David Keen

Object Oriented Programming

Mock Objects and test driven design (TDD)

Test every work of intellect or faith
And everything that your own hands have wrought
— William Butler Yeats

Mock Objects are a relatively new tool in the object oriented programmer's toolkit. Developed by Tim Mackinnon, Steve Freeman, and Philip Craig and first presented at the conference "eXtreme Programming and Flexible Processes in Software Engineering – XP2000", Mock Objects are a powerful aid to Unit Testing and Test Driven Development.

Many programmers are familiar with unit testing. Basically, this involves writing tests for some or all of the classes and methods in a program. There are a number of unit testing frameworks such as the xUnit family, first designed by Kent Beck for SmallTalk but now extended to more than twenty-five different frameworks for languages such as C++, Perl, Java and PHP. The goal of unit testing is to isolate each method or function and, through the use of an automated testing framework, write tests than can be repeatedly run to verify the correctness of the code. Having complete coverage of your code with unit tests gives you the confidence that any change you make to existing code does not introduce new errors in other parts of your program. Unit testing is an important part of the eXtreme Programming methodology as well as other Agile methodologies such as Scrum.
But Test Driven Development is more than just unit testing. Probably the most significant part of Test Driven Development (TDD), also known as Test First Development, is that the tests are written *first*. Before a single line of a method has been written, the test for that method should exist. "No code without tests" is an eXtreme Programming mantra. Apart from completely trivial code, tests are written for nearly all production code; code does not exist unless a test has been written for it.

A basic outline of the TDD use of unit testing is as follows. Say, we want to implement a software class that represents a sales order. This class may need to have a method that calculates the amount of tax for the order. If we are practising TDD, we will start by writing a unit test for this method before we have even written the method signature itself. We will write the test to check for an expected value and then run the test which will, of course, fail. But that is ok. We use the compiler to alert us to the problem of the missing method. We then write *just enough* code to make the test pass in the easiest and simplest way. If that means just returning a constant then that is fine at this stage.

There are actually two distinct phases of coding using Test Driven Development [Fowler99]. The first stage involves writing just enough code for the test to pass and not worrying too much about the elegance of the code itself. The second stage is just as important as the first – refactoring. Refactoring is the process of rewriting existing code without changing its external behaviour. That is, we change not what the code *does* but *how it does it*.
Why do we refactor code? We refactor to: remove duplication, make maintenance easier and improve the clarity of our code [Larman05]. Taking the above example of the sales order class, the code we wrote for calculating the tax may pass the test by returning a constant value, but it is in dire need of refactoring. In this case we are actually refactoring to remove duplication. The duplication in this method is a little more subtle than just duplicate lines of code. In returning a single constant value we are actually duplicating a value we can calculate from the existing information in the sales order class. So we refactor this code to calculate the tax using the fields from the sales order class. Of course we would also have run into problems as soon as we wrote another test condition that tests the code using different values.

This is the TDD method; write a little test, write a little code, write another little test, write some more code and so on. Each of the tiny code writing iterations consists of the coding and refactoring stages.

Larman [Larman05] identifies a number of benefits from following a test driven development methodology:
Tests actually get written – many programmers will write some code with all the good intentions of writing unit tests for their code later. But after the code has been written and superficially appears to work, tests often get forgotten. Writing the tests before the actual code forces programmers to write thorough tests for all production code.

It can be more satisfying for the programmer – writing code to pass tests can give the programmer a tangible, quantifiable, achievable goal. As unit tests should be as focused as possible, they should each be achievable in a day's coding. At the end of the day, the programmer has a visible confirmation of what they have achieved. Writing tests for code you have already worked on is for some reason not nearly as satisfying and can seem a chore. After all, the code is written and it seems to work, so what is the point of writing a test?

Clarification of the object's interface and methods – a great advantage of TDD is that it forces the programmer to not only think about the interface of the object he is designing, but also to think of it from the most important point of view – that of the client object using it.

Provable, repeatable validation of working code – the correctness of the production code can be validated repeatedly in a quantifiable way rather than relying on a vague assumption, belief or hope that the code works as it should.

Gives the confidence to make changes to the code – refactoring, an important part of Test Driven Development is a very dangerous thing to do without the support of a suite of unit tests. After refactoring, the test suite must be rerun to ensure the changes in the code have not introduced any unexpected errors in other parts of the code. Without an automated testing framework it is very difficult to be sure your refactorings have not introduced new bugs.

Most non-trivial code is hard to test in isolation [MFC00]; only the simplest unit tests will not reference any other objects. Rather, the objects we are testing will often need to interact with other objects in the domain and some of these other objects may have no real bearing on what we are trying to test. Unit tests should be as focussed and simple as possible, testing one thing only. Setting up and initialising a number of domain objects not relevant to the test is a waste of time, effort and resources. Not only this, but it also tightly couples your test code to other objects in the system. If someone changes the interface to one of the objects you are using in your unit test then this will cause errors to ripple out to any test that makes use of that object. When Mackinnon, Freeman, and Craig introduced mock objects in their paper *Endo-Testing: Unit Testing with Mock Objects* [MFC00], they suggested a technique in which the domain code is replaced with dummy code that emulates the real object.

