

CS1Bh Lecture Note 9

Development and Testing

This note has three main aims:

1. To explore the process of developing Java code that is fit for its purpose.
2. To explore the use of unit tests as an acceptance test for the individual Java classes being developed.
3. To introduce automated unit testing in the form of the JUnit package¹.
4. To provide a brief introduction to the notion of *refactoring* and its role in the development of Java code.

Most of the material in this note is drawn from current ideas on Object-Oriented development. One particular approach is taken by those advocating *Extreme Programming*². The approach taken by Extreme Programming involves a coordinated collection of development practices that they claim leads to good quality systems. Here we just consider two practices that will help you keep your code development on track. The two are:

Testing: The programmer (and if possible the customer for the program) should write tests for the code before it is developed. The test set should develop with the code. The approach is to do some analysis and design, write some tests to test new parts of the design, write the code then test it. You should not continue until all the tests have been passed. This cycle should be repeated very frequently, 10s of times per day, so the application of the testing process should be automated.

Refactoring: this is a disciplined process of transforming the program so its is better structured and “cleaner” in its design. This includes “small” changes like re-naming of variables to larger changes like identifying new classes or eliminating duplicated code. The aim in refactoring is to improve the code without changing its behaviour³. One simple criterion for this is that if the original program passes a particular collection of tests then the refactored program should pass the same set of tests.

¹ For more detail see: <http://www.junit.org>

²For the gory details see: <ftp://ftp.xprogramming.com/ftp/xpinstall.pdf>

³You can find out a lot more about refactoring in the book: M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Object Technology Series.

Using Tests as a Weak Specification

When we are developing a piece of Java code we have usually been guided to the design for the code from some high level specification of the systems and this has driven the inclusion of particular classes in the design and we will have derived some specification of those classes from the high level design.

For example, in Practical 3 of CS1Ah you were required to implement a calculator. The early part of the practical document contains a discussion of how to go about designing the calculator. In that process the class `PartA` is identified and the need for the various methods and fields in `PartA` is motivated. After reading the early pages of the Practical 3 document we have a fairly clear idea about what each method in `PartA` should do. The question is: how can we use this “clear idea” to help keep the tricky task of implementing the methods in Java on track?

One answer is to try to capture the ideas about the behaviour of methods in a collection of test cases that we can use to check that we are really implementing what we want. For example, we know that the button pushes $0+0=$ should yield the answer 0 and that the input $1+2+3=$ should result in 6 as the answer.

Any Java implementation of the `PartA` class should at least conform to these tests. If we require greater confidence we can always define more tests to check the class. Our aim is to use these tests as a guide to progress as we develop the code. We could just use these tests manually to check the code but fortunately there is a package called JUnit⁴

JUnit

The idea of the JUnit package is to make it easy to construct test cases and even easier to run tests on your code repeatedly as you develop more code. The idea of repeatedly running the tests is that it provides very early detection of potential problems and helps to keep you on track. Suppose you were embarking on `PartA` of Practical 3 the following is a JUnit test suite that check the `add` and `equals` methods are working properly. The code defines a suite of 6 test cases that are automatically run on the code for

The code is best read from bottom to top: the method `suite` returns a test suite consisting of the six individual test cases. The six methods preceding `suite` define the individual test cases. Notice that each of them is defined using the method `AdditionTest` and that this uses the method `complete`. A test case will always make use of one of the `assert` methods provided by JUnit to check that a result is what we expect. Here we use the method `assertEquals` in `complete` to check that the answer we get from our methods in `PartA` correspond to our expectations. The last argument of `assertEquals` is the size of the error we will allow in the equality. This is needed because `double` arithmetic is not always accurate; if the compared values were `ints` we could omit it. The method `setUp` is run before each test case and is used to set up the private variables we use in the test. Here this is just a single instance of `PartA` called `partA`. The `main` method invokes the relevant JUnit methods to run the tests.

⁴The package is available. If you want to use it you need to include the path: `/usr/java/lib/junit/junit.jar` in your `CLASSPATH` environment variable to pick it up when you compile your tests.

```

import PartA;
import junit.framework.*;

public class PartAAddTest extends TestCase {
    private PartA partA;

    public PartAAddTest(String name) {
        super(name);
    }
    protected void setUp() {
        partA = new PartA();
    }
    private void additionTest(int i,int j,int answer){
        partA.add(i);
        int result = partA.equals(j);
        assertEquals(answer,result);
    }
    public void add0to0() {additionTest(0,0,0);}
    public void add0to1() {additionTest(0,1,1);}
    public void add1to0() {additionTest(1,0,1);}
    public void add1to1() {additionTest(1,1,2);}
    public void add9to42() {additionTest(9,42,51);}
    public void compAdd123() {
        partA.add(1);
        partA.add(2);
        int result = partA.equals(3);
        assertEquals(6,result);
    }
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new PartAAddTest("add0to0"));
        suite.addTest(new PartAAddTest("add0to1"));
        suite.addTest(new PartAAddTest("add1to0"));
        suite.addTest(new PartAAddTest("add1to1"));
        suite.addTest(new PartAAddTest("add9to42"));
        suite.addTest(new PartAAddTest("compAdd123"));
        return suite;
    }
    public static void main(String args[]) {
        junit.textui.TestRunner.run(suite());
    }
}

```

Having defined these tests we can start testing our implementation of PartA. If we start of with most of the code for PartA undeveloped with the methods returning 0 at all times then we can see we have work to do. The output we get is:

```
..F.F.F.F.F
Time: 0.464
```

```
FAILURES!!!
Test Results:
Run: 6 Failures: 5 Errors: 0
There were 5 failures:
1) PartAAddTest.add0to1 "expected:<1> but was:<0>"
2) PartAAddTest.add1to0 "expected:<1> but was:<0>"
3) PartAAddTest.add1to1 "expected:<2> but was:<0>"
4) PartAAddTest.add9to42 "expected:<51> but was:<0>"
5) PartAAddTest.compAdd123 "expected:<6> but was:<0>"
```

