat()` to compare the actual result of an operation under test with an expected result. They are all static methods in `org.junit.Assert` class.

Take note that we separate 4 assertions in two test cases (for illustration). For the last assertion for `div()`, we mistakenly assume that 1/2 = 0.5.

JUnit provides a console-version of test-runner `org.junit.runner.JUnitCore` for you to run the tests under command shell, with the following syntax:

```
prompt> java org.junit.runner.JUnitCore TestClass1 [TestClass2 ...]
```

You can also run the test from you Java program, using this statement:

```
org.junit.runner.JUnitCore.runClasses(TestClass1.class, ...);
```

For example, to run our test program:

```
// Set CLASSPATH to include JUnit
prompt> set classpath=.;D:\xxxxxx\junit-4.8.2.jar
// Run test cases, using the JUnit console run-runner JUnitCore
prompt> java org.junit.runner.JUnitCore StaticCalculationTest
JUnit version 4.8.2
..E
Time: 0.008
```

```
There was 1 failure:
1) mulDivTest(StaticCalculationTest)
java.lang.AssertionError: Error in div()! expected:<0.5> but was:<0.0>
......
FAILURES!!!
Tests run: 2,  Failures: 1
```

Observe that the unit test caught the error 1/2 = 0.5. In this case, it could be an error in the test case, or an incorrect assumption in the program under test.

JUnit under Eclipse

To create a test case under Eclipse: "File" ⇒ "New" ⇒ "JUnit Test Case" ⇒ the "New JUnit Test Case" dialog appears. Select "New JUnit 4 Test", and enter the class name StaticCalculationTest.

To run the test: Right-click ⇒ "Run As" ⇒ "JUnit Test". The results are shown in the JUnit console.

Integrating Test Cases into Source Code

Instead of writing a separate class for test cases, we can integrated them into the source code (not sure whether recommended, as you might not want to include the test cases in production). For example,

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

public class StaticCalculation {

   public static int add(int number1, int number2) {
      return number1 + number2;
   }

   public static int sub(int number1, int number2) {
      return number1 - number2;
   }

   public static int mul(int number1, int number2) {
      return number1 * number2;
   }

   public static int div(int number1, int number2) {
      return number1 / number2;
   }

   public static void main(String[] args) {
      System.out.println("1/2 is " + div(1, 2) + ", and is not 0.5");
   }

   // Integrate test case inside the source
   @Test
   public void unitTest() {
      // assertEquals(String message, long expected, long actual)
      assertEquals("Error in add()!", 3, StaticCalculation.add(1, 2));
      assertEquals("Error in sub()!", 1, StaticCalculation.sub(2, 1));
      assertEquals("Error in mul()!", 6, StaticCalculation.mul(2, 3));
      // assertEquals(String message, double expected, double actual, double delta)
      assertEquals("Error in div()!", 0.5, StaticCalculation.div(1, 2), 1e-8);
   }
}
```

You can run the above program normally, starting at main(); or use test-runner to run the test cases.

Test Methods and Test Fixtures

- **Test Methods**: A test methods is annotated with `@Test`. Inside a test method, you can use a variation of the static assert method from `org.junit.Assert` (e.g., `assertTrue()`, `assertFalse()`, `assertEquals()`) to compare the expected and actual results.
- **Test Fixture**: The set of objects that a test method operates on. You declare these objects as private variables. You could initialize them via a public method annotated with `@Before`, and perform clean-up with a public method annotated with `@After`. Each test method runs on its own set of text fixtures. This ensures isolation between the test methods. Two other method annotations, `@BeforeClass` and `@AfterClass`, are available, which run the annotated method *once* before the class and after the class.

Let's rewrite our earlier example in object instead of static methods, to illustrate the test fixture.

