APPENDIX A: MOCKITO UNIT TESTING TUTORIAL

This appendix is a tutorial over how to implement Mockito Unit testing/mocking framework. It also contains a code example of a simple test created exclusively for this tutorial.

The goal with this tutorial is to show how to implement and use the Mockito testing framework. Mockito is a testing framework implemented as an extension to JUnit, a testing framework itself for Java. Mockito allows for mocking of objects. Mocked objects are used in automated unit testing with Mockito. A mock simulates the behavior of an object in order to test another object that is dependent on it. The advantage of this is that the behavior of a mock can be controlled very precisely in a test environment and dependencies between different objects are easily set-up in a separate container.

Before starting the tutorial, it is assumed that Netbeans 7.0.1 or later has been installed and that the user has access to our Java EE Web project, The Recruitment System. Also the project must be imported and implemented in NetBeans. This tutorial has only been tested on a PC running Windows 7. It has not been verified to work on other Operating systems but there should not be any major differences since this tutorial focuses on NetBeans IDE.

1.1 Downloading the necessary files.

Visit the Mockito official website at: https://code.google.com/p/mockito/downloads/list
Download the latest stable build jar file called mockito-all-x.x.x.jar. This file only contains a single jar file for importing the Mockito source files.

1.2 Implementing Mockito to your Java EE Web project

Import the library to the project as described below.
Start NetBeans.
Right-click on the folder *Test Libraries* located inside your Java EE Web project tree. Your Java EE project tree can be found under *Projects* in NetBeans (Figure 1).

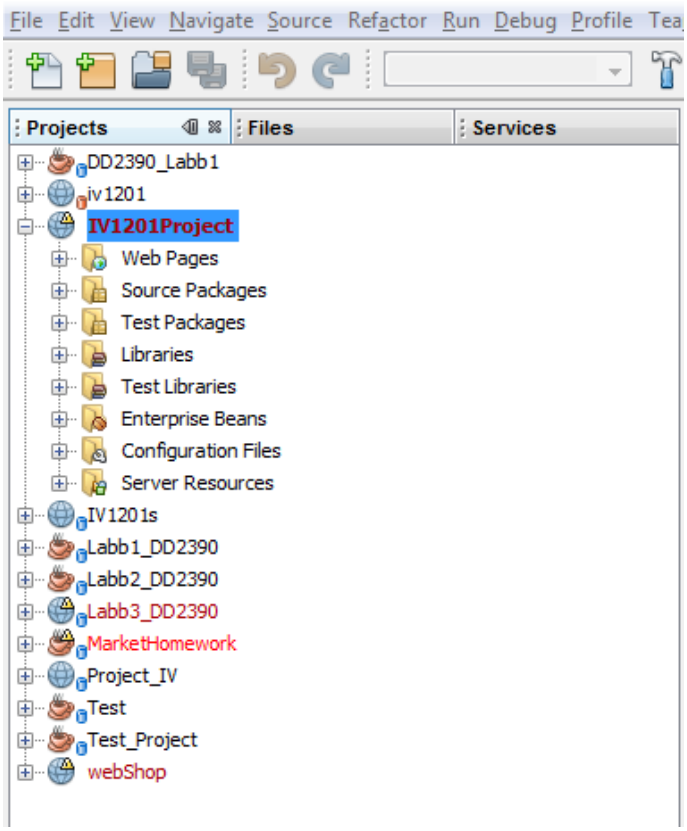

Figure 1 – Project tree in NetBeans IDE.

Left-click *Add Jar/Folder* and then navigate on your PC to the folder which contains the Mockito library that you downloaded earlier from the Mockito website. Choose the Mockito library jar file called *mockito-all-x.x.x.jar*.

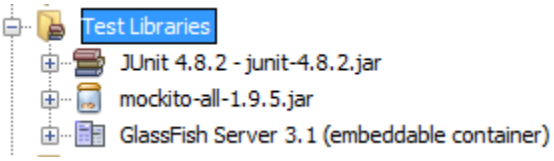

Figure 2 – Mockito framework imported into test libraries.

