The Concept of Mocking

To someone who is new to unit testing, the idea of mock objects can be confusing to say the least. I have covered in previous tutorials how to use various mock object frameworks (EasyMock and jmockit). However in this tutorial, we will focus on the concept of mocking in general. What is a mock object? What is it used for? Why can't I mock object XYZ? Let's look into these questions and maybe clear a bit of the air on the use of mock objects.

Not all code is self contained

When we learn to program, our objects are usually self contained. Any hello world has no dependencies on outside classes (System.out aside) and neither do many of the other classes we write in the process of learning a language. However, in the real world, software has dependencies. We have action classes that depend on services and services that depend on data access objects (DAOs) and the list goes on.

The idea of unit testing is that we want to test our code without testing the dependencies. This test allows you to verify that the code being tested works, regardless of it's dependencies. The theory is that if the code I write works as designed and my dependencies work as designed, then they should work together as designed. The code below would be an example of this:

```
import java.util.ArrayList;

public class Counter {
    public Counter() {
    }

    public int count(ArrayList items) {
        int results = 0;

        for(Object curItem : items) {
            results ++;
        }

        return results;
    }
}
```

I know the above example is about as simple as you get, but it illustrates the point. If you wanted to test the method count, you would write at test that addressed how the count method works. You aren't trying to test that ArrayList works because you assume that it has been tested and works as designed. Your only goal is to test your use of ArrayList.

Now let's look at a slightly more realistic example. Let's consider the code below:

```
public class MichaelsAction extends ActionSupport {

    private LookupService service;

    private String key;

    public void setKey(String curKey) {
        key = curKey;
    }

    public String getKey() {
```

```
        return key;
    }

    public void setService(LookupService curService) {
        service = curService;
    }

    public String doLookup() {

        if(StringUtils.isBlank(key)) {
            return FAILURE;
        }

        List results = service.lookupByKey(key);

        if(results.size() > 0) {
            return SUCCESS;
        }

        return FAILURE;
    }
}
```

If we wanted to test the doLookup method in the above code, we would want to be able to test it without testing the lookupByKey method. For the sake of this test, we assume that the lookupByKey method is tested to work as designed. As long as I pass in the correct key, I will get back the correct results. In reality, we make sure that lookupByKey is also tested if it is code we wrote. We now have a dilemma. How do we test doLookup without executing lookupByKey?

Enter Mock Objects

The concept behind mock objects is that we want to create an object that will take the place of the real object. This mock object will expect a certain method to be called with certain parameters and when that happens, it will return an expected result. Using the above code as an example, let's say that when I pass in 1234 for my key to the service.lookupByKey call, I should get back a List with 4 values in it. Our mock object should expect lookupByKey to be called with the parameter "1234" and when that occurs, it will return a List with four objects in it.

How Mock Objects Work

There are many different mocking frameworks in the Java space. I am not going to discuss any of them in great detail here. However, I will discuss how they work and design considerations you need to take in mind because of their implementations.

There are essentially two main types of mock object frameworks, ones that are implemented via proxy and ones that are implemented via class remapping. Let's take a look at the first (and by far more popular) option, proxy.

A proxy object is an object that is used to take the place of a real object. In the case of mock objects, a proxy object is used to imitate the real object your code is dependent on. You create a proxy object with the mocking framework, and then set it on the object using either a setter or constructor. This points out an inherent issue with mocking using proxy objects. You have to be able to set the dependency up thru an external means. In other words, you can't create the dependency by calling new MyObject() since there is no way to mock that with a proxy object. This is one of the reasons Dependency Injection frameworks like Spring have taken off. They allow you to inject your proxy objects without modifying any code.

The second form of mocking is to remap the class file in the class loader. The mocking framework jmockit is the only framework I am aware of that currently exploits this ability for mock objects. The concept is relatively new (since JDK 1.5 although jmockit supports jdk1.4 thru other means as well) and is provided by the new java.lang.Insturment class. What happens is that you tell the class loader to remap the reference to the class file it will load. So let's say that I have a class MyDependency with the corresponding .class file called MyDependency.class and I want to mock it to use MyMock instead. By using this type of mock objects, you will actually remap in the classloader the reference from MyDependency to MyMock.class. This allows you to be able to mock objects that are created by using thew new operator. Although this approach provides more power than the proxy object approach, it is also harder/more confusing to get going given the knowledge of classloaders you need to really be able to use all its features.

Conclusion

Mock objects are a very valuable tool in testing. They provide you with the ability to test what you write without having to address dependency concerns.