Making a Mockery

Ivan Moore
Connextra Ltd
Studio 312, Highgate Studios
53-79 Highgate Road
London NW5 1TL, England
+44 (0)20 7692 9898
ivan@connextra.com

Sebastian Palmer
Connextra Ltd
Studio 312, Highgate Studios
53-79 Highgate Road
London NW5 1TL, England
+44 (0)20 7692 9898
seb@connextra.com

Abstract

Mock objects are used by Extreme Programmers to write isolated tests, and result in classes that clearly express their dependencies.

Some would assume that using mock objects requires writing lots of extra code. However, a number of tools have been developed for the automatic generation of mock object code.

This paper examines four such tools for Java. Each takes a different approach. The use of any of these tools reduces the amount of code that has to be hand written to implement mock objects, making the use of mock objects simple and efficient in developer time.

This paper will not specifically investigate the existing collection of mock objects provided by the MockObjects project framework, but is about tools for creating new mock objects.

Keywords

Mock objects, unit testing, testing.

INTRODUCTION

"If a collection of rocks is a rockery, a collection of mock objects must be a mockery."[1]

Mock objects[2] are used by many Extreme Programmers[3]. Some programmers consider them to require too much extra coding or to be too difficult. One of the features of the mock object approach is that mock objects, known as "mocks" in this paper, have uniform functionality and predictable code. This enables the development of tools for the automatic generation of mocks.

This paper examines four tools for creating mocks for Java, which collectively will be called CAMP tools (Computer Aided Mock Production. CAMP is a concentrated liquid coffee from McCormick Foods[4]).

This paper assumes some familiarity with the concept of mock objects, but nevertheless, the authors believe that CAMP tools themselves are instructional about mock objects.

THE CAMP TOOLS

The tools described are: EasyMock[5], Extender[6][7], MockCreator[8] and MockMaker[9]. Mockry[10], which became available after this paper was submitted, has not been included. Only the use of the tools will be examined, not their implementation. In order to provide a comparison between the mocks produced by these tools, a simple example will be demonstrated using each tool.

THE MOCK OBJECTS FRAMEWORK

The MockObjects project[11] provides a framework and naming conventions for hand writing mock objects, and pre-written mock objects for various APIs, for example, servlet programming. The MockObjects project framework is used by the mocks produced by MockMaker and MockCreator. The other tools take a different approach. None of the tools examined in this paper are part of the MockObjects project.

THE EXAMPLES

To compare the CAMP tools, an example of trying to write a test using mocks generated by each tool will be described.

The example test code is for a MultiReplacer; part of a (fictional) tool for globally replacing a collection of strings with other strings, in a collection of files.

One of the things that we want to test is that a MultiReplacer correctly reads a configuration string to extract the required search and replacement strings, then applies those changes to a file and reports how many replacements were made.

One way of testing this would be to call a MultiReplacer with some real files, and to check that the files were changed in the expected ways.

If instead of having the MultiReplacer actually modify files itself, we make it responsible only for reading the configuration string and then using a SearchAndReplacer (that we will define) for modifiying files, we can test the reading of the configuration string in isolation from the file modification code. This means that all test cases can

be covered easily, test failures are attributable to a single object and there are no file system dependencies in the tests of the MultiReplacer.

Writing tests before code provides a basis for deriving the interactions of objects. In this case, we want to be sure that the MultiReplacer, given a configuration string (each line of which defines a string to search for and string to replace it with delimited by a '|' character) will make the appropriate calls on a SearchAndReplacer (implemented as a mock).

The test follows, using MockObjects project conventions:

```
public void testDoReplacements() {
String config = "xyz|abc\n"+
                "1|2";
MockSearchAndReplacer aMockSearchAndReplacer = new MockSearchAndReplacer();
aMockSearchAndReplacer.addExpectedReplace("xyz","abc");
aMockSearchAndReplacer.setupReplace(5);
aMockSearchAndReplacer.addExpectedReplace("1","2");
aMockSearchAndReplacer.setupReplace(7);
MultiReplacer aMultiReplacer = new MultiReplacer();
int numReplacements = aMultiReplacer.doReplacements(config, aMockSearchAndReplacer);
aMockSearchAndReplacer.verify();
assertEquals("expected MultiReplacer to think it has done 12 replacements",(5+7), numReplacements);
}
```

The test implies a class MultiReplacer that has a method:

```
public int doReplacements(String, SearchAndReplacer)
```

and an interface SearchAndReplacer that has a method:

```
public void replace(String from, String to);
```

This paper will not deal with the implementation of the real SearchAndReplacer (which does a search and replacement to the contents of a file), but will instead examine how this example test would need to be written for the use of the mock SearchAndReplacer generated by each CAMP tool.

