Test Driven Development

1 More on testing

1.1 Additional annotations

In the former section we have seen how basic testing is performed. A testing method is marked with the annotation `@Test`, and this indicates to JUnit that the method will be run as a test. The testing method contains one (sometimes more) *assertions*, which is nothing more than a call to one of the methods of class `org.junit.Assert`, like `assertTrue(...)` and `assertEquals(...,...)`.

In this section we are going to learn how to create slightly more complicated tests.

Constraints: timeout and expected

Sometimes a bug does not result in an incorrect return value, sometimes it results in an infinite loop or an extremely long response time. If we want to specify a maximum time for our testing methods to run, we can do it by passing a "timeout" parameter to annotation `@Test`, as in the following example:

```
@Test(timeout = 1000)
public void testsThatFinishedBeforeOneSecond() {
    // ...
}
```

If the method does not return a value before 1000 milliseconds have passed, the test will fail.

We can also expect a method to return an exception, as in the following example:

```
@Test(expected = IndexOutOfBoundsException.class)
public void testsNegativeIndecesFail() {
    // ...
}
```

If the method does not throw an `IndexOutOfBoundsException` in this test, the test will fail. Note that this is quite the contrary of the usual behaviour: when a testing method throws an exception, it is usually a symptom of something not working as expected; but this not always the case. Situations in which we may want to expect an exception include requests to lists beyond their size, parsing strings that are not in the right format, and passing negative parameters to methods that only accept positive integers.

Initialisation: Before and After

Testing methods usually require the creation of some object. It is quite common that this is needed for all testing method in a testing class.

Instead of repeating the same code in each and every method, which is boring and error-prone, we can create a method to do that before every testing method is called. This method must be annotated with `@Before`. In the same way, if some cleanup must be performed after each testing method ends and before the next testing method starts, this should be marked with the `@After` annotation. See the example:

```
@Before
public void buildUp() {
    // A file is created here to be used in every test.
}
@After
public void cleanUp() {
    // The file is deleted here, after each test ends
}
```

Assuming that your testing class contains three testing methods, the execution path would be: buildUp, first test, cleanUp, buildUp, second test, cleanUp, buildUp, third test, cleanUp. Note that you can change the names of the methods (buildUp and cleanUp are common choices).

Heavy initialisation: BeforeClass and AfterClass

Using `@Before` and `After` is appropriate in those cases in which initialisation and clean-up are fast. However, if the resources needed are costly to allocate and release, and if they are not changed inside each testing method, then it is better to just do some initialisation at the beginning of all tests and some clean-up at the end of all test.

This is done by using the annotations `@BeforeClass` and `@AfterClass`. Examples of resources that are typically acquired and released once per class include network connections and database connections.

1.2 Mock Objects

There are times in which you find yourself in a situation in which you need something you do not have in order to test what you want to test. For example, you may want to test a method like the following one:

```
// part of HospitalImpl.java
public void addPatient(Patient patient) {
    ...
}
```

when you have not have access to any implementation of `Patient`. The most you might know at this point is the interface `Patient`, i.e. its behaviour, what it does. This may happen if you have not implemented `Patient` yet, or if it will be implemented by another person or team, or if it

is expected to be supplied by third-party software that is not available yet. What to do in such a situation?

The answer is: you *mock up* the object(s) that you need. This mean that you create an object that imitates the behaviour of the object you want to use. As the purpose is only to facilitate the testing of another class, mock-up objects tend to be trivially simple, for example:

```
public class PatientMock implements Patient {
    public String getName() {
        return "Mock"
    };
    public int getRelativeCount() {
        return 5;
    };
    // ...implementations of other methods here...
}
```

In a real implementation of `Patient`, a method such as `getRelativeCount()` is probably complex and requires calling other methods in other objects recursively, but none of that is really relevant for a `Hospital`. The only thing that the `Hospital` usually really cares about is that the method returns some `int` and that is what the mock-up returns (if the definition of `getRelativeCount()` involves additional restrictions, e.g. returning only positive integers, the implementation of `PatientMock` must abide by those restriction in the same way as any other implementation).

Mock objects are also very useful when you want to test something that depends on an object that is costly to create, e.g. because it requires taking data out of a database or from a network connection. In this cases, mocking up ensures that your test runs quickly and just as effectively. Remember that any modern piece of software has thousands of tests to run, so you want to make sure that those tests run very fast; otherwise, the programmers in the team will be tempted not to run them everyday —and that is when problems start to creep in!