Once Mockito has been imported it will be visible as a jar file in the folder *Test Libraries* at your Java EE Web project tree (Figure 2). You are now able to use Mockito as a testing tool.

1.3 Setting up test environment for Mockito

Right click on your Java EE Web project and navigate to *New* and then to *Other…*
Choose the category *JUnit* on the left hand side under *Categories* and choose the file type *Test for existing class* (Figure 3).
NOTE: In later NetBeans builds the category name has been changed to *Unit Tests*.

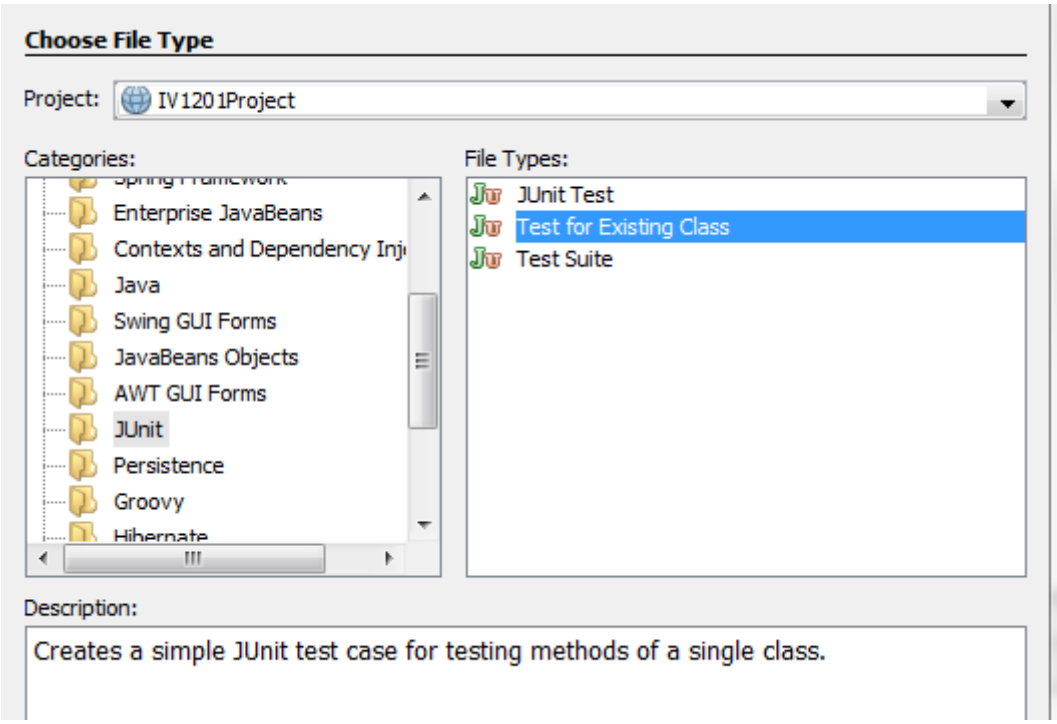

Figure 3 – How to create a new test for existing java class in NetBeans IDE.

Choose the class you want to test from your Java EE Web project either by typing the class name or by browsing to it with the *Browse...* button. For this example, we have chosen the class *DAOFacade.java*. It is recommended that you do the same for easier learning. *DAOFacade* is located in the project source package *controller*.

Click *finish*.

You will now be presented with a new test class called *DAOFacadeTest.java* which contains around 150 lines of automatically generated test code. For this example, please delete this code but leave the imports at the top of the class intact. This new test class, *DAOFacadeTest*, will be located under a new folder called *Test Packages*. The name of test package will be the same as the name for the source package where the tested class resides. For instance, *DAOFacade* is located in the source package *controller* and its test class, *DAOFacadeTest* is found in the test package with the same name, *controller*. Their locations are illustrated in Figure 4 where the project tree is shown. The classes are highlighted in blue.

