SENG275 – Lab 6
Due Sunday March 5, 11:55 pm

Technical note – you now have a gitlab repository named lab06.

Welcome to Lab 6. Throughout this document, some sections will be in Red, and others will be in **bold**. The sections in Red will be marked. This lab is worth 2% of your final grade.

Quick summary of what you need to do:

1. Compete the Worked example using the code provided, and test the rest of the InvoiceFilter.lowValueInvoices() method.
2. Inspect the TodoApplication class in the main folder, and understand the ways in which the class under test is dependent on other software objects in this file.
3. Test the TodoApplication methods, mocking or stubbing other objects as necessary.
4. Use the verification features of Mockito to confirm that the methods of these mock objects have been called as expected.

You're done!

Overview

When we test a certain class (which we call 'the class under test'), we'd like to test it in isolation. In real software however, no class exists in isolation – each takes part in a complicated network of dependencies and transactions.

This creates problems – if we see certain behaviour from the class under test, it's difficult to be certain that the behaviour results from that class rather than one of its dependencies. The interdependence of these classes also make their boundaries unclear, and make it more difficult to ensure that our unit tests are actually testing the smallest possible units of the program.

Our solution is to mock these external classes. We replace them with minimal objects under our control, which we configure to give them only the minimum behaviour necessary to allow us to test our class of interest.

Mockito is the mocking framework we'll use for this lab. We can create a mock/test double of an object of a given class by using this syntax:

SomeClass variableName = mock(*SomeClass*.class)

Types of Test Doubles

The term 'test double' refers to *any* object that stands in for another, regardless of how sophisticated its behaviour is. Our textbook (Chapter 3.2) uses terms that are now industry standard, and so a little jargon needs to be reviewed.

A *dummy* is a test double that has NO behaviour – all we need from it is its bare existence. Dummies are common when we need to test class A, but objects of class A require an object of class B in their constructor. We create a dummy object of class B, pass it to A's constructor, and never refer to our dummy B again. We can create a dummy with Mockito's mock() method.

A *stub* is a test double that has HARD-CODED behaviour. We use stubs when we want total manual control over the behaviour of our object, and know exactly what methods will be called on it, and with exactly what argument values. A stub doesn't know how to handle unexpected method calls or values. Again we use Mockito's mock() method to create a dummy, then

promote it to a stub by specifying behaviour with when() and thenReturn() methods.

A *fake* is a test double that has a real, working implementation, but the implementation we specify is SIMPLER or FASTER than the real object. For example, we could use a fake implemented as an arraylist rather than a remote database. A fake can handle any method calls or input values that the real object can. Fakes are created the same way stubs are, but the value returned is dynamically determined rather than hard-coded.

A *mock* is a test double that acts like a stub or a fake, but also VERIFIES that it is being called correctly – for example, we might decide that a certain method must be called only once – we want the test to fail if that method is ever called a second time in the same test. We create it the same way as above, but then use the verify() method to enforce our usage expectations.

A *spy* is a test double that intercepts actions intended for the real object, RECORDS them, then passes them on, perhaps to a log file. A spy doesn't replace the dependency – it allows us to gather data about the interactions between objects.

Worked Example

Let's take a look at an example and see how we can use mocking to isolate our class under test. InvoiceFilter.java contains some classes that work together to keep track of invoices and filter them based on their values.

The Invoice class is simple enough that we can verify it just based on inspection – its code is just simple accessors and equality boilerplate. We'd like to test InvoiceFilter, but we can't even instantiate an InvoiceFilter object without passing an object of *another* class (IssuedInvoices) to the constructor. We don't even have the implementation of IssuedInvoices to look at – all we have is its interface. There's no way for us to be certain that tests on InvoiceFilter aren't actually testing the behaviour of IssuedInvoices instead.

The solution is to instantiate the required IssuedInvoices object using a mock object that we control.

Basic Dummies

Basic dummies work well if the bare existence of an object of a particular class is enough, and we don't need any actual functionality.

```
IssuedInvoices issuedInvoicesMock = mock(IssuedInvoices.class);
invoiceFilter = new InvoiceFilter(issuedInvoicesMock);
```

issuedInvoicesMock is now a *dummy* of the IssuedInvoices type; it implements every method of the IssuedInvoices class, but all the dummy's methods return default values – zeros for numeric types, nulls for Objects, and false for booleans.

Verifying Interactions

Our goal is to confirm that our class under test is fulfilling its contract – it is calling exactly the methods that it should. For example, when InvoiceFilter.lowValueInvoices() is called:

```
invoiceFilter.lowValueInvoices();
```

that method should call the all() method of IssuedInvoices. We can *verify* that this has happened by asserting:

```
verify(issuedInvoicesMock).all();
```

If our issuedInvoicesMock ever called its all() method, this will pass. Otherwise the test will fail.

