Endo-Testing: Unit Testing with Mock Objects

Tim Mackinnon, Steve Freeman, Philip Craig

(tim.mackinnon@pobox.com, steve@m3p.co.uk, philip@pobox.com)

ABSTRACT

Unit testing is a fundamental practice in Extreme Programming, but most non-trivial code is difficult to test in isolation. It is hard to avoid writing test suites that are complex, incomplete, and difficult to maintain and interpret. Using Mock Objects for unit testing improves both domain code and test suites. They allow unit tests to be written for everything, simplify test structure, and avoid polluting domain code with testing infrastructure.

Keywords: Extreme Programming, Unit Testing, Mock Objects, Stubs

INTRODUCTION

“Once,” said the Mock Turtle at last, with a deep sigh, “I was a real Turtle.”

(Alice In Wonderland, Lewis Carroll)

Unit testing is a fundamental practice in Extreme Programming [Beck 1999], but most non-trivial code is difficult to test in isolation. You need to make sure that you test one feature at a time, and you want to be notified as soon as any problem occurs. Normal unit testing is hard because you are trying to test the code from outside.

We propose a technique called *Mock Objects* in which we replace domain code with dummy implementations that both emulate real functionality and enforce assertions about the behaviour of our code. These Mock Objects are passed to the target domain code which they test from inside, hence the term *Endo-Testing*. Unlike conventional techniques, however, assertions are not left in production code, but gathered together in unit tests.

Writing Mock Objects is similar to writing code stubs with two interesting differences: we test at a finer level of granularity than is usual, and we use our tests and stubs to drive the development of our production code.

Our experience is that developing unit tests with Mock Objects leads to stronger tests and to better structure of both domain and test code. Unit tests written with Mock Objects have a regular format that gives the development team a common vocabulary. We believe that code should be written to make it easy to test, and have found that Mock Objects is a good technique to achieve this. We have also found that refactoring Mock Objects drives down the cost of writing stub code.

In this paper, we first describe how Mock Objects are used for unit testing. Then we describe the benefits and costs of Mock Objects when writing unit tests and code. Finally we describe a brief pattern for using Mock Objects.

UNIT TESTING WITH MOCK OBJECTS

An essential aspect of unit testing is to test one feature at time; you need to know exactly what you are testing and where any problems are. Test code should communicate its intent as simply and clearly as possible. This can be difficult if a test has to set up domain state or the domain code causes side effects. Worse, the domain code might not even expose the features to allow you to set the state necessary for a test.

For example, the authors have written tools to extend the development environment in IBM's VisualAge for Java, one of which generates template classes. This tool should not write a new template class if one already exists in the environment. A naïve unit test for this requirement would create a known class, attempt to generate a template class with the same name, and then check that the known class has not changed. In VisualAge this raises incidental issues such as whether the known class has been set up properly, ensuring that the user has the right

permissions, and cleaning up after the test should it fail – none of which are relevant to the test.

We can avoid these problems by providing our own implementation that simulates those parts of VisualAge that we need to run our test. We refer to these implementations as Mock Objects. Our Mock Objects can be initialised with state relevant to the test and can validate the inputs they have received from our unit test. In the example below, *JUnitCreatorModel* is an object that generates test classes in the VisualAge workspace, and *myMockPackage* and *myMockWorkspace* are Mock Object implementations of interfaces provided with VisualAge:

```
public void testCreationWithExistingClass() {
    myMockPackage.addContainedType(
        new MockType(EXISTING_CLASS_NAME));
    myMockWorkspace.addPackage(mockPackage);

    JUnitCreatorModel creatorModel =
       new JunitCreatorModel(myMockWorkspace, PACKAGE_NAME);

    try {
        creatorModel.createTestCase(EXISTING_CLASS_NAME);
        fail("Should generate an exception for existing type");
    } catch (ClassExistsException ex) {
        assertEquals(EXISTING_CLASS_NAME, ex.getClassName());
    }

    myMockWorkspace.verify();
}
```

There are two important points to note here. First, this test does not test VisualAge, it only tests one piece of code that we have written or, with test-driven programming, are about to write. The full behaviour is exercised during functional testing. Second, we are not trying to rewrite VisualAge, only to reproduce those responses that we need for a particular test. Most of the methods of a mock implementation do nothing or just store values in local collections. For example, the implementation of the class *MockPackage* might include:

```
public class MockPackage implements Package {
  private Vector myContainedTypes = new Vector();
  ...
  public void addContainedType(Type type) {
    myContainedTypes.add(type);
  }
}
```

A Mock Object is a substitute implementation to emulate or instrument other domain code. It should be simpler than the real code, not duplicate its implementation, and allow you to set up private state to aid in testing. The emphasis in mock implementations is on absolute simplicity, rather than completeness. For example, a mock collection class might always return the same results from an index method, regardless of the actual parameters. We have found that a warning sign of a Mock Object becoming too complex is that it starts calling other Mock Objects – which might mean that the unit test is not sufficiently local. When using Mock Objects, only the unit test and the target domain code are real.

