UECS2354 Software Testing
Lab 06: Introduction to Mockito Framework

Introducing the Mockito framework

Mockito, which is evolved from EasyMock, is one of the most popular testing tools available that facilitate the creation of test doubles for testing Java applications (http://mockito.org/). Mockito differs from its predecessor by promoting the use of test-spies over mocks, offering a much cleaner API and very readable error messages. It is also intuitive to use because it follows the natural test code order of arrange, act and assert. In the arrange phase, the test objects which are to be used are set up and initialised with the necessary test doubles. In the act stage, the methods to be tested are invoked on these objects. In the assert stage, the results from running these methods are verified in an appropriate manner. You need to be familiar with the concept of test doubles being used as stubs and mocks.

1. Import the *DemoMockito* project into the Eclipse IDE. Add the Junit, JUnitParams and Mockito (`mockito-all-1.9.5.jar`) libraries to the build path of the project.

2. `RandomAddNumbers.java` and `NewRandomAddNumbers.java` are the applications used in the previous lab, with the concepts of interface and doubles.

3. The test method, `testAddTwoNumbersDummy()` in `NewRandomAddNumbersTest.java` uses the stub approach as in previous lab.

4. The test method `testAddTwoNumbersMockito()` is a reworking of the test method `testAddTwoNumbersDummy()` using the Mockito framework. The main difference is in how the test is being set up.

```
@Test
public void testAddTwoNumbersMockito() {
   RandomNumberFunctionality rnMock =
   mock(RandomNumberFunctionality.class);

   when(rnMock.getRandomInteger(anyInt())).thenReturn(5);

   NewRandomAddNumbers nr1 = new NewRandomAddNumbers(rnMock);

   // Executing the test
   int result = nr1.addTwoNumbers(3);
   assertEquals(8, result);
}
```

We create a test double object `rnMock` that is based on the `RandomNumberFunctionality` interface.
The key statement is `when(rnMock.getRandomInteger(anyInt())).thenReturn(5);` where we setup this test double object to function as a **stub**.

The statement means if the `getRandomInteger()` method call is ever executed on `rnMock`, this method will return the value 5.

The `anyInt()` indicates that it does not matter what is the integer argument passed to `getRandomInteger()`; it will always return 5. This statement essentially performs the role of the `DummyRandomNumber` class that defined in `NewRandomAddNumbersTest`, but in a much more abbreviated form.

We can then proceed to instantiate an object from `NewRandomAddNumbers`, passing it `rnMock` as an argument. The `rnf` instance variable in `NewRandomAddNumbers` will now refer to `rnMock`, so when an invocation is made on `rnf`, the value of 5 will be returned.

The Mockito framework provides a short cut to create dummy classes to be used as stubs or mocks.

5. In the original application, `RandomAddNumbers.java`, the value of the instance variable `rgc` can be set through the constructor if necessary.

```
public RandomAddNumbers(RandomGeneratorClass rgc) {
   this.rgc = rgc;
}
```

The default no-argument constructor will set `rgc` to a new object from the class `RandomGeneratorClass`.

```
public RandomAddNumbers() {
   rgc = new RandomGeneratorClass();
}
```

Notice that this class **DOES NOT involve** the use of **interfaces**, and the `getRandomInteger()` call in **`addTwoNumbers` is performed directly on an object of the actual class** `RandomGeneratorClass`.

6. In the test method `testAddTwoNumbersWithoutInterface()`,we create a test double object `rnMock` based on a concrete class `RandomGeneratorClass`, rather than an interface.

```
public void testAddTwoNumbersWithoutInterface() {
        RandomGeneratorClass rnMock = mock(RandomGeneratorClass.class);
        when(rnMock.getRandomInteger(anyInt())).thenReturn(5);

        RandomAddNumbers nr1 = new RandomAddNumbers(rnMock);

        int result = nr1.addTwoNumbers(3);
        assertEquals(8, result);
    }
```

We are testing the `addTwoNumbers()` from the original application, `RandomAddNumbers`.

Run `NewRandomAddNumbersTest` as a JUnit test to verify that all three test methods within it succeed.

Mockito framework enables straight forward testing of applications without the need to refactor the application to use interfaces and create dummy classes.