But mock objects have a second benefit apart from making testing cheaper and even possible: they also make testing *safer* because they decouple the tests of different classes. Sometimes, errors creep in not in the code of one class or another but in their interaction. This particularly nasty kind of error can sneak in undetected if you test your classes with the actual classes they have to interact (i.e. if you use your actual patient implementation to test your implementation of hospitals). By using mock objects, you make sure that your classes must work independently from one another (depending only on trivial mock-ups) which is a good things. Once this "level" of functionality is secured, you can also write tests (called "integration tests") that use several of your classes together to make sure bigger and bigger parts of your application work as expected.

1.3 Writing code that is easy to test

By now you should be convinced that it is a good thing to test our code automatically. The benefits are multiple: it is not as boring, you find bugs once and only once, your code does not decay over time, you are free from the fear of breaking A when you are fixing B or trying to add a new feature that the client requested just before the deadline, etc. The magnitude of these benefits grows with the size of your code and the size of the programming team.

There are situations, however, in which it is difficult to test code. This usually involves input–output, like reading data from the user, getting data from a network connection, getting data from a Graphical User Interface (GUI), etc. Look for example at the following code:

```
public boolean sendEmail(String to, String from, String msgBody) {
    Connection connection = new SmtpConnection("smtp.bbk.ac.uk");
    connection.send("MAIL TO: " + to);
    connection.send("RCPT FROM: " + from);
    connection.send("DATA");
    connection.send(msgBody);
    connection.close();
    return connection.allOK();
}
```

This is a very simple method that sends an email using the SMTP protocol. It opens a connection to the server, sends the email addresses of the recipient and the destination over it, then send the message of the email, and then closes the connection. If everything is OK, if returns true; if something went wrong (e.g. if the wi-fi connection was broken midway through) it will return false.

The problem with this simple method is that it is very difficult and expensive to test: you need to have access to the internet and you need the machine `smtp.bbk.ac.uk` to be up and running. Those are two factors completely out of your control, so your test suite should not depend on them.

This is another benefit of using mock objects. If we mock the connection up, it is very easy to test this code:

```
/**
 * This connection does nothing but returning an OK.
 */
public class ConnectionMock extends Connection {
    public void send(String s) {}
    public void close()        {}
    public boolean allOK()     { return true; }
}
```

We just need to change our original method to accept a connection as a parameter:

```
public boolean sendEmail(String to, ..., Connection connection) {
    connection.send("MAIL TO: " + to);
    connection.send("RCPT FROM: " + from);
    connection.send("DATA");
    connection.send(msgBody);
    connection.close();
    return connection.allOK();
}
```

Note that the only difference is that the `Connection` is no longer a local variable but is instead *injected* into the method as a parameter. *Dependency injection* is an important aspect of good object–oriented programming and we will learn more about it in the future. For now, the only thing that you need to keep in mind is that we have made our method very easy to test: in testing

code, we will pass a `ConnectionMockup` as a parameter while we will use a real `SmtpConnection` in production code. This general principle of reducing internal dependencies and injecting them from the outside can be applied to any code resulting in easier to test code —more robust code in the long run.

You may be thinking now about how you can make sure that you write your code in a way it will be easy to test automatically, without the need of refactoring it later. The easiest way of doing this is by *writing the test first*. This is quite unintuitive ("how can I test something that does not exist?") but it works surprisingly well in practice. We discuss this in the next section.

2 Test-Driven Development

Test-Driven Development (TDD) is a programming methodology that advocates that tests should be driven before the actual program. This is radically different from writing the program first and then writing the tests as we have done in the past.

The TDD methodology is usually described as consisting of three steps that are repeated in a loop:

1. Write the tests for the next functionality/feature of the program.
2. Write the minimal code that passes all the new tests.
3. Refactor the code to make it clearer and simpler. Run the tests at the end to make sure the final functionality is right.

For the sake of clarity, we are going to describe this cycle with a few additional details (optional steps do not take place on every iteration of the loop, only some times):

1. *Optional*. Create or update the interface for the class that is going to be updated (be it fixing an error or adding a feature).
2. Create the new test.
3. *Optional*. Create any needed structure to make the code compile: e.g. create the class and/or the method to test if it did not exist. Do not write any real code yet, just the bare minimum to make the code compile: return `null` for any complex type, 0 for ints, etc.
4. Run the test and verify that it fails. If it does not fail, that usually means (a) it is not testing anything that was not tested before (and is thus redundant), or (b) it is incorrect (the test, not the —unimplemented— feature) and should be fixed, or (c) you wrote too much code in the previous step. In some rare cases, none of this will be true and the test will still pass, e.g. when testing that the length of an empty list is 0.
5. Write the *minimum* code necessary to make the test pass. Do not write anything beyond that. If you think some important functionality is missing, write a test for it in the next iteration of the TDD loop and complete the implementation of the method.
6. Refactor the code if needed for the sake of simplicity or clarity.