```
public class MyNumber {

   private int number;

   public MyNumber(int number) {   // Constructor
      this.number = number;
   }

   public int getNumber() {
      return number;
   }

   public MyNumber add(MyNumber another) {
      return new MyNumber(this.number + another.number);
   }

   public MyNumber sub(MyNumber another) {
      return new MyNumber(this.number - another.number);
   }
}
```

The JUnit test program is as follows:

```
import static org.junit.Assert.assertEquals;
import org.junit.Before;
import org.junit.Test;

public class MyNumberTest {
   private MyNumber number1, number2;   // Test fixtures

   // Initialize the test fixtures for EACH test
   @Before
   public void setup() {
      number1 = new MyNumber(2);
      number2 = new MyNumber(1);
   }

   @Test
   public void addTest() {
      assertEquals("Error in add()!", 3, number1.add(number2).getNumber());
   }

   @Test
   public void subTest() {
      assertEquals("Error in sub()!", 1, number1.sub(number2).getNumber());
   }
}
```

Again, you could use the JUnit console test-runner or Eclipse to run the tests.

Another Example:

```
import org.junit.*;
import java.util.ArrayList;
```

```
import org.junit.runner.Result;

public class ArrayListTest {
   private ArrayList<String> lst;   // Test fixtures

   // Initialize test fixtures before each test method
   @Before
   public void init() throws Exception {
      lst = new ArrayList<String>();
      lst.add("alpha");    // at index 0
      lst.add("beta");     // at index 1
   }

   // Test method to test the insert operation
   @Test
   public void insertTest() {
      // assertEquals(String message, long expected, long actual)
      Assert.assertEquals("wrong size", 2, lst.size());
      lst.add(1, "charlie");
      Assert.assertEquals("wrong size", 3, lst.size());
      // assertEquals(String message, Object expected, Object actual)
      // Use String.equals() for comparison
      Assert.assertEquals("wrong element", "alpha",   lst.get(0));
      Assert.assertEquals("wrong element", "charlie", lst.get(1));
      Assert.assertEquals("wrong element", "beta",    lst.get(2));
   }

   // Test method to test the replace operation
   @Test
   public void replaceTest() {
      Assert.assertEquals("wrong size", 2, lst.size());
      lst.set(1, "charlie");
      Assert.assertEquals("wrong size", 2, lst.size());
      Assert.assertEquals("wrong element", "alpha",   lst.get(0));
      Assert.assertEquals("wrong element", "charlie", lst.get(1));
   }

   public static void main(String[] args) {
      Result r = org.junit.runner.JUnitCore.runClasses(ArrayListTest.class);
      System.out.println(r.wasSuccessful());
   }
}
```

To run the test, you can either include a `main()` method as above, or use the command-line.

JUnit Package org.junit

The core package for JUnit 4 is `org.unit`, which is simple and elegantly designed.

- `Assert` class: contains `static` methods `assertEquals()`, `assertTrue()`, `assertFalse()`, `assertNull()`, `assertNotNull()`, `assertSame()`, `assertNotSame()`, `assertThat()`, `assertArrayEquals()`.
- `Assume` class: contains `static` methods `assumeTrue()`, `assumeNotNull()`, `assumeThat()`, `assumeNoException()`.
- `@Test`: mark the method as a test method.
- `@Test(expected=IOException.class)`: The test is expected to trigger this exception.
- `@Test(timeout=1000)`: Treat the test as fail, if it exceeds the specified milliseconds.
- `@Before` and `@After`: mark the method as to be run before and after EACH test method, for initializing and cleaning-up test fixtures.
- `@BeforeClass` and `@AfterClass`: mark the method as to be run before and after ONCE for the class.
- `@Ignore`: ignore this test method (annotated with `@Test`). For example,

