Data Structures and Algorithms

Lecture 2 — Testing

Alan P. Sexton

University of Birmingham

Spring 2019

Testing

There are many different approaches, but h 3 *LEVELS* of testing are commonly considered:

Unit testing: Ensure a single class works correctly

- Class tested *in isolation*
- Developed by the developer of the class

Integration testing: Ensure parts work together correctly

- May include external services, e.g. database, authentication
- Typically requires initialising resources to known starting states for each test

End-to-End testing: Ensure the whole system works correctly

- These allow only external views of the system to be accessed
- Involve direct use of the user interface to the system
- Automated using tools that support scripting or recording and play back of user interactions.
- Requires no access to or understanding of the code base
- Usually performed by testing personnel independent of the developers

Terminology

SUT: the *System under Test* is the specific item being tested.

- Unit testing: a single class.
- Integration testing: a set of classes or services
- End-to-end testing: the entire system or application being developed.

DOC A *Depended On Component* is a part of the system that we are not testing in this particular test but which the SUT depends on.

- Unit testing: another class that is used by the SUT class.
- Integration testing: may be a whole service used internally by the SUT, e.g. a database server, a transaction manager, an XML I/O system
- End-to-end testing: something used by the application but outside the control of the application developers: e.g. a web service such as Google Maps, a Single Sign On authentication service that handles more applications than just the SUT.

Unit testing in Java with JUnit

JUnit is a Java framework that provides direct support for unit testing

- JUnit was one of the first Java unit testing libraries
- There are many others (TestNG, Spock, Mockito, PowerMock, EasyMock, Arquillian etc.) but JUnit 4 is the leading framework in industry.
- JUnit 5 has been released recently
  - Requires Java 8, has some extra facilities over JUnit 4 but still runs JUnit 4 tests
  - Industry has not switched to JUnit 5 yet
- Most common initial task for CS student placements is to write unit tests using JUnit 4
- We will study JUnit 4

JUnit Libraries

We will be using the `JUnit 4` libraries: `http://junit.org/junit4/`. There are two libraries to be downloaded from this site and installed

- junit (version 4.12 or later)
- hamcrest-core or hamcrest-all (version 1.3 or later)
    - hamcrest-core is sufficient but hamcrest-all contains all the advanced features of hamcrest

Source Directory Structure for Testing

A normal Eclipse project has a `src` directory, under which the packages and sources of the project are stored. We need a slightly different structure to support automated testing:

- Want to separate test code from production code
- But test code typically needs package access to production code classes
- Solution: Split sources into two sub-folders, each with the same package structure
  - Java (and Eclipse) allow you to have multiple different directories that it treats as source directories
  - Equivalent packages in different source directories are merged

Standard test structure:

```
src/main/java
src/test/java
```

Setting up Directory Structure in Eclipse

(I find it easier to start with the the right structure as described in the Eclipse handout rather than using this approach)
To set the structure up in Eclipse, you need to de-register `src` as a source directory, add the new `src/main`, `src/main/java`, `src/test` and `src/test/java` directories and register the two java directories as source directories. In the Package Explorer:

1. Right-click on `src`, "`Build Path|Remove from Build Path`"
2. Right-click on `src`, select "`New...|Folder`", and set "`Folder Name`" to `main`
3. Similarly, create new folders `src/main/java`, `src/test` and `src/test/java`
4. Right-click on each `src/.../java` directory in turn and select "`Use as Source Folder`"
5. If you had any source files or packages in `src`, drag them into `src/main/java` and check that your program still compiles and runs as before

Running Tests in Eclipse

To run `JUnit` tests in Eclipse, add a new run configuration:

- Open the `Run Configuration` dialog (right-click on the project, “`Properties|Run/Debug Settings` or click on the down arrow beside the green run arrow in the button bar and select `Run Configurations`
- Add a new `JUnit` configuration
- Set “`Name`” to the name of your project followed by “Test”
- Select to run all tests in your project
- Set the “`Test runner`” to “`JUnit 4`”
- Run the tests by running this new configuration

Launching the last Run/Debug configuration

By default, Eclipse *"helpfully"* runs the currently selected resource or currently active editor if possible.

- In the case when your focus is on a JUnit test class, it, again *"helpfully"*, creates a new run configuration for the current test class and runs that.
- This means that if your preferred configuration is to run all your tests, and those are spread over multiple test classes, you will only run a subset of your tests

To fix this:

- Select "`Window|Preferences|Run/Debug|Launching`" and change the "`Launch Operation`" to "`Always launch the previously launched application`". Now the run configuration will only change if you explicitly choose to do so.
- While there, you might like to select "`Terminate and Relaunch while launching`", to avoid getting multiple instances of applications running when you rerun an application