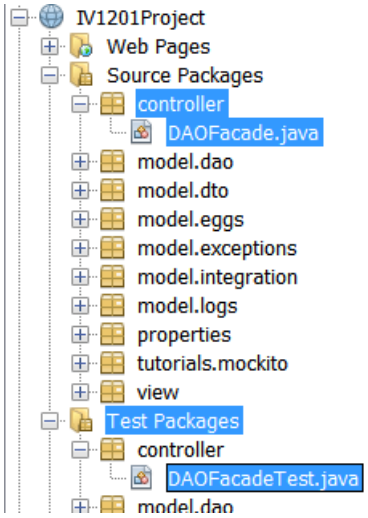

Figure 4 – *DAOFacade.java* and *DAOFacadeTest.java* in their respective locations.

1.4 Writing a simple test with Mockito

At this point, *DAOFacadeTest.java* should only contain the imports. It is now time to write the test in it. The Mockito API provides several methods for many different testing scenarios. For this example, the focus will be on one common method called *verify()*. Before writing the test, it must be defined and, to this end, some context must be given first.

DAOFacade is the controller for the IV1201 web project and it acts as a facade over the lower layers of the system, such as the source package *model*. All calls from the upper layers, for instance the source package *view*, must pass through the controller. Its job is to validate the calls and to delegate them down to the right location in the lower layers.

DAOFacadeTest will test one of those delegations and check that the right method was called in another class, at the lower layers. The method that will be subject to test is called *login()*. This method receives two parameters from the upper layers and passes them to a lower-layer class called *Logic.java* by calling the method *login()* in this class. To clarify this, the method that will be tested in *DAOFacadeTest* is shown in Figure 5.

```java
@Stateless
public class DAOFacade
{

    @EJB
    private Logic logic;

    /*
     * Injection point for Logic. Instead of accessing it as an EJB
     */
    @Inject
    void setLogic(final Logic logic)
    {
        this.logic = logic;
    }

    /**
     * Checks if the username and password are correct.
     *
     * @param username
     * @param password
     * @return 0 if success
     */
    public int login(String username, String password)
    {
        return logic.login(username, password);
    }
```

Figure 5 – Login method in *DAOFacade.java* passes parameters to method of the same name in class *Logic.java.*

The class *Logic* returns to *DAOFacade* an integer 0 if the set parameters are correct. These parameters have to do with the login information and are provided by the user when trying to log in to the system. Now that the test is defined, it is time to give a short description of the methods used in the test at *DAOFacadeTest*.

The method *verify()* is often used to make sure that a certain behavior happened or not. The method also allows for high granularity. For instance, the test can verify that a certain method was called at least three times or that the call is done with a specific set of parameters. In this case, the test will verify that the method in *Logic* was called exactly one time with the correct login parameters.

This implies that there exist dependencies between *DAOFacade* and *Logic*. Since the test is run in a separate container than the project itself, the dependencies must be set up somehow. Thankfully, these can be mocked out using Mockito. This is what mocking is used for and why Mockito was created, to simplify mocking for the user. The tutorial will come back to the properties of Mockito concerning mocks in section 1.6 but for now, it is only shown how it is done. Figure 6 illustrates the mocking of *Logic*, the class for which *DAOFacade* is dependent on.

```
/*
 * Set-up that runs Before the test.
 * Specified with JUnit annotation @before
 */
@Before
public void setUp()
{
    // Mock Logic in order to set up dependencies
    mockedLogic = mock(Logic.class);
    // Pass mocked Logic to DAOFacade
    daof.setLogic(mockedLogic);
}
```

Figure 6 – Set-up stage of test in *DAOFacadeTest.java.* This shows how to mock *Logic.java.*

Please observe the *@Before* annotation. This is a JUnit annotation and it specifies that the method *setUp()* must be executed before the test itself. *@Before* is usually used when there is more than one test method and they share resources such as mocked objects. It is therefore good practice to mock out the required dependencies during the set-up stage of the test instead of directly in the test method, although this is also a viable option. In Figure 7, *DAOFacadeTest* is shown in its entirety. Please use this as a template for your own test class.

```java
package controller;

import model.dao.Logic;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import static org.mockito.Mockito.*;
import org.mockito.runners.MockitoJUnitRunner;
// Needed for Mockito Annotations such as @Mock. Not used in this test class.
@RunWith(MockitoJUnitRunner.class)
public class DAOFacadeTest{
    // Variable instantiations
    DAOFacade daof = new DAOFacade();
    Logic mockedLogic;

    @Before
    public void setUp(){
        mockedLogic = mock(Logic.class);
        daof.setLogic(mockedLogic);
    }
    /*
     * Test of login method in DAOFacade.java.
     * Test verifies that method delegates to right method in lower-layer
     * class called Logic.java exactly once with correct set of paramters.
     */
    @Test
    public void testLogin() throws Exception{
        // Run method login in DAOFacade.java
        daof.login("Mustafa", "Mus");
        // Make verification on method login in Logic.java
        verify(mockedLogic, times(1)).login("Mustafa", "Mus");
    }
}
```