7. In the test method `testFindLargestNumberInRandomArrayMockito()` in `NewRandomFindLargestTest.java`, the `rnMock` test double is set up to behave like a stub.

```
RandomNumberFunctionality rnMock = mock(RandomNumberFunctionality.class);
when(rnMock.getRandomInteger(anyInt())).thenReturn(11, 20, 6, 15, 19);
```

When the `getRandomInteger()` method is invoked on it for the first time, it will return the value 11.
The next time `getRandomInteger()` is invoked, the value 20 will be returned.
The third time, the value 6 will be returned and so on.

When `findLargestNumberInRandomArray()` is called with the parameters 5 and 8, an array of size 5 will be created and populated with the numbers 11, 20, 6, 15 and 19. The largest number in this array is therefore 20, and the `assertEquals()` call succeeds.

```
int result = nr2.findLargestNumberInRandomArray(5, 8);
assertEquals(20, result);
```

The `testFindLargestNumberInRandomArrayMockitoWithoutInterface()` test method performs the test using the original, unfactored version `RandomFindLargest`, with `rnMock` based on the concrete class `RandomGeneratorClass`. Run `NewRandomFindLargestTest` as a JUnit test to verify that all three test methods within it succeed.

```
public void testFindLargestNumberInRandomArrayMockitoWithoutInterface() {

        // setup code
        RandomGeneratorClass rnMock = mock(RandomGeneratorClass.class);
        when(rnMock.getRandomInteger(anyInt())).thenReturn(11, 20, 6, 15, 19);

        RandomFindLargest nr2 = new RandomFindLargest(rnMock);

        //create an array of length 5, and fill it with predetermined numbers from 1 to 8
        int result = nr2.findLargestNumberInRandomArray(5, 8);
        assertEquals(20, result);
    }
```

8. The first two test methods in `NewWorkingWithStringsTest.java`, `testCheckStringLength()` and `testCheckStringLengthV2()` use double object to function as a mock in order to verify that certain method calls were performed on it with certain String values being passed as arguments. The `DummyWork` class keeps track of these values through an internal `ArrayList` of String objects, which it then returns as an array of Strings to the test method for checking using the `assertArrayEquals` method.

9. In the third test method, `testCheckStringLengthMockitoV1()`,we setup the test double `sfMock` and pass it to the constructor of `NewWorkingWithStrings`. The `sfMock` is set up as a mock object which essentially verifies that the `doOtherStuff()` method has been invoked on it exactly 2 times. The test will fail if `doOtherStuff()` has been invoked more or less than 2 times.

```
public void testCheckStringLengthMockitoV1() {

    String [] strArray = {"cat", "houses", "dog", "elephant", "rat"};
    StuffFunctionality sfMock = mock(StuffFunctionality.class);

    NewWorkingWithStrings nww2 = new NewWorkingWithStrings(sfMock);
    nww2.checkStringLength(strArray, 4);

    verify(sfMock, times(2)).doOtherStuff(anyString());
}
```

Using the `anyString()` argument means we don't really care what arguments were passed to `doOtherStuff()` when it was invoked twice. This is not really a proper test, because we also need to know the values passed to `doOtherStuff()` to ensure that the `checkStringLength()` method is functioning as per its requirement. The fourth test method, `testCheckStringLengthMockitoV2()`, accomplishes this.

```
public void testCheckStringLengthMockitoV2() {

    String [] strArray = {"cat", "houses", "dog", "elephant", "rat"};
    StuffFunctionality sfMock = mock(StuffFunctionality.class);

    NewWorkingWithStrings nww2 = new NewWorkingWithStrings(sfMock);
    nww2.checkStringLength(strArray, 4);

    InOrder inOrder = inOrder(sfMock);

    inOrder.verify(sfMock).doOtherStuff("houses");

    inOrder.verify(sfMock).doOtherStuff("elephant");
}
```

We set up the mock object for what is known as in-order verification.
This line verify that when `doOtherStuff()` was called the first time, it was passed the value "`houses`", while this line verify that on the second call it was passed the value "`elephant`".

If different values had been passed in either of these calls, the test would fail. `testCheckStringLengthMockitoV2()` is the Mockito equivalent of the initial test method `testCheckStringLength()`.