A Sample Test

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

public class CalculatorTest
{
        @Test
        public void testAdd()
        {
                Calculator calculator = new Calculator();
                double result = calculator.add(10, 50);
                assertEquals(60, result, 0e-8);
        }
}
```

- Note the imports
- Test methods must have the `@Test` annotation
- Test methods must be of type `public void`
- There can be multiple `assert...` calls
  - Optional initial `String` message parameter
  - First required parameter: expected value
  - Second required parameter: actual value

Assertions

- `assertEquals()/assertNotEquals()`: Uses the `equals()` method to verify that objects are/are not equivalent.
  - The `double` and `float` versions takes an extra `epsilon` parameter to allow approximate matching
- `assertTrue()/assertFalse()`: Checks the condition is true/False
- `assertNull()/assertNotNull()`: Checks the object is/is not null.
- `assertSame()/assertNotSame()` Uses == to verify that objects are/are not the same.
- `assertArrayEquals()` Checks if two arrays contain the same objects in the same order.
- `assertThat()` For checking that an object matches conditions specified by a more complex combination of Hamcrest matchers (auto-generates good failure messages): `https://en.wikipedia.org/wiki/Hamcrest` `http://www.vogella.com/tutorials/Hamcrest/article.html`

JUnit Annotations

@Test The following method is a test method

@Test(timeout=100) The following method will fail if it takes longer than 100 milliseconds

@Test(expected=ArithmeticException.class) The following test method will fail if it **DOES NOT** throw the given exception

@BeforeClass The following *static* method will be invoked only once, before starting all the tests

@AfterClass The following *static* method will be invoked only once, after finishing all the tests

@Before The following method will be invoked before each test

@After The following method will be invoked after each test

@Ignore The following test method, or entire test class, will be ignored during testing. Advantage over simply commenting out: JUnit will warn you about the number of tests ignored.

@RunWith(value = Parameterized.class) Run the following class using the special `Parameterized` runner to execute the tests multiple times with a range of different values

Fixtures

A *fixture* is a common base state that all associated tests should start in.

- Unit tests: It may mean creating an initial set of SUT objects to be used in the tests.
- Integration tests: it may mean clearing out a database and creating an initial set of records in the database.
- End-to-End tests: it could involve populating a database, creating a set of users with different authorisation levels and creating a number of data files for use by the system.

Fixture Setup/Tear down

For unit tests this setup code can be put into a *before method*:

- Annotation `@BeforeMethod`
- Void return type
- No parameters.
- If necessary, tear down with an *after method* (similar to a *before method* but with annotation `@AfterMethod`)

Some setups only need to be done once for all the tests in the class, the suite, the group etc. with correspondingly named annotations (`@BeforeClass`, `@AfterClass`, `@BeforeSuite`, etc.)

Testing Exceptions

Sometimes it is necessary to check that the right exception is thrown for specific error cases.

- One can put the usual "`try ...catch ...finally`" logic into a test
- However, the following, though less flexible, is often sufficient and a great deal simpler:

```
@Test(expected=ArithmeticException.class)
public void testDivideByZeroException()
{
        Calculator calculator = new Calculator();
        int result = calculator.intDivide(2, 0);
}
```

What values should you test?

- Expected normal values: These are a sample of the most usual normal cases, the cases that you expect to be executed most often that should not trigger any special case handling code
- Boundary values: these are cases around where the code must act differently: typically around maximum and minimum values: if your shopping app must show different options for over 18s and under 18s, then test for ages 17, 18 and 19
- Strange values: Test for wierd, unexpected or unreasonable values to ensure that this doesn't break your system. If a person's age is required, what happens if any of the following are entered:
    - a negative number
    - an age well beyond expected human ages
    - a string instead of a number
    - a null value

Parameterized Tests: Imports

- Often we want to run many different examples of the same test but with different input values (e.g in order to test a range of normal values, boundary values, stange values etc.)
- It is not clean to have to copy, paste and edit many very similar tests to try different values
- Solution: use *parameterized tests*

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

import java.util.Arrays;
import java.util.Collection;

import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameter;
import org.junit.runners.Parameterized.Parameters;
import org.junit.Test;
```

Parameterized Tests: The Idea

The idea in parameterized tests is that you create a class with:

- Some **public** class fields for the parameters for a single test
- A constructor that initialises the class parameter fields with the parameters from the arguments to the constructor
- a test method that executes a single test using the class fields as parameters
- A **public static** method to return a list of arrays of Objects, where each array of Objects in the list is one set of parameters for a single test
- Annotations that configure the tests and the error messages for when the tests fail
- With all the above, the framework ensures that the list of parameter Object arrays is generated, and for each array of parameters, an object of the class is created, initialized with one set of parameters, and the test is then executed with those parameters

Parameterized Tests: The Test Class

```
@RunWith(value = Parameterized.class)
public class ParamCalculatorTest
{

//      @Parameter(0)
        public double   expected;
//      @Parameter(1)
        public double   valueOne;
//      @Parameter(2)
        public double   valueTwo;

        public ParamCalculatorTest(double expected,
                                   double valueOne,
                                   double valueTwo)
        {
                this.expected = expected;
                this.valueOne = valueOne;
                this.valueTwo = valueTwo;
        }

        // The Test goes here
}
```

