Unit testing with JUnit and EasyMock

We all have it. It's that piece of code in our project that everyone is afraid to change. It's so confusing no one truly understands what is going on. We are all afraid that if we do change it, we'll break it. You have just discovered my favorite reason for writing unit tests. But how do you write a unit test? What exactly is a unit test? How do I handle dependencies? And how is writing more code going to make my existing code better? This tutorial will show you how.
What is a unit test?

For the case of this tutorial, we'll define a unit test as a test of a single isolated component in a repeatable way. Let's go thru that one section at a time to get a clearer idea of what goes into a unit test.

"a test". This means to verify something is correct. In order for us to have a valid unit test, we need to actually validate that after a start condition A, an end condition B exists. "...a single isolated component...". This is what separates a unit test from other types of tests. In order for it to be a unit test, it must test something in isolation, aka without dependencies. The reason for this is that we are testing the component itself and not it's interaction with other components (that is an integration test). Finally, although most definitions don't include this piece, "...in a repeatable way" is a very important piece of the definition. It's one thing to run a test that passes. It's quite different to have something you can run in a repeatable manor at any point to see if changes you made effected how the component behaves. For example, if you choose to do some refactoring to improve performance, can you rerun your unit test to verify that you didn't change the behavior of the component.

Setup

I will be using Eclipse 3.3 Europa to do this tutorial. To begin, create a new java project and call it JUnitTutorial. Right click on your new project and select New --> Folder. Name it lib and click Finish. Usually you don't want to package your test code with your regular code, so let's make an additional source directory, test. To do that, right click on your new project and select Properties. Select Java Build Path from the available options. In the Java Build Path window, click Add Folder. From the Add Folder dialog, select Create New Folder, name it test and click Finish. Next we need to add JUnit to our build path. Since it comes with Eclipse, all we need to do is to go to the Libraries tab, click the button Add Library, select JUnit and click Next. Select JUnit 4 and click Finish. Click ok to exit the Preferences window.

We will also need to download and add the EasyMock jar files to our project. You can find the jars here. Once you download the zip file (we are using version 2.3 for this tutorial), extract the easymock.jar file and place it in the lib folder you created earlier. In Eclipse, right click on your project and select Properties. On the menu to the left, click Java Build Path and select the Libraries tab. Click the button Add Jar on the right. In the window that pops up, add the easymock.jar and click Ok. Click Ok to close the Properties window. You should now be ready to start your development.

The requirements

In test driven design, we develop the unit test before the functionality. We write a test that verifies that the class should do X after our call. We prove that the test fails, we then create the component to make the test pass. In this case, we are going to create a service with a method that authenticates a user. Below is a class diagram of the scenario.

The interfaces

We will start our coding by defining two interfaces, LoginService and UserDAO We will implement LoginService, however since in this tutorial UserDAO will be mocked, we won't bother implementing it right now. For LoginService, we have a single method that takes a String userName and String password and returns a boolean (true if the user was found, false if it was not). The interface looks like this:

```
/**
* Provides authenticated related processing.
*/
public interface LoginService {

    /**
    * Handles a request to login.  Passwords are stored as an MD5 Hash in
    * this system.  The login service creates a hash based on the paramters
    * received and looks up the user.  If a user with the same userName and
    * password hash are found, true is returned, else false is returned.
    *
    * @parameter userName
    * @parameter password
    * @return    boolean
    */
    boolean login(String userName, String password);
}
```

The UserDAO interface will look very similar to the LoginService. It will have a single method that takes a userName and hash. The hash is an MD5 hashed version of the password, provided by the above service.

```
/**
* Provides database access for login related functions
*/
public interface UserDAO {

    /**
    * Loads a User object for the record that
    * is returned with the same userName and password.
    *
    * @parameter userName
    * @parameter password
    * @return    User
    */
    User loadByUsernameAndPassword(String userName, String password);
}
```

The test case

Before we begin development, we will develop our test. Tests are structured by grouping methods that perform a test together in a test case. A test case is a class that extends junit.framework.TestCase. So in this case, we will begin by developing the test case for LoginService. To start, in your test directory, create a new class named LoginServiceTest and make it extend junit.framework.TestCase.

The lifecycle of a test execution consists of three main methods:

- public void setUp()

setUp is executed before each of the test. It is used to perform any setup required before the execution of your test. Your implementation will override the default empty implementation in TestCase.

- public void testSomething()

testSomething is the actual test method. You may have many of these within a single test case. Each one will be executed by your test runner and all errors will be reported at the end.

- public void tearDown()

tearDown is executed after each test method. It is used to perform any cleanup required after your tests.

