Assignment 3

Unit Testing using Test Doubles & Integration Testing

Due Date: Jan. 30, 2020

Yasaman Amannejad, 2020

Table of Contents

1 SUMMARY

The objective of this assignment is to introduce students to the concept of mocking in unit testing, and also the difference between unit testing and integration testing. This assignment should be completed in groups.

1.1 SOFTWARE UNDER TEST

In this assignment, you will be continuing to work on JFreeChart software from assignment 2 (JFreeChart v1.0.zip).

1.2 SETUP YOUR TEST ENVIRONMENT

The setup for this assignment is the same as assignment 2. You only need to add one more library to your project. If you do not have your project setup, please refer to assignment 2 and follow the setup process.

New library - In this assignment, you need to add the Mockito jar file (download) to your project to complete your task. Please refer to Section 2 for the setup process.

1.3 ASSIGNMENT PROCESS

Part 1 - In this assignment, you are going to write unit tests for methods in the `DataUtilities` class (`org.jfree.data.DataUtilities`). This class has 5 methods. Some of the methods use the interfaces `Values2D` and `KeyedValues`. You are going to use the Mocking techniques to isolate these methods from external classes and test them individually. For designing your tests, similar to assignment 2, you will rely on specifications in the Javadoc. You will use the Mockito library to create your mock objects. When dividing your tasks, make sure that each member of the team gets at least one method that requires mocking (depends on `Values2D` or `KeyedValues` interfaces).

Part 2 - Next, you will be developing integration tests for the integration of `DataUtilities` and `DefaultKeyedValues2D` (`org.jfree.data. DefaultKeyedValues2D`) classes.

1.4 DELIVERABLES

Submit your test codes (unit test and integration) as a Zip file. No report submission for this assignment is required. Marking scheme is provided for you in Section 5.

2 SET UP YOUR PROJECT

1. Open Eclipse.
2. Download the **mockito-all_2.0.2-beta.jar** file and add it to your project.
   https://mvnrepository.com/artifact/org.mockito/mockito-all/2.0.2-beta
3. Go to your project from assignment 2.
4. Right-click on the project name, and them Properties. Go to the Libraries tab, and using Add Jar or Add External Jar, add your Mockito library to your project. The Java Settings dialog should now look like Figure 1, below.

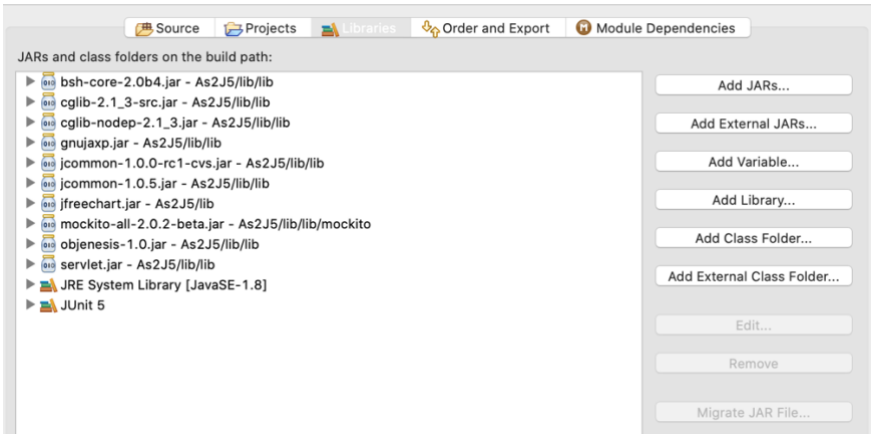


Figure 1 - The Java Settings dialog after adding required archives

3 FAMILIARIZATION

3.1 TEST DOUBLES

When the movie industry wants to film something that is potentially risky or dangerous for the leading actor to carry out, they hire a "stunt double" to take the place of the actor in the scene. The stunt double is a highly trained individual who is capable of meeting the specific requirements of the scene. They may not be able to act, but they know how to fall from great heights, crash a car, or whatever the scene calls for. How closely the stunt double needs to resemble the actor depends on the nature of the scene. Usually, things can be arranged such that someone who vaguely resembles the actor in stature can take their place.

In software, an object under test may have dependencies on other objects. To isolate the behavior of the object in automated testing, it is common to replace real objects with our equivalent of the "stunt double": the Test Double. Test doubles behave like their real equivalent objects, but are actually simplified. This is useful if the real objects are impractical to incorporate into the unit test or are not yet implemented.

Test Double is a generic term used for these simplified objects. Although test doubles come in many flavors (Fakes, Stubs, Mocks, Spy, etc.), people tend to use term Mock to refer to different kinds of test doubles. Here, we are going to first learn about the definition of each type of test doubles:

Fake - Fakes are objects that have working implementations, but are not the same as the real one. Usually they take some shortcut and have simplified version of real object. For example, replacing an object that accesses the database with an object that uses an in-memory implementations, such as a HashMap, could be a shortcut to avoid starting up a database and performing time consuming requests.