Massol's [MH03] definition of a mock object is:

> *"An object created to stand in for an object that your code will be collaborating with. Your code can call methods on the mock object, which will deliver results as set up by your tests."*

The idea of creating dummy objects with method stubs was not a new idea. Programmers had previously created dummy objects with dummy methods when writing and testing code that needed to use objects that may not have been written yet. But these stubs were really only place holders, at most returning fixed values. They were mostly a convenience that allowed tests and code to be written against objects that hadn't been implemented yet.

Mock objects build on this idea of a dummy object with method stubs by giving the mock objects the ability to verify that they have been used correctly by the primary object being tested. It is from this idea that the name of Mackinnon, Freeman and Craig's paper, "Endo-Testing" was derived; while the unit test of the primary object tests things from the outside, the mock object can also verify from the inside that it has been used correctly.

Mock objects, or Mocks as they are known, may test that their methods have been called the correct number of times, with the correct parameters and the correct conditions and that the data returned has been handled the correct way. Mocks can verify that everything that should be expected to happen to the object did. Sometimes the mock could throw an exception immediately, for example for too many method calls or wrong arguments to a method. Other times, such as the case of when a method has not been called enough times, it would have to wait until the object was verified at the end. If the mock throws an exception as soon as it has detected something wrong has happened this reports the failure as close to where it occurred as possible. We don't have to wait until the test has run before we are informed which makes it easier to locate errors and faster to find and fix the problem.

Mock objects do not only help us to write self-contained, focused unit tests. They can also be a powerful tool in Test Driven Development. In the same way that writing the unit test for a piece of code before the actual production code itself is written can help you create lean functional code, writing a mock for an object that has not yet been implemented can help you develop effective domain objects. In both cases the programmer is forced to think about the interface his code presents. The eXtreme Programming methodology emphasises the economical use of programmer time by only implementing the minimum amount of features required. By creating a mock object before the production code it is a simple matter to extract the interface from the mock when implementing the domain object.

Mock objects let us test for conditions that may otherwise be very difficult or impossible to set up. A common use of mocks is for testing of IO errors. Without mocks it can be very hard to test for IO failures but with the use of mock objects we can create a mock that will simulate any error we wish. This highlights another great advantage of using mocks; everything in the test is under our strict control. Good unit tests should test one thing and be isolated from other code as much as possible. If our test only interacts with mock objects rather than real objects then we have effectively decoupled our test from the rest of the code. This decoupling concept can also be applied to hardware in the system. We may be developing software to interact with a specific piece of hardware or another complex system. If we mock the interface to this hardware or system it frees us from the requirement of having the actual equipment available for testing. It also allows us to simulate hardware errors.

While network and IO errors may be hard to simulate without mock objects, there are other conditions that may be nearly impossible. For example, it would be very hard to test a class that uses another object that returns some kind of random number unless we could control the other class as a mock object.

Speed of testing is also an advantage of using mocks. It is usual to have a class in a system for persistent storage access and mock objects can be used to simulate this database connection. Without the use of mocks, a database may have to be dropped, created, populated with appropriate data and queried for each test. For an extensive test suite this could take considerable time, not to mention the requirement that every developer's machine must have a working database installation. However, if we mock the data access interface we can simulate an entire database through the use of mock objects. Not only does this make it easy to have the mock database return specific values but it also removes the considerable overhead of the database operations.
But mock objects are not necessarily a magic bullet for testing and Mackinnon, Freeman and Craig [MFC00] identify some possible problems that may arise when using mock objects.
As with any other code, there is always the possibility of errors in the test code. This is a good reason to keep your test code as simple and focused as possible to minimise the risk that coding errors in the tests return erroneous results.

Problems may also arise when using a mock object in more than a single test. There can be precision errors that may not be noticeable in a single object but once their results are used in a number of calculations unacceptable rounding errors can creep in. Also, more generally, if a mock object is used in several different tests then any errors in the mock will ripple across all the tests it is used in. However, a buggy mock object is more easily pinpointed and fixed than a buggy genuine object.

Also, while using mocks to test the data access layer in a system has a number of speed and testing advantages, it has the disadvantage that while the business logic of the objects can be easily tested this way, there is no easy way of testing the actual raw SQL queries themselves in an application without running them against a real relational database with real representative data. This is a real problem in a database intensive system and the company I work for has a practice of using mocks where possible for all layers above the data access layer but to fall back to testing real SQL statements against real data for the lower layers. It may also be the case that some objects are impossible to create mocks for. This may happen if you are trying to create mocks for an external library that contains many classes with final methods and no interfaces.

There is also the simple fact that it can be quite some effort to mock a large number of classes and this must be weighed against the perceived benefits from doing so. As mock objects get more complex and numerous they get harder to write and maintain. Luckily there are a number of frameworks that are available to make things easier for us, and save us from having to write all the mock object code from scratch. The MockObjects framework [MFC03], developed by the same people who introduced the concept of Mock Objects is a generic framework that assists in the creation of mock objects.