NOT JUST STUBS

As a technique, Mock Objects is very close to Server Stubs [Binder 1999]. Binder's main concerns about using Server Stubs are: that stubs can be too hard to write, that the cost of developing and maintaining stubs can be too high, that dependencies between stubs can be cyclic, and that switching between stub and production code can be risky.

The most important differences between our use of stubs and Binder's is the extent to which we believe that the development of domain code can be driven from the tests and that individual classes can be tested in isolation. As one of our reviewers wrote, "Prior to XP, a tester suggesting [some refactoring of the code for testing] would have been laughed at." Furthermore, fine-grained unit tests combined with refactored Mock Object code, drive down the cost of writing stubs and help to ensure that domain components can be tested independently. Mock Objects that become too complex to manage suggest that their domain clients are candidates for refactoring, and we avoid chaining Mock Objects. Finally, our coding style of passing stub objects as parameters, rather than relinking the domain code, clarifies the scope of unit testing, and reduces the risk of mistakes during a build.

WHY USE MOCK OBJECTS?

Localising unit tests

Deferring Infrastructure Choices

An important aspect of Extreme Programming is not to commit to infrastructure before you have to. For example, we might wish to write functionality without committing to a particular database. Until a choice is made, we can write a set of mock classes that provide the minimum behaviour that we would expect from our database. This means that we can continue writing the tests for our application code without waiting for a working database. The mock code also gives us an initial definition of the functionality we will require from the database.

For example, we might have a method on a class *Employee* that updates name and details atomically:

```
public void updateNameAndDetails(Connection connection,
                    Statement nameStatement,
                    Statement detailsStatement) throws SQLException {
  try {
    nameStatement.executeUpdate(createUpdateNameSql());
    detailsStatement.executeUpdate(createUpdateDetailsSql());
    connection.commit();
  } catch (SQLException ex) {
    connection.rollback();
  }
}
```

With conventional testing, we would have to set up and connect to a real database, but with Mock Objects we can implement our test as:

```
public void testUpdateEmployee() throws SQLException {
  MockStatement nameStatement = new MockStatement();
  MockStatement detailsStatement = new MockStatement();
  MockConnection connection = new MockConnection();

  nameStatement.setExpectedUpdate(NAME_SQL);
  detailsStatement.setExpectedUpdate(DETAILS_SQL);
  connection.setExpectedCommitCalls(1);

  myEmployee.updateNameAndDetails(connection, nameStatement,
                                  detailsStatement);

  connection.verify();
  nameStatement.verify();
  detailsStatement.verify();
}
```

This technique is very easy to implement in modern development environments, especially given industry standard interfaces such as JDBC.

Coping with scale

Unit tests, as distinct from functional tests, should exercise a single piece of functionality. A unit test that depends on complex system state can be difficult to set up, especially as the rest of the system develops. Mock Objects avoid such problems by providing a lightweight emulation of the required system state. Furthermore, the setup of complex state is localised to one Mock Object instead of scattered throughout many unit tests.

One of the authors worked on a project tool that released code from VisualAge to another source control system. As the tool grew, it became increasingly hard to unit test because the cost of resetting the environment rose dramatically. The project tool was later refactored to use mock implementations of both VisualAge and the source control system. The result was both easier to test and better structured.

No stone unturned

Some unit tests need to test conditions that are very difficult to reproduce. For example, to test server failures we can write a Mock Object that implements the local proxy for the server. Each unit test can then configure the proxy to fail with an expected problem and the developers can write client code to make the test pass. An example of this is:

```
public void testFileSystemFailure() {
    myMockServer.setFailure(FILE_SYSTEM_FAILURE);

    myApplication.connectTo(myMockServer);
    try {
        myApplication.doSomething();
        fail("Application server should have failed");
    } catch (ServerFailedException e) {
        assert(true);
    }

    myMockServer.verify();
}
```

With this approach, the mock server runs locally and fails in a controlled manner. The test has no dependencies on components outside the development system and is insulated from other

possible real world failures. This style of test is repeated for other types of failure, and the entire test suite documents the possible server failures that our client code can handle.