```
@Ignore("Under Construction")
@Test
public void someTest() {
   ......
}
```

- @Rule:

Writing Generic Tests with Parameters

[TODO]

Writing Test Cases

How to test a program to ensure correctly? There are two techniques: white-box testing and black-box testing. White-box testing inspect the program codes and test the program logic. Black-box testing does not inspect the program codes, but looking at the input-output, treating the program as a black box.

For black-box testing, the most common approach is to partition the inputs, and design test cases for each input partition. The test cases could test on a typical input value as well as the boundaries.

For example, the following program converts a given mark (0-100) to a letter grade ('A' to 'F'). There is a logical error in the program.

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

public class StaticGrade {

   // Convert given mark into letter grade
   // Assume that mark is between 0 and 100 (inclusive)
   public static char getLetterGrade(int mark) {
      // Assert is disabled by default,
      // To enable, run with option -enableassertions (or -ea)
      assert (mark >= 0 && mark <= 100) : "mark is out-of-range: " + mark;

      if (mark >= 75) {
         return 'A';
      } else if (mark >= 60) {
         return 'B';
      } else if (mark > 50) {   // an logical error here
         return 'C';
      } else {
         return 'F';
      }
   }

   // Test a typical value in each partition
   @Test
   public void testTypical() {
      assertEquals("wrong grade", 'A', StaticGrade.getLetterGrade(95));
      assertEquals("wrong grade", 'B', StaticGrade.getLetterGrade(72));
      assertEquals("wrong grade", 'C', StaticGrade.getLetterGrade(55));
      assertEquals("wrong grade", 'F', StaticGrade.getLetterGrade(30));
   }

   // Test the boundaries of the each partition
   @Test
   public void testBoundaries() {
      assertEquals("wrong grade", 'A', StaticGrade.getLetterGrade(75));
      assertEquals("wrong grade", 'A', StaticGrade.getLetterGrade(100));
      assertEquals("wrong grade", 'B', StaticGrade.getLetterGrade(60));
      assertEquals("wrong grade", 'B', StaticGrade.getLetterGrade(74));
```

```
      assertEquals("wrong grade", 'C', StaticGrade.getLetterGrade(50));
      assertEquals("wrong grade", 'C', StaticGrade.getLetterGrade(59));
      assertEquals("wrong grade", 'F', StaticGrade.getLetterGrade(0));
      assertEquals("wrong grade", 'F', StaticGrade.getLetterGrade(49));
   }
}
```

Try to run the above tests to find the logical error. Take note that `assertEquals()` does not accept `char` as arguments, but upcast to `long`. That is, the output show the `char`'s numeric code.

Automating Unit Tests

You can automate many unit tests via the Apache ANT Builder Tool (Recommended).

You can also program and run a test suite (in JUnit 3.8) as follows:

1. For each test case to be included in the test suite, make your JUnit 4 class accessible to a `TestRunner` designed to work with JUnit 3.8, by declaring a `static` method `suite()` that returns a `test`.

   ```
   // To use JUnit 3.8 test suite
   public static Test suite() {
      return new JUnit4TestAdapter(Test.class);
   }
   ```

2. Write a test suite to include many test classes:

   ```
   import junit.framework.Test;
   import junit.framework.TestSuite;

   public class AllTests {
      public static Test suite() {
         TestSuite suite = new TestSuite(AllTests.class.getName());
         // Call the static method suite() in each case test
         suite.addTest(TestClass1.suite());
         suite.addTest(TestClass2.suite());
         return suite;
      }
   }
   ```

   You can use the test-runner to run the test suite, as before.

JUnit 3.8

JUnit 3.8, which uses *strict naming convention* to denote various entities, is probably outdated. I suggest that you move to JUnit 4, which is more intuitive by using annotation.