Stub - Stub is an object that holds predefined data and uses it to answer calls during tests. It is used when we cannot or don't want to involve objects that would answer with real data or have undesirable side effects. An example can be an object that needs to grab some data from the

database to respond to a method call. Instead of the real object, we introduced a stub that always answers a predefined set of values.

Mock - Mocks are objects that register calls they receive. They may also return predefined outputs like stubs. We use mocks when we don't want to invoke real code or when we only want to verify that a specific method is called. An example can be a method that calls e-mail sending service. We may not want to send e-mails each time we run a test. Instead, all we want to verify is that the email service will be called in certain situation.

Spy – We can use a more capable version of a Test Stub, the Test Spy as an observation point for the indirect outputs of the SUT. Like a Test Stub, the Test Spy may need to provide values to the SUT in response to method calls but the Test Spy also captures the indirect outputs of the SUT as it is exercised and saves them for later verification by the test. So, in many ways the Test Spy is "just a" Test Stub with some recording capability.

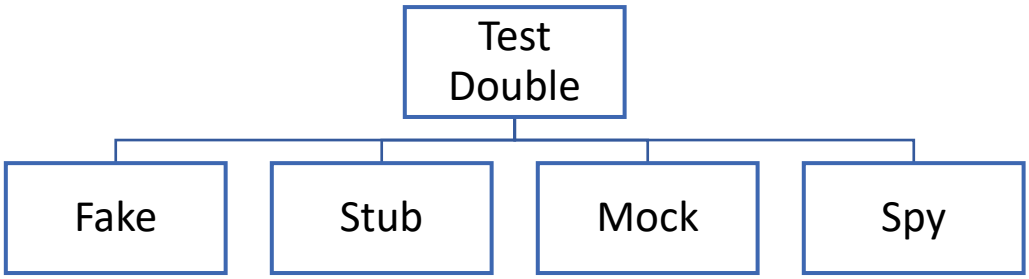


Figure 2 – different types of test double

Figure 2 shows the different types of test doubles. Regardless of which of the variations of Test Double we choose to use, we must keep in mind that we don't need to implement the whole interface of the real object. We only provide whatever functionality is needed for our particular test.

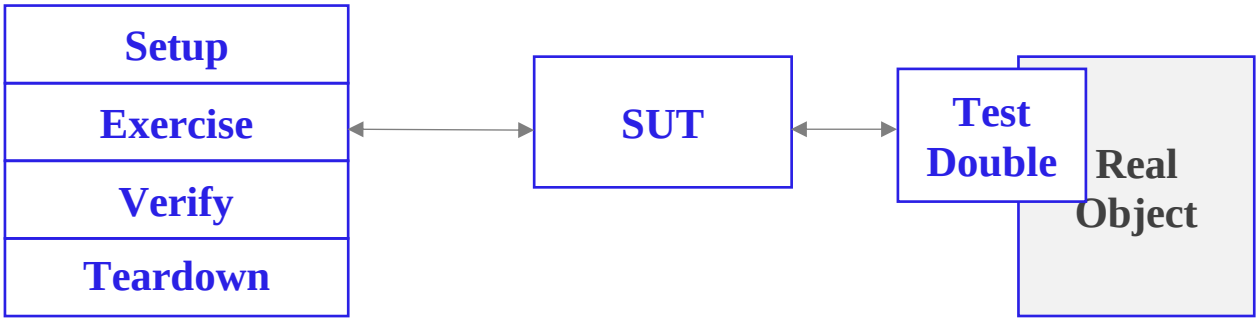


Figure 3 – Interactions between your test code, SUT and test doubles

In this assignment, we will be using Mockito library. A full documentation can be accessed at: https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html

Let's now create our first test mock:

5. Note that some methods in `DataUtilities` use the interfaces `Values2D` and `KeyedValues`. Because the methods take in interfaces as parameters, you will not know how the inherited classes may function. Mocking allows us to return any values or throw any exceptions we want, when we want.

To get you started, include the following example in your `DataUtilitiesTest` test code:

```
import static org.mockito.Mockito.*;

class DataUtilitiesTest {

    private Values2D value;

    @Before
    static void setUp () throws Exception {
        value = mock(Values2D.class);
        when(value.getColumnCount()).thenReturn(4);
        when(value.getRowCount()).thenReturn(3);

        when(value.getValue(0, 2)).thenReturn(5);
        when(value.getValue(1, 2)).thenReturn(7);
        when(value.getValue(2, 2)).thenReturn(1);
    }



}
```

Figure 4 – Creating a mock object

The variable `value` is a mock instance of the Values2D interface. Based on the defined behaviour, it represents a 2D object (3 X 4). The value for three elements are defined for this object. As mentioned before, you only need to define the behaviour as much as you need. The remaining elements are not defined, as they were not needed for the test we are planning to write.

$$\text{value} = \begin{bmatrix} ? & ? & 5 & ? \\ ? & ? & 7 & ? \\ ? & ? & 1 & ? \end{bmatrix}$$