So to begin flushing out our test case, we'll start with the setUp method. In this method, we'll instantiate an instance of the service to be tested. We'll also create our first mock object, UserDAO. You can see the source of our test below.

```
import junit.framework.TestCase;
import static org.easymock.EasyMock.createStrictMock;
import static org.easymock.EasyMock.expect;
import static org.easymock.EasyMock.replay;
import static org.easymock.EasyMock.verify;
import static org.easymock.EasyMock.eq;
/**
* Test case for LoginService.
*/
public class LoginServiceTest extends TestCase{

    private LoginServiceImpl service;
    private UserDAO mockDao;

    /**
    * setUp overrides the default, empty implementation provided by
    * JUnit's TestCase.  We will use it to instantiate our required
    * objects so that we get a clean copy for each test.
    */
    @Override
    public void setUp() {
        service = new LoginServiceImpl();
        mockDao = createStrictMock(UserDAO.class);
        service.setUserDAO(mockDao);
    }
}
```

EasyMock works by implementing the proxy pattern. When you create a mock object, it creates a proxy object that takes the place of the real object. The proxy object gets it's definition from the interface you pass when creating the mock. We will define what methods are called and their returns from within our test method itself.

When creating a mock object, there are two types, a mock and a strict mock. In either case, our test will tell the mock object what method calls to expect and what to return when they occur. A basic mock will not care about the order of the execution of the methods. A strict mock, on the other hand, is order specific. Your test will fail if the methods are executed out of order on a strict mock. In this example, we will be using a strict mock.

The next step is to create our actual test method (for reference, we will not be implementing a tearDown method for this test case, it won't be needed in this example). In our test method, we want to test the following scenario:

Even with the very basic method we want to test above, there are still a number of different scenarios that require tests.

We will start with the "rosy" scenario, passing in two values and getting a user object back. Below is the source of what will be our new test method.

```
...
/**
* This method will test the "rosy" scenario of passing a valid
* username and password and retrieveing the user. Once the user
* is returned to the service, the service will return true to
* the caller.
*/
public void testRosyScenario() {
    User results = new User();
    String userName = "testUserName";
    String password = "testPassword";
    String passwordHash =
        "ýö &  I7ýýýNi=.";
    expect(mockDao.loadByUsernameAndPassword(eq(userName), eq(passwordHash)))
        .andReturn(results);

    replay(mockDao);
    assertTrue(service.login(userName, password));
    verify(mockDao);
}
...
```

So let's go thru the code above. First, we create the expected result of our DAO call, results. In this case, our method will just check to see if an object was returned, so we don't need to populate our user object with anything, we just need an empty instance. Next we declare the values we will be passing into our service call. The password hash may catch you off guard. It's considered unsafe to store passwords as plain text so our service will generate an MD5 hash of the password and that value is the value that we will pass to our DAO.

The next line is a very important line in our test that alot happens, so let's walk thru it step by step:

- expect(mockDao.loadByUsernameAndPassword()
This is a call to the static method EasyMock.expect. It tells your mock object to expect the method loadByUsernameAndPassword to be called.

- eq(userName), eq(passwordHash)
This code isn't always needed. When EasyMock compares the values passed to the method call, it does and == comparison. Because we are going to create the MD5 hash within our method, an == check will fail, so we have to use one of EasyMock's comparators instead. The eq comparator in this case will compare the contents of the string using it's .equals method. If we were not doing the MD5 hash, this line would be expect(mockDao.loadByUsernameAndPassword(userName, password).andReturn(results);

- .andReturn(results);
This tells our mock object what to return after this method is called.

The final three lines are the ones that do the testing work. replay(mockDao); tells EasyMock "We're done declaring our expectations. It's now time to run what we told you". assertTrue(service.login(userName, password)); does two things: executes the code to be tested and tests that the result is true. If it is false, the test will fail. Finally, verify(mockDao); tells EasyMock to validate that all of the expected method calls were executed and in the correct order.

So that's it for the test. Now all we have to do is write the code to make it pass. You can find that below.

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;


public class LoginServiceImpl implements LoginService {

    private UserDAO userDao;

    public void setUserDAO(UserDAO userDao) {
        this.userDao = userDao;
    }

    @Override
    public boolean login(String userName, String password) {
        boolean valid = false;
        try {
            String passwordHash = null;
            MessageDigest md5 = MessageDigest.getInstance("MD5");
            md5.update(password.getBytes());
            passwordHash = new String(md5.digest());

            User results =
                userDao.loadByUsernameAndPassword(userName, passwordHash);
            if(results != null) {
                valid = true;
            }
        } catch (NoSuchAlgorithmException ignore) {}

        return valid;
    }
}
```

Conclusion

So that is it. I hope this gives you a more in depth view into JUnit and EasyMock. Unit testing is something that once you get used to it, makes you code better, provides you with a safety net for future refactoring and protects you from being burned by API changes. I strongly encourage that you give it a try. Until next time.

Attachments

Eclipse Project: JUnitClass.zip