THE UNIVERSITY OF WESTERN ONTARIO

Computer Science 2212b - Winter 2014
Introduction to Software Engineering

Lab 3 - Unit Testing with JUnit

Part I
Introduction to Unit Testing with JUnit

Estimated Time: 45-60 minutes

1 Overview of Unit Testing

Unit testing is the process of testing the correctness of individual *units* of our programs. A unit is typically considered to be a method, but may also be a class. When we write a set of unit tests for a particular method in one of our classes, for instance, we write short fragments of code that we can then execute to test the method and ensure that it produces correct results. Over time, we build up an arsenal of tests that can be continually rerun as changes are made to our program. This protects us against *regressions* – that is, bugs that have crept up in our methods over time as changes are made to them. Assuming we have written a comprehensive set of tests for our program, when we make a change to the program, we can run our tests and be assured that our program still works as intended and that we have not introduced any bugs into the program.

Unit tests are expected to be as independent as possible – a unit test for a given method should strive to test only that method, and no others, where possible. By striving for independent unit tests, we can be assured that a failing unit test is failing due to the method under test being incorrect, rather than failing because of other methods upon which that method might depend. Of course, our methods often need to call other methods in our classes, so we often *stub out* those other methods and have them return known values to the method under test. This allows us to independently test the method without worrying about whether or not other methods upon which it depends might be causing it to fail. We will discuss *stub methods* in class, but will not cover them in this lab.

In addition to unit tests, we usually also write *integration tests*, which are not intended to be independent. Instead, they test that our units work together properly as a whole. For instance, while a unit test might test a single method in a class independently, an integration test would test to ensure that the method works properly with all the other methods upon which it depends. Both unit and integration testing are vital to delivering quality software. Nevertheless, as we are just dipping our feet into the testing waters, we will restrict our focus to unit testing in this course. Computer Science 4472a/b – Specification, Testing, and Quality Assurance – covers testing in greater depth and is highly recommended for all students.

2 Unit Testing Frameworks

Rather than writing our test code from scratch, we typically rely on *unit testing frameworks* to help us to avoid reinventing the wheel every time we want to write a unit test *suite* (i.e. a set of unit tests). A unit testing framework generally provides a way to group together related unit tests (such as all the tests for a particular class). It also provides us with helper methods that make testing various assertions easy. For instance, a framework might provide a method like `assertArrayEquals` to easily test if two arrays contain the same elements. Finally, it also provides a way of running a suite of unit tests and reports the results of those tests.

JUnit is a popular unit testing framework for Java that provides all of the aforementioned features, making it simple to write unit tests for the methods in our Java classes. Additionally, Maven provides a JUnit plugin that makes it easy to run our unit tests by simply issuing the command `mvn test`. In this lab, we'll gain a basic understanding of JUnit and see how to unit test our code.

3 Getting Started

1. Change to your individual Git repository, create a `lab3` directory, and change to it:

```
$ cd ~/courses/cs2212/labs
$ mkdir lab3
$ cd lab3
```

2. Create a file `pom.xml` with the following elements:

   - `groupId`: `ca.uwo.csd.cs2212.USERNAME`
   - `artifactId`: `USERNAME-lab3`
   - `version`: `1.0-SNAPSHOT`

   Refer to lab 2 for the full details of creating a `pom.xml` (remember that you will also need a `modelVersion` tag). Of course, `USERNAME` should be replaced with your UWO username in **lower case**.

3. Add a dependency to the `pom.xml` file on the following:

   - `groupId`: `junit`
   - `artifactId`: `junit`
   - `version`: Look up the latest version on http://search.maven.org

   Again, refer to lab 2 for the details of adding a dependency to your `pom.xml` file.

4. Create the directory structure to host our project:

```
$ mkdir -p src/{main,test}/{java,resources}
```

   Recall that our source code goes in a subdirectory of the `src/main/java` directory. Similarly, our unit tests will go in a subdirectory of the `src/test/java` directory. Once again, this is Maven's *convention over configuration* approach at work: it assumes you put your code in the right place, and your tests in the right place, and it will then take care of the rest.

5. Create the directory structure to house your Java package:

```
$ mkdir -p src/main/java/ca/uwo/csd/cs2212/USERNAME
```

   Again, `USERNAME` should be your UWO username in **lowercase**.