7. Start the cycle again with a new feature.

Note that a "new feature" does not need to be a "new method". It is common to write several tests for each method: one for the usual behaviour, a few for corner cases, a few for exceptional situations, etc. Each of these "features" requires a full cycle.

There are four main benefits for this style of programming:

- As the tests are written in advance, the *production* code is written in a way that is easy to test.
- As the tests are written in advance, all the code is tested by at least one testing method. Otherwise, programmers can forget to test some methods or be lazy about it (and often will).
- Errors are detected early, when they are cheap to fix. It is more difficult for errors to remain undetected until later in the development, when it can be more costly to fix them.
- Writing the tests first makes the programmers think about the real specification of the class or method, focusing on *what* needs to be done before their short-term memory is filled with *how* to do it.

This last point is maybe the most important although it is often overlooked. Programming is an inherently complex endeavour, that requires keeping a lot of things in short–term memory, and humans have proved time and time again that they are prone to errors in such a situation. Our evolutionary past never prepared us for challenges in which we need to remember the details of two dozen classes and the parameters of each of their methods, the side–effects they may cause, and how they connect to another hundred classes in the periphery of the problem at hand. Every effort that reduces the number of things that a programmer needs to keep in their heads is a good thing: methods instead of repeated code, local instead of global (or static) variables, structured code instead of arbitrary jumps in the code, encapsulation (private fields), immutability... The evolution of programming languages is a list of features that reduce the amount of data that programmers must keep in their heads at all times, enabling them to produce more complex code with apparently simpler tools, producing code that does not collapse under the weight of its own complexity. TDD is yet another stepping stone in that path: it allows programmers to think first about "what" their code should do and to give an answer to that question before they start asking themselves "how" to do it. Answering "how" always involves answering "what" at the same time (the traditional way of programming that we have been following up to now) and it is more difficult although this is not always evident —especially for novice or very old programmers. This is a common source of errors, especially when maintaining legacy code.

The main shortcoming of TDD is that it is more difficult to appreciate progress at the beginning of the project. Time is spent writing tests so there are fewer "tangible" features can be shown to managers or clients. However, the effort pays in the long run when the code evolves in a controlled way, with a strong battery of tests that ensures that bugs do not re-appear and that the functionality is always moving forwards and never backwards. Remember: if you do not test your own code ruthlessly, your users will. It is better if you find (and fix) most errors before they do.

The TDD methodology can be combined with the "find bugs once" strategy we already know. A program can be developed using TDD, finding most bugs in the process, but some bugs may appear later in the development cycle; if that happens, the "find bugs once" strategy results in adding new tests that will find them as soon as they reappear.

TDD and refactoring

Most programmers do not write new code from scratch. This may happens when learning the basics at college, when you start a pet project[1], or when you create / join a start-up, but most programmers come into projects where some code is already up and running and they need to maintain it. This is known as *legacy code*.

Maintaining legacy code involves mainly two activities: adding new features to the old code base and fixing errors in it. TDD can help in both situations.

One of the main problems of an extended code base, especially if it does not include an extensive battery of tests, is that it is very brittle. Making changes in some section of the code can break something in a different section, one that the programmer is not even aware of. This makes code difficult to change. Some people refer to this as the *viscosity* of code, using the metaphor of a tar pit: the more you try to break free, the more entangled you are until you cannot move any more (and die). This is why nobody likes to work with legacy code.

TDD is a way of breaking free of the tar pit. By writing your tests before you modify anything, you reduce the probability of things breaking without anybody noticing (before the client does). The more thoroughly you test a section of the code, the more confident you can be that your changes or somebody else's will not break that section. Over time, if you can afford the patience and persevere applying TDD to all changes anywhere in your code, your base of legacy code will become more robust, more tolerant to change, and more capable of adapting to the changing needs of your clients. You will have escaped the tar pit.

[1] Which you should do because it is a magnificent way of showing potential employers what you are capable of.