Let's begin with an Example

Below is a Java program to be tested. Note that there is a logical error in the program.

```
1  public class Grade {
2     public static char getLetterGrade(int mark) {
3        // assume that mark is between 0 and 100 (inclusive)
4        assert (mark >= 0 && mark <= 100) : "mark is out-of-range: " + mark;
5        if (mark >= 75) {
6           return 'A';
7        } else if (mark >= 60) {
8           return 'B';
9        } else if (mark > 50) {    // an logical error here
10          return 'C';
11       } else {
12          return 'F';
13       }
14    }
```

```
15   }
```

The unit-test program (using JUnit framework) is as follows. Black-box test cases are set up to test typical values as well as boundary values.

```
 1   import junit.framework.Test;
 2   import junit.framework.TestCase;
 3   import junit.framework.TestSuite;
 4
 5   public class GradeUnitTest extends TestCase {
 6
 7      public GradeUnitTest(String name) { super(name); }
 8      protected void setUp() throws Exception { super.setUp(); }
 9      protected void tearDown() throws Exception { super.tearDown(); }
10
11      public void testTypical() {      // test a typical value in partitions
12         assertEquals("wrong grade", 'A', Grade.getLetterGrade(95));
13         assertEquals("wrong grade", 'B', Grade.getLetterGrade(72));
14         assertEquals("wrong grade", 'C', Grade.getLetterGrade(55));
15         assertEquals("wrong grade", 'F', Grade.getLetterGrade(30));
16      }
17
18      public void testBoundaries() {   // test the boundaries of the partitions
19         assertEquals("wrong grade", 'A', Grade.getLetterGrade(75));
20         assertEquals("wrong grade", 'A', Grade.getLetterGrade(100));
21         assertEquals("wrong grade", 'B', Grade.getLetterGrade(60));
22         assertEquals("wrong grade", 'B', Grade.getLetterGrade(74));
23         assertEquals("wrong grade", 'C', Grade.getLetterGrade(50));
24         assertEquals("wrong grade", 'C', Grade.getLetterGrade(59));
25         assertEquals("wrong grade", 'F', Grade.getLetterGrade(0));
26         assertEquals("wrong grade", 'F', Grade.getLetterGrade(49));
27      }
28
29      public static Test suite() {   // For putting into a TestSuite.
30         return new TestSuite(GradeUnitTest.class);
31      }
32
33      public static void main(String[] args) {
34         junit.textui.TestRunner.run(GradeUnitTest.class);
35      }
36   }
```

Compile and execute the program (with JUnit JAR file included in the `CLASSPATH`) as follows. Note that one of the unit-test cases catches the logical error.

```
> javac -cp .;c:\junit\junit-3.8.2.jar GradeUnitTest.java
> java -cp .;c:\junit\junit-3.8.2.jar GradeUnitTest
```

```
..F
Time: 0.006
There was 1 failure:
1) testBoundaries(GradeUnitTest)junit.framework.AssertionFailedError: wrong grade expected:<C> but was:<F>
        at GradeUnitTest.testBoundaries(GradeUnitTest.java:23)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at GradeUnitTest.main(GradeUnitTest.java:34)
FAILURES!!!
Tests run: 2,  Failures: 1,  Errors: 0
```

JUnit Terminology

- Class `TestCase`: A class that contains *test methods* should derive from this superclass. Each `TestCase` can include many *test methods*.

- Test Methods: A test methods must be named `testXxx()`. This is because JUnit uses the *reflection* mechanism to find and run these methods. Inside a test method, you can use a variation of the `assert()` method (e.g., `assertTrue()`, `assertFalse()`, `assertEquals()`) to compare the expected and actual results.
- Test Fixture: The set of objects that a test method operates on. You declare these objects as a private variable, and initialize them by overriding the `setUp()` or via the constructor. You can perform clean-up operations by overriding `tearDown()`. Each test method runs on its own `TestCase` instance with its own set of text fixtures. This ensures isolation between the test methods.
- Class `TestSuite`: You can combine many `TestCases` (e.g., written by different person) into a *test suite*, and run them at once.
- Class `TestRunner`: for running the `TestCase` or `TestSuite`.