In the case of an expensive widget, we define similar unit tests. We can configure the mock widget with the desired state and check that it has been used correctly. For example, a unit test that checks that the widget is polled exactly once when a registration key is sent would be:

```
public void testPollCount() {
    myMockWidget.setResponseCode(DEVICE_READY);
    myMockWidget.setExpectedPollCount(1);

    myApplication.sendRegistrationKey(myMockWidget);

    myMockWidget.verify();
}
```

The mock widget lets us run tests on development machines with no actual widget installed. We can also instrument the mock widget to verify that it was called correctly, which might not even be possible with the real widget.

Better tests

Failures fail fast

Domain objects often fail some time after an error occurs, which is one reason that debugging can be so difficult. With tests that query the state of a domain object, all the assertions are made together after the domain code has executed. This makes it difficult to isolate the exact point at which a failure occurred. One of the authors experienced such problems during the development of a financial pricing library. The unit tests compared sets of results after each calculation had finished. Each failure required considerable tracing to isolate its cause, and it was difficult to test for intermediate values without breaking encapsulation.

On the other hand, a mock implementation can test assertions each time it interacts with domain code and so is more likely to fail at the right time and generate a useful message. This makes it easy to trace the specific cause of the failure, especially as the failure message can also describe the difference between the expected and actual values.

For example, in the widget code above, the mock widget knows that it should only be polled once and can fail as soon as a second poll occurs:

```
class MockWidget implements Widget {
   ...
   public ResponseCode getDeviceStatus() {
     myPollCount++;
     if (myPollCount > myExpectedPollCount) {
       fail("Polled too many times", myExpectedPollCount,
               myPollCount);
     }
     return myResponseCode;
   }
}
```

Refactored assertions

When testing without Mock Objects, each unit test tends to have its own set of assertions about the domain code. These may be refactored into shared methods within a unit test, but the developer has to remember to apply them to new tests. On the other hand, these assertions are built into Mock Objects and so are applied by default whenever the object is used. As the suite of unit tests grows, a Mock Object will be used throughout the system and its assertions applied to new code. Similarly, as the developers discover new assertions that need to be made, these can be added once in the Mock Object where they will automatically apply to all previously existing tests.

During development, the authors have come across situations where assertions in their Mock Objects have failed unexpectedly. Usually this is a timely warning about a constraint that the programmers have forgotten, but sometimes this is because the failing constraints are not always relevant. These cases suggest candidates for refactoring of either the domain code or Mock Objects, and help to push the developers towards a better understanding of the system.

Effects on coding style

We have found that developing with Mock Objects has had beneficial effects on the coding style of our teams.

First, in languages with controlled scope such as Java, detailed unit testing can be difficult without either breaking the scope by giving test code access to class or package features, or by moving the unit tests to domain packages. Stroustroup introduced the friend function into C++ to solve just this problem [Stroustroup 1992]. Whatever the solution, such code contradicts the intention of the design. Developing with Mock Objects reduces the need to expose the structure of domain code. A test knows more about the behaviour and less about the structure of tested code.

Second, singleton objects are increasingly recognised as a doubtful practice [C2]. Unit testing in the presence of singletons can be difficult because of the state that must be managed between tests. Furthermore, the singleton objects might not have methods to allow a unit test to set up the state it needs or query the results afterwards. Developing with Mock Objects encourages a coding style where objects are passed into the code that needs them. This makes substitution possible and reduces the risk of unexpected side-effects.

Thirdly, developing with Mock Objects teases out different aspects of functionality into smaller, more specialised classes which are easier to understand and modify. In practice, this means pushing behaviour towards Visitor-like objects [Gamma 1994] that are passed around; we call these *Smart Handlers*. For example, rather than having code that queries attributes from an object and prints each one to a writer, a first step would be to pass a writer to the object which then prints out its attributes. This preserves the encapsulation of the object.