Figure 7 – *DAOFacade.java*. This can be used as a template.

With the test class provided, it is now time to run the test. *DAOFacadeTest* can also be accessed directly at its test package called *controller*.

1.5 Executing a test

To execute the test, right click on the test class at the project tree and choose *Run File* (Figure 8). Alternatively, all test classes under the folder *Test Packages* can be run simultaneously by right-clicking on the project instead and chose *Test*.

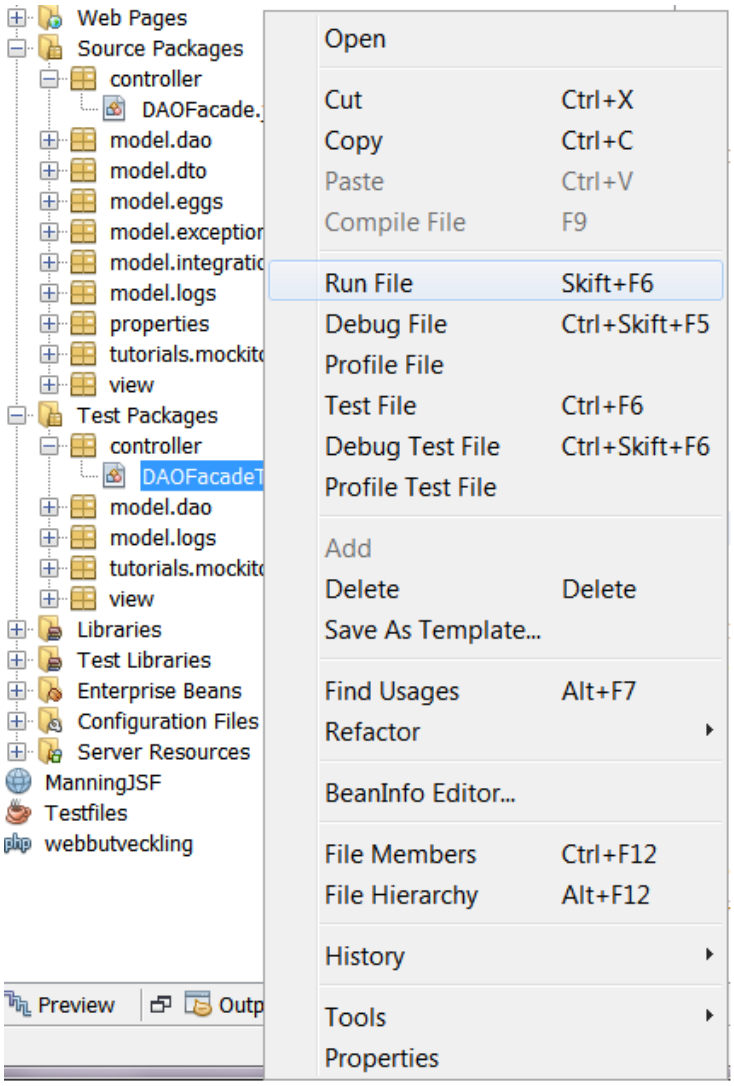

Figure 8 – How to run a test class in NetBeans IDE.

A new window will appear showing the test results at the lower left in NetBeans (Figure 9).