The last test method, `paramTestCheckStringLengthMockito()`, is the Mockito equivalent of the earlier `testCheckStringLengthV2()` which uses parameters. Notice how the `inorder.verify()` method call is used in a loop to verify that the `doOtherStuff()` method was called a certain number of times with certain values passed for each call.

```
public void paramTestCheckStringLengthMockito(String[] strArray, int strLimit,
                                              String[] expectedResults) {

    StuffFunctionality sfMock = mock(StuffFunctionality.class);
    NewWorkingWithStrings nww2 = new NewWorkingWithStrings(sfMock);
    nww2.checkStringLength(strArray, strLimit);

    InOrder inOrder = inOrder(sfMock);

    for (int i = 0; i < expectedResults.length; i++)
        inOrder.verify(sfMock).doOtherStuff(expectedResults[i]);
}
```

10. `NewWorkingWithStringsTest` works with the refactored version `NewWorkingWithStrings`. We can also write Mockito tests that work with the original, unfactored version `WorkingWithStrings`. Write the test as a short exercise to ensure you understand this concept properly.

    Exercise:
    Refer to `SampleClass.java` and `NewSampleClass.java`. Create Mockito tests that test `sampleMethod()` in `SampleClass` as well as `NewSampleClass` through the setting up of mock or stub objects as appropriate.

Using other Mockito features

We are going to explore some other features of Mockito that further help the facilitation of unit testing. The full list of features and illustrative examples of Mockito can be found at: http://mockito.org/.

1. Refer to `ShowOtherStuff.java`, at the top of this class, there is another class `Student` defined as well as two other interfaces, `MoreStuff` and `ReturnStuff`. Bear in mind that although the demonstration of tests using Mockito will be based on these interfaces to keep matters simple, Mockito test doubles can also be based on concrete classes.

2. Refer to `ShowOtherStuffTest.java`, we set up the test doubles to return references to objects as well as arrays. We set up `msMock` so that when the `getIntArray()` method is invoked on it, the `intArray` array is returned.

```
public void testSomeMethod() {
        MoreStuff msMock = mock(MoreStuff.class);
        Student stud = new Student("Peter");
        int [] intArray = {1, 3, 5, 7};

        when(msMock.getIntArray()).thenReturn(intArray);
        when(msMock.getStudent()).thenReturn(stud).thenReturn(new Student("Paul"));
    }
```

We also set up `msMock` so that when the `getStudent()` method is invoked for the first time on it, it returns the `stud` object; and when the method is invoked for the second time on it, it returns a new `Student` object.

In `someMethod()` in `ShowOtherStuff.java`, a call is made to `firstMethod()` of the `ms` interface reference variable.

```
result = ms.firstMethod() + 10;
```

Assume that it is possible the code in this method can throw a `RuntimeException` (although we do not show the actual concrete class which implements `MoreStuff` that does this). We also know that `someMethod()` must be able to handle this exception appropriately if indeed it is thrown. It does this by nesting the statement within a `try-catch` block.

Thus, if calling `firstMethod()` actually results in an exception being thrown, the `catch` block will be executed and `defaultValue` will be returned from `someMethod()` (currently `defaultValue` is 5); otherwise the result returned from `firstMethod()` is also returned from `someMethod()`.

In order to test `someMethod()` properly, we have to run `someMethod()` so that it produces these two possible outcomes that can result from calling `firstMethod()`.

3. In `ShowOtherStuffTest.java`, `msMock` is set up so that when `firstMethod()` is called on it the first time, it will throw a `RunTimeException`; and when `firstMethod` is called on it the second time, it returns the value 5.
   We then create an object from `ShowOtherStuff` and pass `msMock` to it through the `setMoreStuff()` method, before we invoke `someMethod()` on it twice.

```
when(msMock.firstMethod()).thenThrow(new RuntimeException()).thenReturn(5);

ShowOtherStuff sos = new ShowOtherStuff();
sos.setMoreStuff(msMock);

int result = sos.someMethod();
assertEquals(5, result);

result = sos.someMethod();
assertEquals(15, result);
```

   For the first invocation, we expect an exception to be thrown and hence the value returned should be 5.
   For the second invocation, we expect `firstMethod()` to return the value 5 and therefore `someMethod()` will return the value 15.
   We have therefore accomplished a thorough test of `someMethod()` to show that it works correctly with or without an exception being thrown.