6. Mirror this directory structure within the `src/test/java` directory:

```
$ mkdir -p src/test/java/ca/uwo/csd/cs2212/USERNAME
```

7. We'll assume we're building a simple `Calculator` class in this lab. Furthermore, let's adopt a test-driven approach where we write the tests first and the code later. Thus, create a class `CalculatorTest.java` in your test directory (i.e. in `src/test/java/ca/uwo/csd/cs2212/USERNAME`) with the contents below. Note that this file goes under the `src/test/java` tree and **not** the `src/main/java` tree.

```
package ca.uwo.csd.cs2212.USERNAME;

import junit.framework.Assert;
import org.junit.Test;

public class CalculatorTest {

  @Test
  public void testPassing() {
    Assert.assertEquals(true, true);
  }

  @Test
  public void testFailing() {
    Assert.assertTrue(false);
  }

}
```

This class is not very useful as it stands, but serves to demonstrate a few JUnit concepts and conventions:

- We typically name our test classes so that the first part is the name of the class being tested (`Calculator`) and the second part is the suffix `Test` to indicate that the class contains a set of unit tests for the `Calculator` class.
- We import several packages provided by JUnit.
- Each method that is *annotated* by the `@Test` annotation will be executed by JUnit as a unit test.
- Although not required, each unit test method begins with the prefix `test` , by convention.
- Finally, JUnit provides us with various *assertions*. In the snippet above, we see two assertion methods being used: `assertEquals` , which tests if two values are equal, and `assertTrue` , which tests if a value is `true` .

8. Run the `mvn test` command:

```
$ mvn test
.
.
.
-------------------------------------------------------
 T E S T S
 -------------------------------------------------------
 Running ca.uwo.csd.cs2212.jshantz4.CalculatorTest
 Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.218 sec <<< FAILURE!

 Results :

 Failed tests:   testFailing(ca.uwo.csd.cs2212.jshantz4.CalculatorTest)

 Tests run: 2, Failures: 1, Errors: 0, Skipped: 0
```

Observe that Maven compiles our test code and executes our tests. It then prints a summary indicating the number of tests that were run, and how many failures occurred. For each failure, it prints the name of the test method that failed – in this case, `testFailing` .

Of course, `testFailing` was intended to fail. We called `assertTrue(false)` in this test method. This assertion will test if the value passed to it is `true` . If so, the test passes. If not, the test fails, and JUnit reports the error.

Now that we understand the basics of JUnit, let's dive in and start testing!

4 Testing the `Calculator` Class

We want to build a `Calculator` class, and we've heard mumblings about this *Test-Driven Development* (TDD) craze that everyone seems to love these days, so let's try it out.

1. Edit the `CalculatorTest` class and remove the `testPassing` and `testFailing` methods.

2. Let's create a test for the `add` method first:

```
@Test
public void testAddTwoPositiveNumbers() {
  Calculator calculator = new Calculator();
  int result = calculator.add(5, 4);
  Assert.assertEquals(9, result);
}
```

   Here, we create a new instance of our (non-existent) `Calculator` class, call its `add` method, passing the values `5` and `4`, and we assert that the result is equal to `9`.

3. Now let's run our test and verify that it fails:

```
$ mvn test
```

   In fact, the test doesn't even compile, since the `Calculator` class does not exist. No matter how silly it may seem, it is important to get in the habit of running the test first to ensure it fails. Sometimes, we'll write a test that is not testing what we *intend* it to test, and it will silently pass because of some other code we've written. Hence, we ensure that the test fails first.

4. We now want to write the *bare minimum* amount of code to get this test passing. Create a file `Calculator.java` in `src/main/java/ca/uwo/csd/cs2212/USERNAME`:

```
package ca.uwo.csd.cs2212.USERNAME;

public class Calculator {

  public int add(int a, int b) {
    return 9;
  }
}
```

   Again, we're focused here on writing the *minimum amount of code* required to get this test passing. We'll refactor later, when needed. This helps us to avoid writing code that we may never need, and avoids introducing unnecessary complexity into our classes. This example may seem silly, but it illustrates the point: just get the test passing.

5. Run the tests again and ensure the test passes (you may want to open two terminal windows so that you can run your tests in one window, and code in another):

```
$ mvn test
```

   The test should now be passing. In Test-Driven Development, we use a cycle called *Red-Green-Refactor*. In other words, we ensure the test fails (i.e. that it *goes red*), we write the minimum amount of code to make it pass (i.e. to make it *go green*), and we later *refactor* to remove any duplication. You might be wondering where the duplication lies in the code we've written so far. Well, we've used the value `9` in our `testAddTwoPositiveNumbers` method, and we've returned the value `9` in our `add` method, so that value is being duplicated. We will remove this duplication soon.