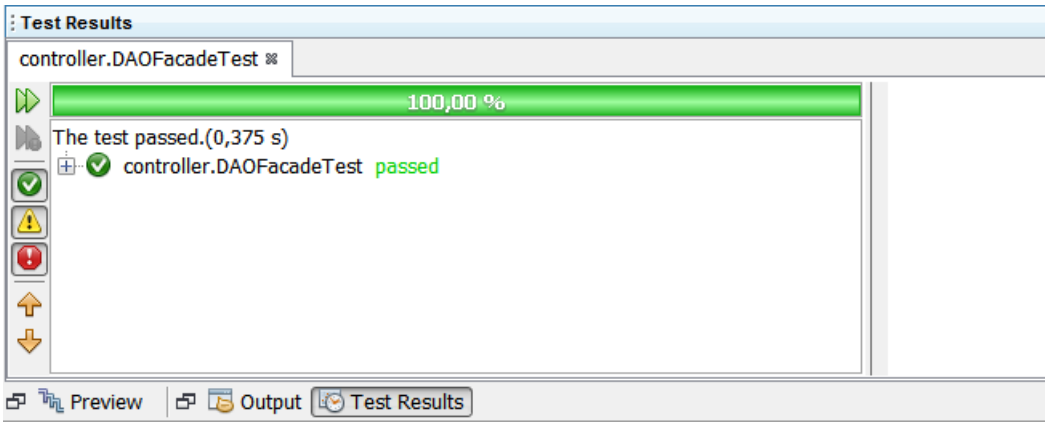

Figure 9 – Results from a passed test.

1.6 The Mockito API

Please disregard any project specific naming such as class names and package names in the following example. This example was coded for the sole purpose of showing the Mockito API in a tutorial. The goal with this example is to familiarize the reader with the structure of the code more and to see how Mockito is being used. The example also uses the method *verify()* which should now be familiar to the reader. This is important because this section focuses more on the theory behind the Mockito API and establishes the terminology used when talking about unit testing in general.

The test example illustrated in Figure 10 below, uses a method from the Mockito framework called *verify()*. As mentioned in section 1.4, *verify()* can be used whenever a verification of some sort needs to be done, however *verify()* is often used to check that a certain method call invokes the right method in another class. Dependencies between classes and method calls between them are verified. This type of testing is called integration testing. During such a test, the interactions between modules are tested. A module can be different things depending on the nature of the test. In this case, a module is just a java class. More specifically, it is *ClassA.java*.

This class is dependent of *ClassB.java* in such a way that a the only method found in *ClassA*, *methodA()* delegates the work to the one method in *ClassB* called *methodB*. *ClassA* is essentially a facade over *ClassB*, much like the case of *DAOFacade.java* and *Logic.java* in section 1.4. The example here, just as before, verifies that the method in *ClassB* was actually called from the method in *ClassA*.

In order to make such a test, dependencies between the two classes need to be set up. This is where Mockito is used. With Mockito, the dependencies can be mocked by creating what is called a mocked object of a class for which the tested class is dependent upon. In the example, *ClassA* is dependent on *ClassB* because its method invokes a method in *ClassB*. This is why *ClassB* is mocked. It is recommended that the comments in the code is read and understood (Figure 10).

```java
public class ClassATest
{
    /*
     * Mock out dependency to ClassB using Mockito. If static imports are used,
     * it is enough with just typing mock instead of Mockito.mock.
     */
    ClassB b = mock(ClassB.class);

    /*
     * Create object of ClassA that receives an object of classB. In this case,
     * a mocked object.
     */
    ClassA a = new ClassA(b);

    /*
     * Test run of methodA, of ClassA. Test verifies different things about
     * methodB in classB. This is done by mocking dependencies to ClassB.
     */
    @Test
    public void testMethodA1() throws Exception
    {
        // Call methodA in ClassA.
        a.methodA("anyString");

        // Verify that methodB was actually invoked with the correct
        // parameter when methodA was called.
        verify(b).methodB("anyString");
    }
}
```

Figure 10 – Complete source code for test class.

The test class above tests the only method found in the SUT. SUT stands for System under test and it is an abbreviation used to refer to whatever is being tested. As a consequence, an SUT can mean different things depending on the test case. In this case however, the SUT refers to the java class that is being tested, namely *ClassA*. Furthermore, in section 1.4 the SUT would be *DAOFacade*.