Now, let's write the test method. In the test method, you will treat the mock object, as a real object and write your tests to compare the values. Remember that in addition to the comparison done with Assert methods, you can verify the number of method invocations with verify method.

```
import static org.mockito.Mockito.*;
class DataUtilitiesTest {

	private Values2D value;

	@Before
	static void setUp() throws Exception {
		value = mock(Values2D.class);
		when(value.getColumnCount()).thenReturn(4);
		when(value.getRowCount()).thenReturn(3);

		when(value.getValue(0, 2)).thenReturn(5);
		when(value.getValue(1, 2)).thenReturn(7);
		when(value.getValue(2, 2)).thenReturn(1);
	}

	@Test
	void test() {
		assertEquals(13, DataUtilities.calculateColumnTotal(value, 2), .01d);
		verify(value, times(3)).getValue(anyInt(), anyInt());
	}
}
```

Figure 5 – Using a mock object in a test

For writing your test cases, you should use the Javadoc specifications and design your test cases. In your implementation, when accessing an external object, you need to mock that object.

3.2 INTEGRATION TESTING

Integration testing is a level of software testing where individual units are combined and tested as a group. After each method being tested in isolation, the purpose of the integration testing is to test the interactions between these units.

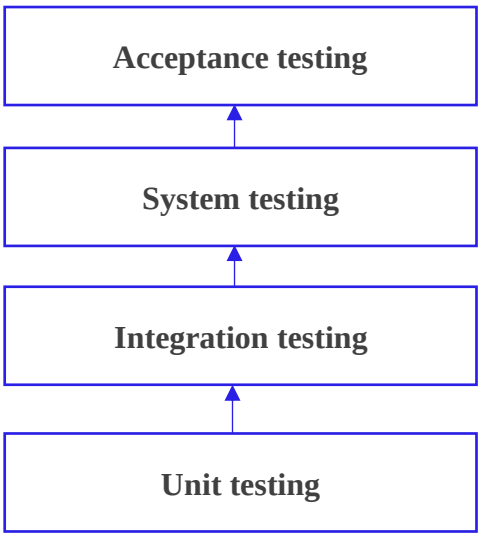


Figure 6 – Levels of testing

In integration testing, any external dependency that is part of your test scope, does not require a test double, and can be called directly.

4 INSTRUCTIONS

4.1 DEVELOPMENT OF UNIT TEST CODE

6. You are required to create unit tests for the following class. This class has 5 methods. Take a minute to browse the Javadoc API specifications of each of these methods in Javadoc.

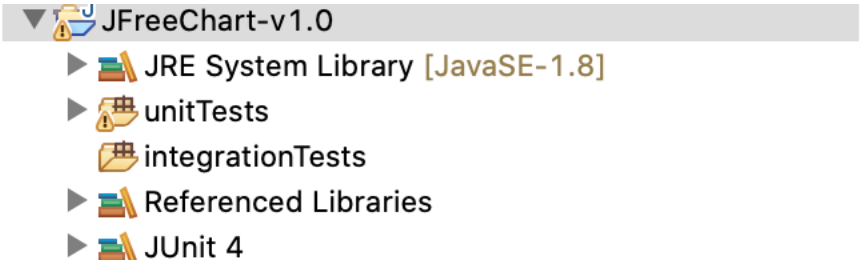
   `org.jfree.data.DataUtilities` (in the package `org.jfree.data`)

   Make sure that each member of the team gets at least one of the methods that require mocking.

7. Write unit tests for all methods in the `DataUtilities` class. Use the mocking library when necessary. You may use any of the following test doubles in your tests: Stub, Mock, or Spy. Using Mockito can create any of these test doubles.

4.2 DEVELOPMENT OF INTEGRATION TEST CODE

To understand the difference between Unit testing and Integration testing, you are going to implement another set of test cases. To differentiate them from the previous set of tests that you have created, you may create a separate folder under your test folder.



8. In your integration tests, write test cases that test the integration of `DataUtilities` class and the `DefaultKeyedValues2D` class.
   `DefaultKeyedValues2D` exists in `org.jfree.data.DefaultKeyedValues2D` package and is one of the classes that implements the `Valued2D` interface (More precisely, `DefaultKeyedValues2D` class implements `KeyedValues2D` interface which extends `Valued2D`).
9. When writing your integration tests, only test the interaction between `DataUtilities` and `DefaultKeyedValues2D` classes. In another word, if `DefaultKeyedValues2D` class has methods that are not called from `DataUtilities` class, testing those methods should not be part of your integration tests.

5 SUBMISSION

5.1 DELIVERABLES

Submit your test codes (unit test and integration) as a Zip file. No report submission for this assignment is required. Submit your zip folder in Blackboard.

5.2 MARKING SCHEME

Marking scheme	
Completeness - is there any missing test cases based on the test case design approach? - are all dependencies tested properly in the integration tests?	40%
Correctness - do the tests actually test what they are intended to test? - are the external dependencies mocked properly in the unit tests? - are the actual objects are called properly in the integration tests?	40%
Adherence to the specification in Javadoc (are tests generated only based on the specifications in the Javadoc files)	10%
Clarity (test code style and quality)	10%