Thus, the code changes from:

```
public void printPersonReport(Person person, PrintWriter writer) {
    writer.println(person.getName());
    writer.println(person.getAge());
    writer.println(person.getTelephone());
}
```

to:

```
public void printPersonReport(Person person, PrintWriter writer) {
    person.printDetails(writer);
}
```

```
public class Person {
    public void printDetails(PrintWriter writer) {
        writer.println(myName);
        writer.println(myAge);
        writer.println(myTelephone);
    }
    ...
}
```

Which can be tested with:

```
void testPersonHandler() {
    myMockPrintWriter.setExpectedOutputPattern(
        ".*" + NAME + ".*" + AGE + ".*" + TELEPHONE + ".*");

    myPerson.setName(NAME);
    myPerson.setAge(AGE);
    myPerson.setTelephone(TELEPHONE);

    myPerson.printDetails(myMockPrintWriter);

    myMockPrintWriter.verify();
}
```

This test is verifying two things at once: that *Person* is managing its details correctly *and* that these details are being rendered correctly. As this code becomes more complex it becomes difficult to test cleanly because the generic *println* method used in *printDetails* loses information about our understanding of the domain.

Instead, we can write a handler object to reify this dialogue between a writer and a *Person*. *Person* would then have a method to pass its internal contents to a handler:

```
public void handleDetails(PersonHandler handler) {
    handler.name(myName);
    handler.age(myAge);
    handler.telephone(myTelephone);
}
```

This separates the input and output aspects of rendering a *Person* on a writer, and we can now test each aspect independently. The unit test for the handler inputs would then be:

```
void testPersonHandling() {
    myMockHandler.setExpectedName(NAME);
    myMockHandler.setExpectedAge(AGE);
    myMockHandler.setExpectedTelephone(TELEPHONE);

    myPerson.handleDetails(myMockHandler);

    myMockHandler.verify();
}
```

followed by a separate unit test to check that the domain code for *PersonPrintHandler*, a printing implementation of *PersonHandler*, outputs itself correctly:

```
void testPersonPrintHandler() {
    myMockPrintWriter.setExpectedOutputPattern(
        ".*" + NAME + ".*" + AGE + ".*" + TELEPHONE + ".*");

    myPrintHandler.name(NAME);
    myPrintHandler.age(AGE);
    myPrintHandler.telephone(TELEPHONE);

    myPrintHandler.writeTo(myMockPrintWriter);

    myMockPrintWriter.verify();
}
```

Factoring out these two aspects makes it much easier to test and implement new features, such as renderings to XML or encrypted mail.

These three effects mean that code developed with Mock Objects tends to conform to the Law of Demeter [Lieberherr 1989], as an emergent property. The unit tests push us towards writing domain code that refers only to local objects and parameters, without an explicit policy to do so.

Interface discovery

When writing code that depends on other related objects, we have found that developing with Mock Objects is a good technique for discovering the interface to those other objects. For each new feature, we write a unit test that uses Mock Objects to simulate the behaviour that our target object needs from its environment; each Mock Object is a hypothesis of what the real code will eventually do. As the cluster of a domain object and its Mock Objects stabilises, we can extract their interactions to define new interfaces that the system must implement. An interface will consist of those methods of a Mock Object that are not involved with setting or checking expectations. In statically typed languages, one then replaces the references to the Mock Object in the domain code with the new interface.

For example, the *Person* class shown above would initially use a *MockPersonHandler* to get its unit tests running:

```
public class Person {
    public void handleDetails(MockPersonHandler handler) {
        handler.name(myName);
        handler.age(myAge);
        handler.telephone(myTelephone);
    }
    ...
}
```

When the tests all run, we can extract the following interface:

```
public interface PersonHandler {
    void name(String name);
    void age(int age);
    void telephone(String telephone);
    void writeTo(PrintWriter writer);
}
```

We would then return to the Person class and adjust any method signatures to use the new interface:

```
public void handleDetails(PersonHandler handler) { ... }
```

This approach ensures that the interface will be the minimum that the domain code needs, following the Extreme Programming principle of not adding features beyond our current understanding.

LIMITATIONS OF MOCK OBJECTS

As with any unit testing, there is always a risk that a Mock Object might contain errors, for example returning values in degrees rather than radians. Similarly, unit testing will not catch failures that arise from interactions between components. For example, the individual calculations for a complex mathematical formula might be within valid tolerances, and so pass their unit tests, but the cumulative errors might be unacceptable. This is why functional tests are still necessary, even with good unit tests. Extreme Programming reduces, but does not eliminate, such risks with practices such as Pair Programming and Continuous Integration. Mock Objects reduce this risk further by the simplicity of their implementations.

In some cases it can be hard to create Mock Objects to represent types in a complex external library. The most difficult aspect is usually the discovery of values and structures for parameters that are passed into the domain code. In an event-based system, the object that

represents an event might be the root of a graph of objects, all of which need mocking up for the domain code to work. This process can be costly and sometimes must be weighed against the benefit of having the unit tests. However, when only a small part of a library needs to be stubbed out, Mock Objects is a useful technique for doing so.