6. Let's write another test:

```
@Test
public void testAddZeroAndOnePositiveNumber() {
  Calculator calculator = new Calculator();
  int result = calculator.add(0, 4);
  Assert.assertEquals(4, result);
}
```

7. Run the tests and ensure the latest test fails.

8. Refactor the code to remove duplication and make all tests pass:

```
public int add(int a, int b) {
  return a + b;
}
```

9. Run the tests and ensure they all pass.

10. Let's add two more tests for the `add` method. Normally, we'd write them one by one and run them, but we'll add both here for the sake of brevity:

```
@Test
public void testAddTwoNegativeNumbers() {
  Calculator calculator = new Calculator();
  int result = calculator.add(-4, -5);
  Assert.assertEquals(-9, result);
}

@Test
public void testAddZeroAndOneNegativeNumber() {
  Calculator calculator = new Calculator();
  int result = calculator.add(0, -4);
  Assert.assertEquals(-4, result);
}
```

11. Run the tests and ensure they all pass. We normally like our tests to fail first, and then write the code to ensure they pass. In this case, however, the code we've already written causes them to pass. This is fine, but in these cases, we need to be vigilant that we're actually testing what we are *intending* to test, and that the tests aren't just passing accidentally due to a bug in our tests.

You may be wondering: *how do we know what to test*? For instance, how many tests should we write for the `add` method before we can be satisfied that it is thoroughly tested? We will be discussing this at length in class, but, for now, you should try to test the various combinations of operands that may be possible: test adding two positive numbers; test adding zero and a positive; test adding zero and a negative; and test adding two negatives.

What about overflow, though? What if we try passing the largest possible integer that can be represented in Java as one argument to `add`, and we pass `1` as the second argument? This is a *boundary* case, and we want to try to test boundary cases as well. Let's add a test.

12. Let's assume that if we have integer overflow, we want the `add` method to throw an `ArithmeticException`:

```
import java.lang.ArithmeticException;
.
.
.
@Test(expected=ArithmeticException.class)
public void testAddOverflow() {
  Calculator calculator = new Calculator();
  calculator.add(Integer.MAX_VALUE, 1);
}
```

    Notice that we don't have any assertions in this method. Instead, we pass the `expected=ArithmeticException.class` argument to the `@Test` annotation to indicate to JUnit that we expect the code in this test method to throw an `ArithmeticException`. If it does, then the test passes. Otherwise, the test fails if the code executes successfully with no exception thrown (or with an exception of a different type).

13. Run the tests and ensure that the latest test fails.

14. Edit the `add` method in `Calculator.java` as follows:

```
public int add(int a, int b) throws ArithmeticException {
  int result = a + b;

  if ((a > 0) && (b > 0) && (result < 0))
    throw new ArithmeticException("Overflow");
  else
    return result;
}
```

    Recall from CS 2208 that we can detect overflow if the two values being added are positive, but the result is negative.

15. Run the tests and ensure they all pass.

16. We now wish to test for underflow:

```
@Test(expected=ArithmeticException.class)
public void testAddUnderflow() {
  Calculator calculator = new Calculator();
  calculator.add(Integer.MIN_VALUE, -1);
}
```

17. Run the tests and ensure that the latest test fails.

18. Modify the `add` method as shown:

```
int result = a + b;

if ((a > 0) && (b > 0) && (result < 0))
  throw new ArithmeticException("Overflow");
else if ((a < 0) && (b < 0) && (result > 0))
  throw new ArithmeticException("Underflow");
else
  return result;
```

    Again, recall from CS 2208 that underflow occurs when the operands are negative, but the result is positive.

19. Run the tests and ensure they all pass.

20. We now have some subtle duplication that we can refactor our. Modify the `add` method as shown:

```
int result = a + b;

if (((a ^ result) & (b ^ result)) < 0)
  throw new ArithmeticException("Overflow / Underflow");
else
  return result;
```

    Note that we are using `&` and **not** `&&`. The `&` operator is the *logical AND* operator, while the `^` operator is the *logical XOR* operator. We won't concern ourselves with why this expression works – if you are curious, try out some examples in binary on paper to see what is going on.

21. Run the tests and ensure they all pass. We have successfully refactored out duplication in our code. Moreover, we have made a change to our code and were able to rerun our tests to ensure that the code still works correctly after making the change. This particular refactoring, however, might be debatable in its utility. After all, we've removed duplication, but we've perhaps done so at the expense of readability.

5 Setup / Teardown

At all times, we should strive to keep our code (tests or otherwise) **DRY**: **D**on't **R**epeat **Y**ourself. That is, we should seek to reduce and eliminate duplication in our code, where possible. This way, when it comes time to make changes to a particular element in our code, we can be assured that we only have to make that change in one place. This is a primary reason that we use constants, for example.

In our `CalculatorTest` class, we have some duplication that might be nice to remove: each of our test methods creates a new instance of a `Calculator` object. Fortunately, JUnit provides a way to execute a method *before* each test that will perform some sort of *setup* that is needed by all of our tests in a given class. Similarly, it provides a mechanism to execute a method *after* each test is run to perform some sort of *teardown*, such as disconnecting from a database or closing a file.