Only the first test is passed (because the answer is zero). If we then start to program the add method but leave the equals method undeveloped we still see errors:

```
..F..F.F.F
Time: 0.455
```

```
FAILURES!!!
Test Results:
Run: 6 Failures: 4 Errors: 0
There were 4 failures:
1) PartAAddTest.add0to1 "expected:<1> but was:<0>"
2) PartAAddTest.add1to1 "expected:<2> but was:<1>"
3) PartAAddTest.add9to42 "expected:<51> but was:<9>"
4) PartAAddTest.compositeAdd123 "expected:<6> but was:<3>"
```

Finally if we notice that we have to define some of the equals method we succeed and then can go on to do some more design work:

```
.....
Time: 0.363
```

```
OK (6 tests)
```

The most important point to note about JUnit is that it allows the tests to be applied quickly and easily. It is quite possible to apply them after even small pieces of coding and that the test set can be extended easily as development proceeds.

Getting a test suite

It is usually impossible to check all the executions of a unit (very often there are infinitely many). *Coverage criteria* are used to quantify, at least heuristically, how comprehensive a test suite is. The criteria most used in practice are:

- **Statement coverage.** A test suite has perfect statement coverage if each statement of the program is executed in some test case.⁵

⁵For simplicity we assume that for each statement there is at least one input such that the program, given this input, will execute the statement.

- **Edge or branch coverage.** A test suite has perfect edge or branch coverage if each branch of each decision point (if-then-else, while statements, etc.) is traversed by some test case.
- **Condition coverage.** For each 'atomic' condition in the boolean expressions of decision points there are two test cases in which the condition is evaluated to *true* and *false*, respectively.
- **Multiple condition coverage.** Each possible valuation of the atomic conditions in the boolean expressions of decision points appears in some test case.

Let us see this at work on an example. Consider the method

```
void method1(int x,int y,int z,int w) {
    y = y + 1;
    if (x==y && z>w) {
        x = x+1;
    }
}
```

A suite with perfect statement coverage is

$$\{x \mapsto 2, y \mapsto 1, z \mapsto 4, w \mapsto 3\}$$

However, this suite does not traverse the *false* branch of the conditional, and so it does not have perfect edge coverage. The suite

$$\begin{aligned} &\{x \mapsto 2, y \mapsto 1, z \mapsto 4, w \mapsto 3\} \\ &\{x \mapsto 3, y \mapsto 2, z \mapsto 5, w \mapsto 7\} \end{aligned}$$

has perfect edge coverage, but no test case evaluates the atomic condition $x==y$ to *false*, and so it does not have perfect condition coverage. The suite

$$\begin{aligned} &\{x \mapsto 3, y \mapsto 2, z \mapsto 5, w \mapsto 7\} \\ &\{x \mapsto 3, y \mapsto 4, z \mapsto 7, w \mapsto 5\} \end{aligned}$$

has perfect condition coverage, but not perfect edge coverage. It does not have perfect multiple condition coverage either. Finally, the suite

$$\begin{aligned} &\{x \mapsto 2, y \mapsto 1, z \mapsto 4, w \mapsto 3\} \\ &\{x \mapsto 3, y \mapsto 2, z \mapsto 5, w \mapsto 7\} \\ &\{x \mapsto 3, y \mapsto 4, z \mapsto 7, w \mapsto 5\} \\ &\{x \mapsto 3, y \mapsto 4, z \mapsto 5, w \mapsto 6\} \end{aligned}$$

has perfect multiple condition coverage.

In practice it is very hard to obtain a suite with perfect coverage (of any of the kinds listed above). However, it is easy to get an estimate measure the quality of a given suite as the ratio of the visited statements/branches/valuations to their total number.

Refactoring

We will return to the issue of refactoring in later notes but it is worth mentioning here because the capability to apply tests quickly and conveniently is an essential prerequisite to allowing refactoring.

As development proceeds we frequently revise our earlier decisions and make small changes in direction in the development of the code. These small deviations accumulate and fairly quickly we find that we are using classes defined earlier but we are forced to use small work-arounds to accommodate the mismatch between our understanding of the design when we defined the class and our current understanding. Even in quite small projects this can quickly lead to code that is hard to understand and difficult to maintain.

By continually refactoring code we can make a better job of keeping the code clean and well structured. The aim of refactoring is to change the structure of the code without changing its behaviour. If we have an automated test suite we can easily detect if a refactored version of our code fails a test and so fails to meet the criterion for being a refactoring. Without such a test we cannot tell if our refactoring has been successful.

Refactoring is a highly disciplined activity. There are recognised patterns for refactoring code and in general these should be followed strictly. Often it is tempting just to change the code without following the standard patterns. This usually leads to difficulties because the resulting code is not a refactoring to the original code.

Summary

- This note has concentrated on the use of one particular type of testing (unit testing) to provide one particular kind of information about the code. There are many other reasons to test and testing methods.
- Here tests are used to check the individual classes are keeping to their specification. In this case the unit tests are written before the code and used to help direct the development.
- Automated testing is very useful since it allows us to use the tests frequently to help avoid the introduction of serious errors that remain undetected.
- The quality of a test suite can be measured using coverage criteria.
- Automated tests are a useful underpinning to the process of refactoring code. Because they provide an easy way of checking that the refactoring has preserved the behaviour of the code.

*Stuart Anderson.
Javier Esparza, 20th February 2003.*