4. The second test method, `testNumTimesCalled()`, presents a variety of ways available in Mockito to verify the number of times a particular method was called.

```
public void testNumTimesCalled() {

    MoreStuff msMock = mock(MoreStuff.class);
    ReturnStuff rsMock = mock(ReturnStuff.class);
    ShowOtherStuff sos = new ShowOtherStuff();
    sos.setMoreStuff(msMock);
    sos.setReturnStuff(rsMock);

    sos.anotherMethod();

    verify(msMock, atLeastOnce()).secondMethod("Great");
    verify(msMock, atLeast(3)).secondMethod("cat");
    verify(msMock, atMost(2)).secondMethod("dog");
    verify(msMock, never()).secondMethod("bird");
    verify(msMock, never()).thirdMethod(anyString());

    verifyZeroInteractions(rsMock);
}
```

The **first** `verify()` method checks that `secondMethod()` has been called with the parameter `Great` at least once. This method will only fail if `secondMethod()` has not been called with this parameter at all.
The **second** `verify()` method checks that `secondMethod()` has been called at least 3 times with the parameter `cat`. This method will only fail if `secondMethod()` has been called less than 3 times with this parameter.
The **third** `verify()` method will fail if `secondMethod()` has been called with the `dog` parameter more than 2 times.
The **fourth** verifies that `secondMethod()` has never been called with the parameter `bird` (it is ok if it was called with other parameters), while the **fifth** verifies that `thirdMethod()` has never been called at all, regardless of parameter value.
The **last** verifies that no methods have been called on the `rsMock` test double.
Notice that the last 3 verifications are a bit unusual, because they test for a negative (i.e. that a method or test double was not used) rather than for a positive (i.e. that a method or test double was used in a certain way).

This is useful in some situations where there are multiple method calls placed in complex nested `if-else-if` or `switch` structures. In such situations, we might want to make sure that certain method calls do not accidentally happen due to logic bugs in the `if-else-if` conditional statements or `switch` structures.

5. Refer to `ShowOtherStuff.java`, the `demoArgumentMatchers()` method in `ShowOtherStuff` basically passes the parameters from its method signature to `methodWithArgs()` when this is called on the `rs` mock object.

   Now look at the last test method, `testDemoArgumentMatchers()` in `ShowOtherStuffTest.java`. We set up `rsMock` to return certain values depending on the parameters passed to the `methodWithArgs()` method.

```
public void testDemoArgumentMatchers() {
    ReturnStuff rsMock = mock(ReturnStuff.class);
    ShowOtherStuff sos = new ShowOtherStuff();
    sos.setReturnStuff(rsMock);

    //#1
    when(rsMock.methodWithArgs(anyInt(), anyString(),
    anyString())).thenReturn("something else");
    //#2
    when(rsMock.methodWithArgs(anyInt(), eq("great"), anyString())).thenReturn("a");

    //#3
    when(rsMock.methodWithArgs(eq(5), startsWith("cat"), anyString())).thenReturn("b");

    //#4
    when(rsMock.methodWithArgs(anyInt(), contains("love"),
    endsWith("dog"))).thenReturn("c");
    sos.demoArgumentMatchers(400, "great", "wonderful");
```

```
    sos.demoArgumentMatchers(5, "catnap", "sleep");
    sos.demoArgumentMatchers(-3, "loveboat", "bigdog");
    sos.demoArgumentMatchers(5, "catlove", "bigdog");
    sos.demoArgumentMatchers(2, "catnap", "anything");
}
```

#1, regardless of what is being passed as the parameters, the result "`something else`" is returned.
#2, if the 2nd parameter has the value `great` regardless of the value of the first and third parameters, the result "`a`" is returned.
#3, if the first parameter is 5, the second String parameter start with the sequence `cat`, then the result "`b`" is returned.
#4, if the second String parameter contains the substring "`love`", and the third parameter ends with the sequence "`dog`", then the result "`c`" is returned.
These statements call `demoArgumentMatchers()` with a combination of different values for its 3 parameters. The output to the console from the `println()` statements in `demoArgumentMatchers()` identify the matching argument patterns based on the result returned.

```
public void demoArgumentMatchers(int x, String s1, String s2) {
    System.out.print ("Calling methodWithArgs with : ");
    System.out.println (x + " " + s1 + " " + s2);
    String result = rs.methodWithArgs(x, s1, s2);
    System.out.println ("result returned : " + result);
}
```

Run `ShowOtherStuffTest` as a JUnit test and verify the output from the Console view.

Notice that the call `sos.demoArgumentMatchers(5, "catlove", "bigdog");` matches the last two argument patterns specified.

```
when(rsMock.methodWithArgs(eq(5), startsWith("cat"),
anyString())).thenReturn("b");

when(rsMock.methodWithArgs(anyInt(), contains("love"),
endsWith("dog"))).thenReturn("c");
```

In such an instance, the **most recently specified** pattern for that mock will be the one that matches and the value **"c" is returned**.