In order to generate the mock implementation, EasyMock and MockCreator need an interface, SearchAndReplacer. Extender and MockMaker need either an interface or a class. As can be seen from the example, working out the interface implied by a test is usually quite straightforward.

The classes produced by EasyMock, MockCreator and MockMaker provide slightly different ways of setting expectations about the number of times a method is called, the parameters methods are called with and to setup return values for methods. Extender does not directly provide such code, but rather provides an empty implementation of a class that can be (manually) overridden to provide such behaviour.

HAND WRITTEN USING MOCKOBJECTS FRAMEWORK

The MockObjects project provides useful classes for writing mocks by hand, such as classes for setting and verifying expectations about the number of times a method is called or the parameters it is called with. Writing mocks by hand allows for great flexibility in the mock code. Mocks can be customized for ease of writing tests and clarity of test code. To give an example: where a method on a mock takes a number of parameters and the purpose of our test is ensuring that one of these parameters is correct (and the values of the other parameters don't matter for that test), we can express that more clearly in the code if we only set up that one expectation; most of the tool-based approaches would require us to set expectations for all of the parameters.

Although writing mocks by hand allows for flexibility, it can also be very repetitive. In most cases a call to a mock simply increments a call counter, checks parameter values and returns a preset return value.

EASYMOCK

The EasyMock approach removes the need for writing a mock at all. To use EasyMock, the EasyMock jar file has to be included on the classpath, and then EasyMock mocks can be written using classes from this jar (as shown in the example code below) without any further steps being necessary. EasyMock tests create mocks dynamically at test run time, with each EasyMock object having two distinct phases.

In the setup phase, an EasyMockControl is used to set up the mock. Parameter expectations are set up by making calls on the mock in setup mode. For each call, the EasyMockControl is then set up with return values and call counts for the mock. The EasyMockControl is then activated and the mock works in the normal way as the test is run, throwing Exceptions when expectations are not fulfilled.

The approach is very elegant, but it does not allow the programmer to customize the mocks, and only works in JDK 1.3.1 and later.

The MultiReplacer test would be written as follows:

```
public void testDoReplacement() throws Exception {
String config = "xyz|abc\n"+
                "1|2";
MockControl aMockSearchAndReplacerControl =
EasyMock.controlFor(SearchAndReplacer.class);
SearchAndReplacer aMockSearchAndReplacer = (SearchAndReplacer)
        aMockSearchAndReplacerControl.getMock();
aMockSearchAndReplacer.replace("xyz","abc");
aMockSearchAndReplacerControl.setReturnValue(5);
aMockSearchAndReplacer.replace("1","2");
aMockSearchAndReplacerControl.setReturnValue(7);
```

```
aMockSearchAndReplacerControl.activate();
MultiReplacer aMultiReplacer = new MultiReplacer();
int numReplacements = aMultiReplacer.doReplacements(config,
aMockSearchAndReplacer);
aMockSearchAndReplacerControl.verify();
assertEquals("expected MultiReplacer to think it has done 12
replacements",(5+7), numReplacements);
}
```

EasyMock requires the setting up of expectations for every method call. That is, if 'doReplacements' also calls another method on the MockSearchAndReplacer in addition to the calls of 'replace' that we are testing for, then the MockSearchAndReplacer will throw an AssertionFailedError. If we want to allow our implementation of MultiReplacer to make some other method calls on the MockSearchAndReplacer, then the MockSearchAndReplacer needs to be set to expect those method calls even if it is not important to the intent of the test. In our view, this is both a strength and a weakness. The programmer is forced to set expectations for every method call. Clearly this is a rigorous approach, but it means that the purpose of a test is easily obscured by over-specification.

Ideally, a test should specify only the things that are intended to be tested; in fact, this is one of the motivations for using mock objects. Using the EasyMock approach means that all method calls must be specified, including those that are not relevant to the purpose of the test. It is our experience that tests are harder to understand if they have to specify more things than you intended to test.

Removing the need to write mock code is elegant and eliminates problems associated with maintaining generated code. As the mocks are generated dynamically, and no extra source created, EasyMock mocks cannot be customised. This restriction has a benefit; it enforces standards so EasyMock mocks always behave in the same way.

EXTENDER

Extender doesn't provide much structure so it's difficult to say what the best way to use it is. Extender is used as a command line tool; given the name of a class/interface that exists on the classpath, Extender outputs the source for a subclass/implementing class, which can then be saved to a file. The classes produced by Extender simply implement/override all the methods of the interface/superclass to throw a RuntimeException, thus preventing accidental use of any of the methods.