Sometimes we want to verify that a method was called, but that method takes arguments. If we want to check for a specific method, we can provide it in the method call itself, and if we don't care what the arguments are, we can write our verify statement with one of the any() argument substitution functions.

```
verify(issuedInvoicesMock).save(any());
```

Other times we want to be more specific about our verification. For example, we may wish to verify that all() was called exactly once:

```
verify(issuedInvoicesMock, times(1)).all();
```

If we want to verify that NO methods were called on a mock object, we can use:

```
verifyNoInteractions(issuedInvoicesMock);
```

If we want to verify that NO other methods were called on a mocked object OTHER than the ones we have already verified, we can use:

```
verifyNoMoreInteractions(issuedInvoicesMock);
```

An aside: Logging

How do we know what methods we should verify?

It would help us if the documentation clearly stated what external class methods it invokes. For example, lowValueInvoices() could specify that it calls IssuedInvoices.all(), and no other methods in any other classes. Then we would know that we just need to verify that one method and no others.

Unfortunately, the documentation doesn't state this. In lieu of documentation, we sometimes have to resort to code tracing. A (perhaps) easier way to do this is to use Mockito's logging capabilities. We can turn IssuedInvoices from a dummy into (sort of) a spy, and then get it to report back to us the methods that were called on it.

```
IssuedInvoices issuedInvoicesMock = mock(IssuedInvoices.class,
withSettings().verboseLogging());
```

When we then run `lowValueInvoices();`, we get:

```
############ Logging method invocation #1 on mock/spy ########

issuedInvoices.all();

   invoked: -> at
lab06.InvoiceFilter.lowValueInvoices(InvoiceFilter.java:42)

   has returned: "[]" (java.util.LinkedList)
```

```
############ Logging method invocation #2 on mock/spy ########

issuedInvoices.all();

   invoked: -> at
lab06.InvoiceFilterTest.example(InvoiceFilterTest.java:23)

   has returned: "null"
```

So one approach is to mock every external class you might need with logging enabled, and see the full list of interactions from your class under test.

Stubbing

By default, test doubles created using the mock() method are dummies - their methods return values, but those values are `0`, `null` and `false`. We may need to turn this dummy into a Stub, by specifying other return values for its methods.

For example, if we call invoiceFilter.lowValueInvoices(), we're asking for a list of all the invoices with values less than 100. lowValueInvoices() calls the all() method of our dummy IssuedInvoices, which returns an empty list to the method we're testing. Let's change that behaviour by hard-coding all() to return some invoices in a list:

```
when(issuedInvoices.all()).thenReturn(List.of(new Invoice(43), new Invoice(99)));
```

We have now promoted our dummy to a stub – it returns hard-coded values instead of default values. Now when our method under test calls the all() method of our mock object, it will return a list of these two low-value invoices. Since they're both less that 100 in value, they should be returned by lowValueInvoices(), so let's check:

```
assertThat(invoiceFilter.lowValueInvoices()).containsExactly(new Invoice(43), new Invoice(99));
```

Mocking

Finally, we'll promote our stub to a mock, by verifying that the expected methods have been called, and that no others have been called.

```
verify(issuedInvoices, times(1)).all();
verifyNoMoreInteractions(issuedInvoices);
```

AssertJ assertions

Let's return to our assertion statement. Here, we see an assertThat() call, which looks like it returns an object that has an containsExactly() method. Where did all this come from?

AssertJ is an extremely extensive and flexible 3rd party library of assertions, and we'll only scratch the surface of its complexity in this course. We brought it in to the file with:

```
import static org.assertj.core.api.Assertions.*;
```

The basic structure of an AssertJ assertion looks like this:

assertThat(*some_object_or_value*).*has_some_relation_to*(*some_other_object_or_value*)

The number of relations we can use is immense; to get a feel for them, try typing:

```
assertThat("hello").
```

(note the period), and let the auto-complete function show you some of the relations available. Other assertJ assertions that will be useful in this course:

```
assertThat(some_number).isNotEqualTo(some_other_number);
assertThat(some_object_reference).isNotNull();
assertThat(some_object).isSameAs(some_other_object_reference);
assertThat(some_condition).isTrue();
assertThat(some_condition).isFalse();
assertThat(some_list).containsExactly(item1, item2, etc...)
assertThat(some_list).containsExactlyInAnyOrder(item1, item2, etc...)
```

Try modifying an assertJ assertion so that it fails, and note the way the output differs from our usual Junit assertions.

Exercise 1

The code in the worked example above should be enough to write a allLowValueInvoices() test – one that tests the InvoiceFilter when IssuedInvoices returns a list of invoices, all of whose values are below 100.

Complete the tests in InvoiceFilterTest that are empty:

- Create a dummy of any class dependencies (like IssuedInvoices)
- Promote those dummies to stubs by providing them with hard-coded behaviour that allows your tests to explore the scenarios tested.
- Promote the stubs to mocks by verifying that the expected methods (and *only* the expected methods) are called.

Exercise 2

In TodoApplication.java, you'll find simple code for a multi-user TODO list. Test the addTodo, retrieveTodos and completeAllTodos methods using the same steps as above:

- Indentify and create dummies of any class dependencies.
- Promote those dummies to stubs by providing them with hard-coded behaviour that allows your tests to explore the scenarios tested.
- Promote the stubs to mocks by verifying that the expected methods (and *only* the expected methods) are called.