The #1 argument pattern provides a match-all for cases where the 3 parameters do not match with any of the others argument patterns (#2, #3 or #4). If it is removed, the call to `methodWithArgs()` with a set of parameters that are not matched by argument patterns #2, #3 or #4, a `null` is returned instead. Comment out #1 and rerun the JUnit test to verify that this is the case.

6. Mockito is able to set up spy objects. Spies are very similar to the test doubles, with the main difference that **spies require concrete objects to be created** from the classes they are based on.

Spies will allow method calls on themselves to go through to the concrete objects they are based on. Test doubles however do not, as they are not based on concrete objects of a class, rather on the specification of the class instead.

7. Refer to `SampleClassTest.java` and `SampleClass.java`. The two methods `getNumberFromFile()` and `writeDataToFile()` in the classes `FileReaderClass` and `FileWriterClass` in `SampleClass` have an additional `println` statement to provide output to the console as an indication that they have being called.

   In `SampleClassTest`, `readSpy` and `writeSpy` are created as spy objects based on the `FileReaderClass` and `FileWriterClass` objects, `frc` and `fwc`.

   ```
   FileReaderClass readSpy = spy(frc);
   FileWriterClass writeSpy = spy(fwc);
   ```

   The `readSpy` is set up to behave as a stub to return 100.

   ```
   when(readSpy.getNumberFromFile()).thenReturn(100);
   ```

   We also verify that `writeDataToFile()` was called on `writeSpy` exactly 5 times with the value 100 passed each time.

   ```
   verify(writeSpy, times(5)).writeDataToFile(100);
   ```

   Run `SampleClassTest` as a JUnit test, notice that the output in the console view indicates that the `getNumberFromFile()` and `writeDataToFile()` methods have being called from the `frc` and `fwx` objects.

   Every time a call is made to a method on a spy object, the method of the concrete object that the spy is based on is also called as well. In this example, the `FileReaderClass` and `FileWriterClass`. That's why we see output in the console view.

   However, for a normal test double that is based on a concrete class, this does not occur. A normal test double created using the mock method will not relay method calls on itself to the actual class object, regardless of whether it is set up as a stub or mock. This is because the creation of a normal test double does not require the creation of objects from the concrete classes that they will be based on, unlike the case of spies.

   To verify this, comment out the following statements:

   ```
   FileReaderClass readSpy = spy(frc);
   FileWriterClass writeSpy = spy(fwc);
   ```

   Uncomment the following statements that create `readSpy` and `writeSpy` as normal test doubles.

   ```
   FileReaderClass readSpy = mock(FileReaderClass.class);
   FileWriterClass writeSpy = mock(FileWriterClass.class);
   ```

Save the changes and run it again. Notice that now there is no longer any output to the console, indicating that neither the `getNumberFromFile()` and `writeDataToFile()` methods are called from the `frc` and `fwc` objects.

8. When spy objects are used, it usually implies that there is some form of interaction between the SUT and the DOC since the method calls on the spy objects are also relayed on to methods of different classes as well.

   This is a less than ideal situation, since unit tests should be isolated in order to facilitate the process of locating a bug in the event of a test failure. However, spy objects can be useful when working with legacy code where the source code is not available for modification, or when needing to create a partial mock.

   A partial mock is useful in a situation where we want to be able to control the behaviour of some methods in a concrete class to facilitate testing, while allowing the other methods in that class to function in their normal manner.

9. For example, `readSpy` has been set up as a spy object. We stub the method `getNumberFromFile()` from the `frc` object that `readSpy` is based on; so that even though this method is called on the `frc` object, the actual value that is returned is not 10 (which is the value specified in the actual implementation of `getNumberFromFile`), but rather the value 100.

   If it has not been stubbed, then the return value will be 10 as in actual implementation. Comment out the following statement and run as Junit test again.

   ```
   when(readSpy.getNumberFromFile()).thenReturn(100);
   ```

   Notice that the `verify()` failed as the parameter passed to `writeDataToFile()` is not 100 anymore.

   So, we can control which methods get stubbed and which methods are allowed to run in an object of a concrete class. This is possible with a spy object or a partial mock, but not with a real test double or real mock. We can add several other methods to `FileReaderClass` in `SampleClass` and include lines to call these methods via the `frc` reference variable in `sampleMethod()` to verify this.