To write a test that uses a mock based on these generated classes, the programmer must provide a subclass overridden with the appropriate implementations of the relevant methods from the superclass. The simplest way to do this is usually by creating an anonymous inner class, since the number of methods to be overridden will usually be quite small.

The test code shown has been written to be as close in behaviour as the code for the other tools as possible:

```
public void testReadConfigExtender() throws Exception {
String config = "xyz|abc\n"+
                "1|2";
SearchAndReplacer aMockSearchAndReplacer = new
ExtenderMockSearchAndReplacer(){
        int callNumber = 0;
        public int replace(String arg1, String arg2) {
                callNumber++;
                switch (callNumber) {
                        case 1 : {
                        assertEquals("xyz", arg1);
                        assertEquals("abc", arg2);
                        return 5;
                        }
                        case 2 : {
                        assertEquals("1", arg1);
                        assertEquals("2", arg2);
                        return 7;
                        }
                }
        fail("shouldn't have been called >2 times");
        return 0;
        }
};
MultiReplacer aMultiReplacer = new MultiReplacer();
int numReplacements = aMultiReplacer.doMutations(config,
aMockSearchAndReplacer);
assertEquals("expected MultiReplacer to think it has done 12
replacements",(5+7), numReplacements);
}
```

Extender therefore does not automatically generate mock implementations. It just implements a class that can be overridden just for those methods needed for the mock for a test. The act of writing the mock (usually setting up expectations and verifying them) is left to the programmer. The programmer can choose to use the MockObjects project framework, or implement the overridden methods any other way, for example, in the way shown above. Extender provides a way of making mocks for concrete classes whose interface the programmer has no control over. Examples of such code can be found in the JDK libraries, such as java.net.URL and java.io.InputStream.

MOCKCREATOR

MockCreator is a tool for VisualAge for Java that generates and saves a mock class for an interface. Using MockCreator is very straightforward. A menu item is available on interfaces that creates a mock implementation in a manually selected package. The user does not have to save the source to a file manually; the new class is added into the VisualAge for Java workbench automatically.

The mocks generated by MockCreator require the tests to be written slightly differently than the MockObjects project conventions (arguably neater). The test code for our example would be:

```
public void testDoReplacement() throws Exception {
String config = "xyz|abc\n"+
                "1|2";
CreatorMockSearchAndReplacer aMockSearchAndReplacer =
new CreatorMockSearchAndReplacer();
aMockSearchAndReplacer.expectReplace("xyz","abc",5);
aMockSearchAndReplacer.expectReplace("1","2",7);
MultiReplacer aMultiReplacer = new MultiReplacer();
int numReplacements = aMultiReplacer.doReplacements(config,
aMockSearchAndReplacer);
aMockSearchAndReplacer.verify();
assertEquals("expected MultiReplacer to think it has done 12
replacements",(5+7), numReplacements);
}
```

The mocks created by MockCreator use the MockObjects project framework but not it's conventions. Instead of two calls, i.e. 'setExpected' and 'setup', as per MockObjects project conventions, only one 'expect' call is required that both sets the expectation and the return value.

MockCreator mocks do not require expectations to be set for all methods that all called. Using the MockObjects project framework, if an expectation is not set then none is attempted to be verified; i.e. not setting an expectation that method 'foo' is called means that whether 'foo' is called or not does not effect the running of the test. To set an expectation that 'foo' is not called is a different matter and is an expectation that needs to be explicitly set.

The code generated by MockCreator can be modified by hand if necessary.

MOCKMAKER

MockMaker can be used either as a command line tool or through a simple GUI to create the source code for a mock class for either an interface or a class. It also provides integration with JBuilder, and integration with VisualAge for Java is due for release soon.

MockMaker can be configured to use the MockObjects project conventions. The source code for MockSearchAndReplacer would therefore allow the test to be written exactly the same as shown earlier for the hand written mock.

There are differences between the code produced by MockMaker and in the mocks in the MockObjects project framework in the naming of instance variables and the strings that are used in messages when tests fail, but these are of less importance than the conventions for the names of public methods.

As with MockCreator, MockMaker uses the MockObjects project framework and so does not require overspecifying of tests. Similarly, because code is generated, it can be customized as necessary.

The code generated by MockMaker could be neater, and it does not include all the import statements necessary for the generated code to compile unchanged, however the import statements can usually be worked out by an IDE.