1. Edit the `CalculatorTest` class to add the following:

```
import org.junit.Before;
.
.

public class CalculatorTest {

  private Calculator calculator;

  @Before
  public void createCalculator() {
    this.calculator = new Calculator();
  }

  .
  .
  .

}
```

The `@Before` annotation specifies that the `createCalculator` method is to be run before each of our `@Test` methods are run.

2. From each `@Test` method, remove the line that instantiates a `Calculator` object.
3. Run the tests and ensure they all pass. We've DRY-ed up our test code!

If we wanted to run a method *after* each test method in a class, we would use the `@After` annotation. `@Before` methods are useful for setting up complex objects that we need to use in multiple tests. They not only help us to reduce duplication, but also remove complexity from our test methods and allow them to focus on unit testing, rather than building complex objects.

6 Writing Testable Code

Where possible, we should also strive to write *testable* code. If we can't test our code, then how will we ever know that it works correctly? Test-Driven Development is an excellent way to ensure that the code we are writing is testable, since we write all of our tests first.

Let's look at an example of some untestable code, and see how we might make it testable. Suppose we're building a `SavingsAccount` class. This class will have a method `calculateInterest` that will be called each day. If it's the last day of the month, this method will calculate the interest accrued for the account over the month and add it to the account's balance. Otherwise, if it's any other day in the month, the method will do nothing.

```
public class SavingsAccount {

  private double balance;

  public SavingsAccount() {
    this.balance = 0;
  }

  public void credit(double amount) {
    this.balance += amount;
  }

  public void debit(double amount) {
    this.balance -= amount;
  }

  public void calculateInterest() {
    Calendar today = Calendar.getInstance();

    if (today.get(Calendar.DAY_OF_MONTH) == 1) {
      this.balance *= (1 + (0.025 / 12.0));
    }
  }
}
```

How can we test the `calculateInterest` method? After all, it only performs an operation on the first day of the month. We would have to write a unit test and then run it on the first day of the month to ensure that the interest was calculated correctly. We would then have to write another unit test and run it on every other day of the month to ensure that the method did nothing. Surely, this is not a viable solution.

Instead, we need to make the code testable. We can do this by using the *Inversion of Control* principle: rather than having our class create the objects upon which it depends (in this case, the `Calendar` class), we will *pass in* the objects it needs! In doing so, we are *inverting its control* over its dependencies. Let's see what a revised method might look like:

```
public void calculateInterest(Calendar currentDate) {
  if (currentDate.get(Calendar.DAY_OF_MONTH) == 1) {
    this.balance *= (1 + (0.025 / 12.0));
  }
}
```

Compare this method to the original method above. In this version of the method, rather than instantiating the `Calendar` object ourselves, we take it as a parameter. Consequently, this code is now testable. One unit test can pass in a `Calendar` object representing the first day of the month to ensure that interest is added to the account. Other unit tests might try passing in `Calendar` objects representing other days in the month to ensure that no interest is added to the account.

When you are coding, try to find places where you could apply the Inversion of Control principle to make your code more testable.

7 Other JUnit Tips

A few more tips on writing good JUnit tests:

- Make your test method names descriptive (e.g. `testAddTwoPositiveNumbers` rather than `testAddMethodCase1`). This will help you to identify exactly what is wrong when reading the output produced by `mvn test`.
- Make only one assertion per test method. One should be able to look at the output of `mvn test`, see which tests are failing, and know exactly what is wrong. For instance, if `Assert.assertEquals(-4, result);` is failing, we know that the result returned was not equal to `-4`. However, if we had multiple assertions in the same test method, then we would not know which assertion was causing the test method to fail.
- If you want to temporarily disable a test method, annotate it with the `@Ignore` annotation (e.g. `@Ignore("This test is slow - skipping for now")`). However, don't forget to later remove the annotation! Also, to use this, remember to import the `org.junit.Ignore` class.
- Avoid checking in code to your repository that has failing tests. In other words, *don't break the build*!
- Don't wait until all your code is written to write your tests. Either adopt a test-driven approach in which you write the tests first, or write the tests for a given method immediately after coding it. Otherwise, you'll have an enormous task on your hands.
- See the JUnit documentation for a list of all assertions:
  http://junit.sourceforge.net/javadoc/org/junit/Assert.html
- See the JUnit wiki for more information on it, and for more advanced usage:
  https://github.com/junit-team/junit/wiki

8 Submitting Your Lab

Commit your code and push to GitHub. To submit your lab, create the tag `lab3` and push it to GitHub. For a reminder on this process, see Lab 1.