One important point that we have learned from trying to retrofit Mock Objects is that, in statically typed languages, libraries should define their APIs in terms of interfaces rather than classes so that clients of the library can use such techniques. We have used Mock Objects in the context of several application server environments and have sometimes found that the use of Java visibility modifiers, without corresponding public interfaces, makes unit testing more difficult to set up, although not impossible. For example, we were able to extend VisualAge because the tool API was written in terms of interfaces, whereas the Vector class in Java 1.1.x has many final methods but no interface, making it impossible to substitute.

A PATTERN FOR UNIT TESTING

As the authors worked with Mock Objects, they found that their unit tests developed a common format:

- Create instances of Mock Objects
- Set state in the Mock Objects
- Set expectations in the Mock Objects
- Invoke domain code with Mock Objects as parameters
- Verify consistency in the Mock Objects

With this style, the test makes clear what the domain code is expecting from its environment, in effect documenting its preconditions, postconditions, and intended use. All these aspects are defined in executable test code, next to the domain code to which they refer. We sometimes

find that arguing over which objects to verify gives us better insight into a test and, hence, the domain.

In our experience, this style makes it easy for new readers to understand the unit tests as it reduces the amount of context they have to remember. We have also found that it is useful for demonstrating to new programmers how to write effective unit tests. For example, we have been using pair programming as an interview technique and have found that candidates have been able to make valid contributions to production code within an afternoon.

We use this pattern so often that we have refactored common assertions into a set of Expectation classes [Mackinnon 2000], which makes it quick to write many types of Mock Object. Currently we have refactored this code into the classes, *ExpectationCounter*, *ExpectationList* and *ExpectationSet*. For example, the *ExpectationList* class has the following interface:

```
public class ExpectationList implements Verifiable {
    public ExpectationList(String failureMessage);
    public void addExpectedItem(Object expectedItem);
    public void addActualItem(Object actualItem);
    public void verify() throws AssertionFailedException;
}
```

where the *verify* method asserts that matching actual and expected items were inserted in the same order during the test, and where they don't it prints out an error message that indicates where the differences occur. A Mock Object that cares about sequence would either extend or delegate to an *ExpectationList*.

CONCLUSIONS

We and our colleagues have used Mock Objects on several projects, such as web servers, desktop applications and IDE extensions. We have mocked up standard libraries and parts of several application servers, and developed a significant project using Mock Objects throughout. Our experience is that Mock Objects is an invaluable technique for developing unit tests. It encourages better-structured tests and reduces the cost of writing stub code, with

a common format for unit tests that is easy to learn and understand. It also simplifies debugging by providing tests that detect the exact point of failure at the time a problem occurs. Sometimes, using Mock Objects is the only way to unit test domain code that depends on state that is difficult or impossible to reproduce. Even more importantly, testing with Mock Objects improves domain code by preserving encapsulation, reducing global dependencies, and clarifying the interactions between classes. We have been pleased to notice that colleagues who have also adopted this approach have observed the same qualities in their tests and domain code.

REFERENCES

[Beck 1999] Kent Beck. *Extreme programming explained: embrace change.* Reading Mass.: Addison-Wesley, 1999.

[Binder 1999] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Reading Mass.: Addison-Wesley, 1999.

[C2] Various Authors, *SingletonsAreEvil* [WWW] available from: http://c2.com/cgi/wiki?SingletonsAreEvil (Accessed: April 7, 1999)

[Gamma 1994] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley, 1994.

[Lieberherr 1989] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE Software* 6(5):38-49, September 1989.

[Mackinnon 2000] Tim Mackinnon, *JunitCreator* [WWW] available from: http://www.xpdeveloper.com/cgi-bin/wiki.cgi?JUnitCreator (Accessed: February 17, 2000)

[Stroustroup 1992] Bjarne Stroustroup. *The design and evolution of C++*. Reading Mass.: Addison-Wesley, 1992.

ACKNOWLEDGEMENTS

We would like to thank the reviewers and the following colleagues for their contributions to this paper: Tom Ayerst, Oliver Bye, Richard Karcich, Matthew Cooke, Sven Howarth, Tung Mac, Peter Marks, Ivan Moore, John Nolan, Keith Ray, Paul Simmons, J. D. Weatherspoon.

NOTES

We will be posting example code at http://www.xpdeveloper.com.

TRADEMARKS