Writing Tests

Step 1: Extend a subclass of `junit.framework.TestCase`:

```
import junit.framework.*;
public class JUnit38DemoArrayList extends TestCase {
   public JUnit38DemoArrayList(String name) { super(name); }   // constructor
}
```

Step 2: If two or more test methods use a common set of test objects (called *test fixtures*), declare the test fixtures as instance variables. For example, suppose we are testing the class `ArrayList`.

```
private ArrayList<String> lst;     // declare test fixture instance.
```

Step 3: Initialize the text fixture. You can override `setUp()` or use the constructor. Each test method runs on its own `TestCase` instance. This provides isolation between test methods. Each test method invoke the constructor to construct an instance of the `TestCase`, followed by `setUp()`, run the steps coded inside the test method, and the `tearDown()` before exiting. The test methods may run concurrently. For example, let's initialize our test fixture `ArrayList` with two `String` elements.

```
// Initialize the test fixture used by all the test methods
protected void setUp() throws Exception {
   lst = new ArrayList<String>();
   lst.add("alpha");    // at index 0
   lst.add("beta");     // at index 1
}
protected void tearDown() throws Exception { super.tearDown(); }  // for clean-up operation
```

Step 4: Write the test methods for this `TestCase`. All the test methods must be named `testXxx()`, as JUnit uses *reflection* to find these test methods. For example,

```
// test method to test the insert operation
public void testInsert() {
   assertEquals("wrong size", 2, lst.size());  // error message, expected, actual
   lst.add(1, "charlie");
   assertEquals("wrong size", 3, lst.size());
   assertEquals("wrong element", "alpha",   lst.get(0));
   assertEquals("wrong element", "charlie", lst.get(1));
   assertEquals("wrong element", "beta",    lst.get(2));
}
// test method to test the replace operation
public void testReplace() {
   assertEquals("wrong size", 2, lst.size());
   lst.set(1, "charlie");
   assertEquals("wrong size", 2, lst.size());
   assertEquals("wrong element", "alpha",   lst.get(0));
   assertEquals("wrong element", "charlie", lst.get(1));
}
```

Step 5: You can now run the `TestCase`, using JUnit-provided `TestRunner`. There are two versions of `TestRunner`:

text-based `junit.textui.TestRunner`, and GUI-based `junit.swingui.TestRunner`. To use the text-based `TestRunner`, you could include a `main()` method as follows:

```
public static void main(String[] args) {
   junit.textui.TestRunner.run(JUnit38DemoArrayList.class);
}
```

The expected outputs are:

```
..
Time: 0.001

OK (2 tests)
```

You can also invoke the `TestRunner` from command-line:

```
> java junit.textui.TestRunner JUnit38DemoArrayList
```

You can invoke the GUI-based `TestRunner` from command-line:

```
> java junit.swingui.TestRunner JUnit38DemoArrayList
```

Step 6: If there are many `TestCases` (could be written by different people), you can put them together into a `TestSuite` and run all the `TestCases` at once. To do so, in each of the `TestCases`, create a static method `suite()` to extract all the test methods as follows:

```
// In JUnit38DemoArrayList Class - do the same for other TestCase classes
public static Test suite() {
   return new TestSuite(JUnit38DemoArrayList.class);
}
```

Next, write a class to include all the `TestCases` into a `TestSuite`.

```
import java.framework.*;
public class AllTests {
   public static void main(String[] args) {
      junit.textui.TestRunner.run(suite());
   }
   public static Test suite() {
      TestSuite suite = new TestSuite();
      suite.addTest(JUnit38DemoArrayList.suite());
      // other TestCase classes
      //suite.addTest(OtherTestCase1.suite());
      //suite.addTest(OtherTestCase2.suite());
      return suite;
   }
}
```

Automating Unit Testing with Apache's ANT

Apache's ANT is the *de facto* standard for automated building of Java applications (similar to Unix's "`make`" utility). You can download ANT from ant.apache.org (download the ZIP version, and unzip it to a directory of your choice).