SUMMARY OF DIFFERENT APPROACHES

Tool	Advantages	Disadvantages
EasyMock	No code to maintain Enforces standards of mocks	Only works for JDK1.3.1 or later Can require overspecifying of tests
Extender	Works for classes and interfaces	Does not generate much of the mock implementation
MockCreator	VisualAge integration makes it easy to use	Only available in VisualAge for Java
MockMaker	Works for older JDKs and not tied to IDE Follows MockObjects project conventions for public methods	Code generated could be improved and shorter

CONCLUSIONS

Writing mocks by hand is a repetitive and time-consuming task that can be automated. Most of the CAMP tools are useful in this regard.

Further advantages of the CAMP tools (except Extender) are that they enforce coding standards (in generated mocks) and produce simple mock objects of predictable and consistent behaviour.

Most of the tools provide implementations of mocks that increment a call counter, compare the parameters with expected parameters and, where appropriate, return some pre-set value or object.

Like all code, mocks should be refactored to remove redundancy and duplication. In practice, however, auto-

matic generation of mock code seems to discourage this. The approach with CAMP tools is usually a cycle of write test, make mock, run test to fail, write application code, refactor application code, leave mock alone.

An automatically generated mock contains a complete but simple implementation for a given interface; for example, including a method call counter for each method whether or not that method's call counts are tested in any test. Therefore, automatically generated mocks may contain more code than handwritten mocks, which usually evolve alongside the test code and which only contain code that is needed to make a specific suite of tests run.

The automatically-generated code may therefore contain redundant code. Large numbers of unused setup and setExpectation methods in a CAMP-generated mock suggests that not all test cases have been covered. We believe that mock code, because it performs in a very simple and predictable way, is different than production code in terms of refactoring. Production code should be refactored so that it can be changed and read easily. Automatically-generated mock code is easy to read because it is predictable and easy to change because a change to the interface means a predictable change to the generated mock implementation; all the programmer needs to do is regenerate the mock.

Some people have reported that debugging using CAMP generated mocks can be more difficult than with hand written mocks, because the generated code is not always very elegant, and for the reasons described earlier may contain much more code than you are really interested in. Furthermore, as EasyMock uses reflection to implement mocks, it might be more difficult to debug when using EasyMock mocks, particularly if the code you are trying to debug is itself reflective.

In test-first programming, the design of interfaces evolves with the test code. As new tests are written, new methods are identified and added to the interface.

Mocks for which a developer has the source code can be customized to make the tests more expressive and more flexible; for example, if you only want to test one of the parameters that a method is called with, the mock can be customized with a method for setting only that expectation.

The tools differ in the tradeoffs they make between the degree of control that programmers want through customization and the ease of changing the interface.

EasyMock accommodates changes to interfaces because it generates mocks dynamically; if an interface is changed in a backward compatible way then tests using EasyMock mocks do not need to be changed.

CAMP tools that generate and save the mock code (MockCreator and MockMaker) do not accommodate changes to interfaces so readily. MockCreator overwrites existing mock code, and existing mocks must be deleted each time the interface is modified. In a future release, MockMaker will work incrementally, allowing the programmer to customize the code of the mock after it has been generated, then add a method and rerun the tool without undoing those changes.

The EasyMock approach means that generated mocks cannot be customized. Perhaps this is a good thing: in our experience, it is rare for a mock to need to do anything other than check parameters, check call counts, return pre-set values or throw Exceptions. Our experience of mocks that do more than this is that they lead to confusion when trying to understand test code; mocks should not do very much.

Our experience has been that automatically generating mocks saves time and produces the mocks that we want: simple, consistent and predictable.

ACKNOWLEDGEMENTS

Special thanks to John Nolan for the title and the CAMP acronym. Thanks to Rachel Davies for comments on this paper, to users of MockMaker who have made useful suggestions, encouraging comments and contributed code, and to Matt Cooke, who has taken over as manager of the MockMaker project. Thanks to the reviewers from their helpful comments. Also thanks to Connextra for providing a useful testing ground for MockMaker.

REFERENCES

1. John Nolan, private communication, 2001.
2. Mackinnon T, Freeman S, Craig P. Endo Testing: Unit Testing with Mock Objects, *Extreme Programming eXamined*, Addison-Wesley, 2001.
3. Kent Beck. *eXtreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.
4. McCormick Europe Condiments Division http://www.mccormick.com and http://www.sybertooth.com/camp/
5. http://www.easymock.org/
6. XPUniverse, Raleigh, 2001.
7. http://groups.yahoo.com/group/extremeprogramming-seattle/files/extender.jar
8. http://www.abstrakt.de/en/mockcreator.html
9. http://mockmaker.sourceforge.net
10. http://mockry.sourceforge.net
11. http:// www.mockobjects.com