The method in *ClassB* does nothing. The SUT, *ClassA*, has a constructor that receives an object of *ClassB* which is used to access the method found in *ClassB*. In the test class, instead of an object of the real *ClassB*, a mocked object is sent to *ClassA* using Mockito in order to mock the dependencies between *ClassA* and *ClassB*. It is important to realize that unit tests are run in a separate container to avoid contamination of the project. This way, the behavior of the project is in no danger of being affected by the tests.

Figure 11 illustrates the constructor found in the SUT, **(a)** and how this is used by the test class to mock out the dependencies in the test container, **(b)**.

```
/**
 * Constructor of ClassA receives
 * an object of ClassB
 * @param b
 */
ClassA(ClassB b)
{
    this.b = b;
}
```

(a)

```
/*
 * Mock out dependency to ClassB
 * using Mockito. If static imports
 * are used, it is enough with just
 * typing mock instead of
 * Mockito.mock.
 */
ClassB b = mock(ClassB.class);
/*
 * Create object of ClassA that
 * receives an object of classB.
 * in this case a mocked object.
 */
ClassA a = new ClassA(b);
```

(b)

Figure 11 – (a) shows the constructor in the SUT. (b) Shows the test class using the constructor to pass a mocked object to the SUT.

In order for the test to run, dependencies to *ClassB* need to be mocked first. There are several ways to do this. One way is shown in Figure 12.

```
ClassB b = mock(ClassB.class);
```

Figure 12 – One way to mock when static imports are used. If not, the Mockito class must be explicitly referenced.

Another way to mock is to use the Mockito annotation for mocking *@Mock* (Figure 13).

```
@Mock
private ClassB b;
```

Figure 13 – One way to mock using annotations.

To enable Mockito specific annotations, the code must specify how to run the test class. This is done using the JUnit annotation *@RunWith* (Figure 14). The annotation is put on top of the class definition.

```
@RunWith(MockitoJUnitRunner.class)
public class DAOFacadeTest
{
```

Figure 14 – JUnit Annotation *@RunWith* to enable Mockito annotations.

Please observe that no expectation was set up before the test. This is done after the fact and it is a feature unique to Mockito. Usually, an expectation is set up before the method is called, where the expected outcome is specified.

This structure is known as the expect-run-verify pattern and it is one that does not need to be followed when using Mockito, making the test more intuitive. In the case above, the test only verifies that *methodB* was invoked with the correct parameter, *"anyString"*, when *methodA* was called. This was the expected outcome of the test and Mockito offers many options to tailor the verification.

For example, the test can check that a specific method was invoked a specified number of times. Using the same example classes, such verification would look like this (Figure 15).

```
verify(b, Mockito.times(1)).methodB("anyString");
```

Figure 15 – Verify that *methodB* was invoked exactly one time with the parameter "*anyString*".

Several verifications can be made in the test method by simply typing verify again with new logic, as seen in Figure 16.

```
verify(b).methodB("anyString");
verify(b, Mockito.times(1)).methodB("anyString");
```

Figure 16 – Several verifications in succession.

A test class can have more than one test method. This is accomplished by adding the JUnit annotation *@Test* right above the intended test method. This is illustrated by Figure 17.

```
@Test
public void testMethodA1() throws Exception
{
    // Call methodA in ClassA.
    a.methodA("anyString");

    // Verify that methodB was actually invoked with the correct
    // parameter when methodA was called.
    verify(b).methodB("anyString");
}

@Test
public void testMethodA2 ()throws Exception
{
    // Call methodA in ClassA.
    a.methodA("anyString");

    // Verify that methodB was actually invoked with the correct
    // parameter when methodA was called.
    verify(b, Mockito.times(1)).methodB("anyString");
}
```

Figure 17 – Several test methods using the JUnit annotation @Test.

The way of passing a mocked object as illustrated in Figure 11 can become problematic when the code base becomes more complex and data structures such as EJB's are used. In some cases, like in Java EE Web projects with the MVC-model-structure, constructors for passing class objects may not be a viable option due to the need of encapsulating and having low coupling between the different layers. To solve this issue, an injection point that sets the EJB to be the mocked object can be used to pass it to the SUT when running a test.