We shall use ANT to automate building and testing. First, create a configuration file called "`build.xml`" as follows:

```
<?xml version="1.0"?>
<!-- to save as "build.xml" -->

<project name="Black-Box Unit Test Demo" default="run" basedir=".">
  <!-- build all classes in this directory -->
  <!-- To run this: use "ant build" -->
  <!-- need to include junit.jar in the classpath -->
  <target name="build">
  <javac srcdir="${basedir}"/>
  <echo message="Build done" />
```

```
  </target>

  <!-- Test and build all files  -->
  <!-- To run this: use "ant" (default) or "ant run" -->
  <target name="run" depends="build">
  <java taskname="Test" classname="GradeTestCase" fork="true" failonerror="true" />
  <echo message="Unit Test done" />
  </target>

  <!-- delete all class files -->
  <!-- To run this: use "ant clean" -->
  <target name="clean">
  <delete>
  <fileset dir="${basedir}" includes="*.class" />
  </delete>
  <echo message="clean done" />
  </target>
</project>
```

To build using the above build file, run "`ant`". (By default, it executes "`ant run`", which is depends on "`build`", "`build`" get executed to compile the program, then "`run`" get expected to run the testing. To run only the compilation, use "`ant build`". To run only the cleanup, use "`ant clean`".)

```
prompt> ant
```

```
Buildfile: build.xml

build:
    [javac] Compiling 4 source files
     [echo] Build done

run:
     [Test] ..F
     [Test] Time: 0.005
     [Test] There was 1 failure:
     [Test] 1) testBoundaries(GradeTestCase)junit.framework.AssertionFailedError: expected:<C> but was:<F>
     [Test]     at GradeTestCase.testBoundaries(GradeTestCase.java:23)
     [Test]     at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
     [Test]     at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
     [Test]     at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
     [Test]     at GradeTestCase.main(GradeTestCase.java:34)
     [Test]
     [Test] FAILURES!!!
     [Test] Tests run: 2,  Failures: 1,  Errors: 0
     [Test]
     [echo] Unit Test done
```

[TODO] to be continued...

Unit Testing Best Practices (From JUnit FAQ)

The followings are extracted from JUnit FAQ:

1. When should the tests be written?
   Tests should be written before the code. Good tests tell you how to best design the system for its intended use. They also prevent tendencies to over-build the software. When all the tests pass, you know you're done. Whenever a customer reports a bug, first write the necessary unit test(s) to expose the bug(s) and fix them. This make it almost impossible for the same bug to resurface later.
2. Do I have to write a test for everything?
   No, just test things that could reasonably break. Don't write tests that turn out to be testing the operating system or environment or the compiler. For example,

   ```
   public class AClass {
      int x;
   ```

```
    public AClass(int x) { this.x = x; }
    int getX() { return x; }
    void setX() { this.x = x; }
}
```

A test that testing `getX(setX(y)) == y` is merely testing for `this.y = y`, i,e, testing the compiler! This can't break unless the compiler or the interpreter break!

3. How often should I run my tests?
   Run unit test as often as possible, ideally every time the code is changed. Run all your acceptance, integration, stress, and unit tests at least once per day (for your nightly-built).

REFERENCES & RESOURCES

- JUnit mother site @ www.junit.org.
- JUnit API documentation @ http://junit.sourceforge.net/javadoc_40.
- Kent Beck and Erich Gamma, "JUnit Cookbook" @ http://junit.sourceforge.net/doc/cookbook/cookbook.htm.
- JUnit A Cook's Tour (for JUnit 3.8) @ http://junit.sourceforge.net/doc/cookstour/cookstour.htm.

Latest version tested: JDK 1.6, JUnit 4.8.2
Last modified: October, 2010

Feedback, comments, corrections, and errata can be sent to Chua Hock-Chuan (ehchua@ntu.edu.sg) | HOME