Worksheet 6B: Testing, Part 2 (Java)

Updated: 17th January, 2020

There are two versions of this worksheet. This is the Java version.

When attempting this worksheet, refer to your *completed solutions* to worksheet 6A. We're now going to implement those test designs in code. Make sure you have the lecture notes handy too!

1. Basic Test Code

First, we're going to look at a pure-Java way of writing test code. Obtain copies of the files listed in the table below, and put them into an **appropriate directory**.

These files contain production code implementations of the submodules discussed in worksheet 6A; specifically:

Production code file	Submodule/method	Part of worksheet 6A
`UniAdmin.java`	`calcGrade`	Questions 1a and 2a
`HouseCalc.java`	`roomVolume`	Question 1b
`CharacterUtils.java`	`charCase`	Question 1c
`CharacterUtils.java`	`substr`	Question 1d
`HealthCalc.java`	`uvRating`	Question 2b

These methods all have faults, and we're going to expose those faults through testing.

(a) Let's attack `calcGrade` first. In worksheet 6A, you used both equivalence partitioning *and* boundary value analysis, so you would have come up with two different test designs.

(b) To write the test code, create a new file `UniAdminTest.java` (in the same directory), containing a `main` method, which calls a "`testCalcGradeEqa`" method. Use the examples in the lecture notes to guide you in converting your test design into test code.

When you're done, compile and run the code:

```
[user@pc]$ javac UniAdminTest.java UniAdmin.java
```

```
[user@pc]$ java -ea UniAdminTest.java
```

If your test design is complete, and your test code properly written, this *should* produce an `AssertionError`, indicating that one of the test cases failed. (Don't worry *just yet* about trying to fix the fault. We'll get back to that.)

(c) Add another method "`testCalcGradeBva`" to `UniAdminTest.java` and implement the boundary value analysis test design. Call this method also and run the test implementation. Observe whether any test case fails.

Getting started: Parts of your test code should look like this (depending on what test data you chose):

```
String actual;
actual = UniAdmin.calcGrade(55);
assert "5".equals(actual) : "...";
```

Remember to use the appropriate technique for comparing the *expected* and *actual* results. In this case, `calcGrade` returns a string, so we use `.equals(...)`.

Also note that we have "`UniAdmin.calcGrade`" because that's how we perform a method call across different files (for `static` methods anyway).

Faults and boundary value analysis: `UniAdmin.calcGrade` actually contains *two* defects, one of which might only show up if you use boundary value analysis to design your test cases.

Arrays and `for` loops: Feel free to use or not use arrays and `for` loops in your test code. However, make sure you understand the basic principles of writing test code *first*.

(d) Implement test code for the remaining production code!

You should create files contain the test code for the corresponding production code files:

I. `HouseCalcTest.java`

II. `CharacterUtilsTest.java`

III. `HealthCalcTest.java`.

Inside each there should be a test method to test each production code method.

2. JUnit Test Code

Before doing any conversion, you might like to make a copy of your work from Question 1, so you can refer back to it.

Now, convert your test code so that it uses JUnit, referring to the lecture notes (Testing part 2) for guidance. When it comes time to run your tests, you will need to:

- Obtain a copy of `ise-test.zip`.
- Unzip it into the directory where your JUnit test code lives.

  Be careful of where things end up. The `ise-test.zip` file contains files "`unittest`" and "`unittest.bat`", and a directory called "`buildsystem`". These all need to be placed inside the *very same* directory as your .java files (and not up or down a level, for instance).

- To run your JUnit tests on Linux or MacOS, type:

```
[user@pc]$ ./unittest
```

- To run them on Windows:

```
C:\Users\Myself\ISE\> unittest
```

Note: Be patient. The first time you do this, it may take a couple of minutes, but should be much faster after that.

This approach to running tests implicitly uses a tool called Gradle to smooth over what can otherwise a fairly complicated process, since JUnit is a third-party library.

You *can* also use an IDE (Integrated Development Environment) to do this if you like, and in the real world this is likely to be what you will use. *However,* IDEs can also have a significant learning curve, so unless you've already overcome that learning curve, it may just make things harder for the time being.

3. Debugging

By now you will have had a chance to compare the two approaches. Hopefully you've noticed that the JUnit output is somewhat more informative (even if it appears to be slightly cryptic).

Can you identify some of the faults in the production code? See if you can fix them (or some of them), and re-run your tests to observe the difference.

End of Worksheet