This means altering the SUT by adding this ability. In the context of testing, this violates the rule of not altering the SUT just for the sake of testing it. However, by adding this ability to the SUT, neither its logic nor the overall structure is changed. It is therefore, an acceptable solution to the problem and it is how it is done in *DAOFacadeTest* in section 1.4. Figure 18 shows such a case, where dependencies between two classes at different layers of the system are set up using EJB's and not by passing objects of classes using constructors.

```
/**
 * JSF Managed bean used to check and store the user identity.
 *
 * @author admin
 */
@ManagedBean
@SessionScoped
public class AuthenticationBean implements Serializable {

    @EJB
    private DAOFacade daof;
```

Figure 18 – *AuthenticationBean.java* in the upper layer of the project has access to methods in *DAOFacade* in the next, lower layer.

In this case, a class called *AuthenticationBean.java*, located in the upper layer of the Java EE Web project, passes all methods to the lower layers of the system via the facade, *DAOFacade*. *DAOFacade* is the controller and it is located one layer down, in the source package *controller*. *AuthenticationBean* does not see the lower layers. Instead, it sees only a facade over it which it interacts with, much like what the SUT *ClassA* does with *ClassB*. The difference here is that, instead of having a constructor in *AuthenticationBean* that receives an object of the facade, the facade is set as an EJB in *AuthenticationBean* with the *@EJB* annotation (Figure 18 again). This yields better encapsulation of the code layers and lower coupling between them, some of the goals with the MVC-model.

Now, a test is created under these circumstances. The SUT is now *AuthenticationBean* and the method in it that will be tested is called *login()*. This method passes login information to the lower layers by calling a method in *DAOFacade* and not any methods directly from the lower layers. Those methods are hidden from *AuthenticationBean*. Since *DAOFacade* is an EJB in *AuthenticationBean* this can be done, even though *AuthenticationBean* does not have constructor for *DAOFacade* as in the case of *ClassA* in Figure 11 - (a). Figure 19 illustrates the portion of the code from the method that shows how the work is delegated to *DAOFacade*.

```
public String login()
{
    if (daof.login(username, password) == 0)
    {
```

Figure 19 – login method in the new SUT, *AuthenticationBean.java,* passing on to *login()* in *DAOFacade.java*.

The test verifies that the method *login()* in *DAOFacade* was actually called exactly once with two specific strings a parameters. This is shown in Figure 20.

```
/*
 * Tests the method used when admins try to login.
 */
@Test
public void TestAdminLogin()
{
    authBean.login();
    verify(mockedDAOFacade, times(1)).login("Terminator",
                                            "c00lk1ll3rb0y#96");
}
```

Figure 20 – The test for the SUT.

The depending class, *DAOFacade*, is mocked as usual in the test class (Figure 21).

```
@Before
public void setUp()
{
    // Mock the class DAOFacade
    mockedDAOFacade = mock(DAOFacade.class);

    // Pass the mocked object to the SUT.
    authBean.setDAOFacade(mockedDAOFacade);
```

Figure 21 – *DAOFacade.java* mocked as in a previous test example (Figure 6).

As mentioned before, the difference lies in the SUT. In order for the test to pass, the SUT must be able to receive a mocked object somehow. For this reason, an injection point is added to the SUT that explicitly sets an object of type *DAOFacade* equal to the EJB for *DAOFacade* in the SUT (Figure 22).

```
public void setDAOFacade(final DAOFacade daof) {
    this.daof = daof;
}
```

Figure 22 – The added code in the SUT.

In the test class, after mocking *DAOFacade*, the mocked object is set to be equal to the EJB in the SUT through this injection point. This way, the SUT uses the mocked object instead of the EJB. If this injection point is not present, the SUT tries to pass on to the method in the real *DAOFacade* instead of using the mocked object. This results in a null pointer exception since the EJB is not defined within the context of the test. The dependencies have not been passed to the SUT, even though *DAOFacade* is mocked.

This example showed how to mock dependencies to EJB's. Mockito offer more ways of mocking other types of dependencies such as different types of contexts but examples on how to do this is not covered in this tutorial.

End of tutorial.