Parameterized Tests: The Test

```
@Test
public void sum()
{
  Calculator calc = new Calculator();
  assertEquals(expected,
               calc.add(valueOne, valueTwo),
               1e-8);
}

@Parameters(name="{index}: {0} = add({1}, {2}")
public static Collection<Double[]> getTestParameters()
{
  return Arrays.asList(new Double[][]
      {
        { 2d, 1d, 1d }, // expected, valueOne, valueTwo
        { 3d, 2d, 1d }, // expected, valueOne, valueTwo
        { 4d, 3d, 1d }, // expected, valueOne, valueTwo
      });
}
```

Test Driven Development

Historically: programmers develop their code and then test it. In 1999 the rise of *Extreme Programming* led to reversing this order into what is called *Test Driven Development (TDD)*:
Basic idea is to repeat the following short cycle:

- write an automated test, which initially can only fail as there is no production code that can satisfy it
- write the smallest, simplest production code to make the test pass
- *refactor* the code to improve it to an satisfactory standard of quality without changing the functionality of the code.

TDD 1: Initial Tests (RED)

- Write a *small* set of tests and ensure that they fail.
- It is important that the tests are run and shown to fail at this stage because this guarantees that the tests actually work.
- If the tests pass when they are initially written, then one could not be sure whether they pass because the production code is correct or simply because there is an error in the test code.

TDD 2: Initial Production Code (GREEN)

- write the production code that makes the test pass
- This code should be kept as small and simple as possible
- May be of low quality or inefficient
- Tests now passes, hence it does what it must do successfully

TDD 2: Refactoring (REFACTOR)

- clean up and improve the code quality **without** changing the functionality of the code
- This may involve:
  - renaming variables, methods and classes
  - modifying the code to move methods from one class into another where they are more appropriate
  - breaking a long and messy method into a number of calls to new smaller simpler methods
  - identifying where duplication of code is occurring and rewriting to avoid and remove duplicates
  - etc.
- Refactoring should never start if any tests are failing and after refactoring there still should be no tests failing.

Summary: RED, GREEN, REFACTOR

Programming errors (aka Bugs) and TDD

The TDD process extends to when a bug is found in a program:

- This could be a test that used to pass and now fails . . .
- . . . or could be that some program behaviour is noted that should not occur
    - e.g. the program crashes or generates an incorrect output.
- first step is to add a new test to demonstrate the bug
    - As with all new tests, this test initially fails
    - By failing it demonstrates the presence of the bug in question.
- Then continue as before
    - add/fix the production code until the test passes, then refactor.
- The new test remains in the set of tests for the system:
    - if this bug reappears later due to some future development changes, it will be immediately identified and can be dealt with.

Test Code Generation from Production Code

There are a number of tools available that analyse production code and generate tests that match that code. In general this is contrary to the TDD philosophy:

- You have to have production code to generate the tests from, contrary to TDD practice
- Automatically generated tests can only reflect the nature of the existing code, not that of the intention of the code writer

While there is a place for such test generation, it is often more in the area of dealing with *legacy code*;

- old code which may need to be maintained but which does not follow current programming practice and/or
- for which little or no current expertise is available.

Production Code Generation from Test Code

When, following TDD, one adds a test in a modern IDE for which the appropriate production code classes or methods are not available, the IDE is often helpful enough to be able to generate skeleton code for the missing classes and methods with very little human intervention. This can significantly speed up overall coding time.

Reasoning behind TDD

Traditional non-TDD approach: *code-first test-later (CFTL)*.

- Common source of bug in CFTL is the programmer building his or her deep knowledge of the code into the tests.
    - If tests are written after the code, then code quirks and the possible faulty assumptions are already fixed in the mind of the programmer and leads to coding those same faulty assumptions into the tests themselves.
    - If the tests are written before the code, then the programmer is more likely to avoid such faulty assumptions as the programmer has not yet faced the coding problems that triggered the flawed reasoning. Instead he or she writes the tests with few assumptions about how the code will actually work.

Reasoning behind TDD, continued

- In TDD, the tests are written to exercise the API
    - Thus the programmer *experiences* the interface that he or she is intending to develop.
    - At this point any inconsistencies or inelegancies in the interface become apparent and leads to better designs of the interfaces.
    - In particular, this helps to avoid the problem of *speculative bloat* of interfaces, where a programmer, not being sure whether some features of the interface might be useful in the future, tends to add in everything he or she can think of in case it might be useful.
    - Finally, having had to think through the interface issues in order to write the tests, the programmer is in a much better position to understand the issues raised by the interface, and less likely to make incorrect assumptions when implementing the corresponding production code.

TDD further work

- Read the Wikipedia entry on Test Driven Development: `http://en.wikipedia.org/wiki/Test_driven_development`
- Take one of the early exercises in your Autumn Semester Java Workshop module and re-implement it from scratch using a test driven development approach. Compare your two different development experiences.