Astels [Astels03] lists the common usage pattern of mock objects:

1. Create the mock objects
2. Set the mocks' state and expectations
3. Execute your test code using mocks
4. Have the mocks verify the expectations that were set

The MockObjects framework assists in this process by providing a number of classes that take some of the repetitiveness out of constructing the mocks. At the top of the com.mockobjects package class hierarchy is the `Verifiable` interface. This provides the `verify()` method that should be overridden in your mock object implementations to check that the mock was used in the expected way.

Extending this interface is the Expectation interface. From the source JavaDoc [MFC03a]: "An Expectation is an object that we set up at the beginning of a unit test to expect certain things to happen to it. If it is possible to tell, the Expectation will fail as soon as an incorrect value has been set. Call verify() at the end of a unit test to check for missing or incomplete values. If no expectations have been set on the object, then no checking will be done and verify() will do nothing."

Two of the most important classes that implement the Expectation interface are: ExpectationValue and ExpectationCounter. ExpectationValue objects are used to hold expected and actual values. When the mock object's verify() method is called the actual values are compared to the expected values. This class is often used to verify the correctness of arguments passed to methods.
The ExpectationCounter class is used to set expected and actual counts of something, usually the number of times a method is called. There is also an ExpectationDoubleValue class for testing doubles using a set tolerance for equality and an ExpectationSegment class for testing substring values.

An example mock object that uses the MockObject framework is given below (Listing 1). It is taken from the examples in the MockObjects source code distribution[MFC03]. You can see the basic steps required when creating a mock object.
The mock class extends the abstract MockObject class and implements the interface of the domain object we are mocking; in this case IntCalculator. A field is used to hold the value to be returned by the mock and ExpectationValues are created for the method arguments. A setExpectedCalculation() method is written to set the expected values for the ExpectationValue objects and a setUpResult() method sets the result to be returned from the calculation.
The calcucate() method from the domain object is overridden and sets the actual values passed into the object and returns the preset result.
The verify() method is where the mock object verifies each of the ExpectationValues and reports whether they are as expected or not.

```
package com.mockobjects.examples.calcserver;

import com.mockobjects.ExpectationValue;
import com.mockobjects.MockObject;

public class MockCalculator extends MockObject implements IntCalculator {
   private int myResult;
   private ExpectationValue myValue1 = new ExpectationValue("value1");
   private ExpectationValue myValue2 = new ExpectationValue("value2");
   private ExpectationValue myOperation = new ExpectationValue("operation");
   private String myBadOperation;


   /**
    * MockCalculator constructor comment.
    */
   public MockCalculator() {
      super();
   }


   public int calculate(int value1, int value2, String operation) throws CalculatorException {
      myValue1.setActual(value1);
      myValue2.setActual(value2);
      myOperation.setActual(operation);
      return myResult;
   }


   public void setExpectedCalculation(int value1, int value2, String operation) {
      myValue1.setExpected(value1);
      myValue2.setExpected(value2);
      myOperation.setExpected(operation);
   }


   public void setupResult(int result) {
      myResult = result;
   }


   public void setupThrowBadOperation(String opName) {
      myBadOperation = opName;
   }


   public void verify() {
      myValue1.verify();
      myValue2.verify();
      myOperation.verify();
   }
}
```

Listing 1: MockCalculator class [MFC03]

This simple example only used ExpectationValues to verify the arguments passed to the object. If we wanted to test for the number of times the calculate() method was called we could have used the ExpectationCounter class.

While the MockObjects framework has simplified much of the effort in writing mock objects, it can still be some work creating mocks for complex classes or a large number of objects. The MockMaker tool [MFC02], developed in late 2000 by the creators of the MockObjects framework, can generate complete mock objects from the source code to your domain objects. Listing 2 shows the generated code from running the MockMaker Eclipse plugin on the IntCalculator interface used in the previous example. The generated code uses ExpectationLists and ExpectationCounters to count the number of method calls.

```
import mockmaker.ReturnValues;
import mockmaker.VoidReturnValues;
import mockmaker.ExceptionalReturnValue;
import com.mockobjects.*;
import IntCalculator;
public class MockIntCalculator implements IntCalculator{
  private ExpectationCounter myCalculateCalls = new ExpectationCounter("IntCalculator CalculateCalls");
  private ReturnValues myActualCalculateReturnValues = new ReturnValues(false);
  private ExpectationList myCalculateParameter0Values = new ExpectationList("IntCalculator int");
  private ExpectationList myCalculateParameter1Values = new ExpectationList("IntCalculator int");
  private ExpectationList myCalculateParameter2Values = new ExpectationList("IntCalculator java.lang.String");
  public void setExpectedCalculateCalls(int calls){
    myCalculateCalls.setExpected(calls);
  }
  public void addExpectedCalculateValues(int arg0, int arg1, String arg2){
    myCalculateParameter0Values.addExpected(new Integer(arg0));
    myCalculateParameter1Values.addExpected(new Integer(arg1));
    myCalculateParameter2Values.addExpected(arg2);
  }
  public int calculate(int arg0, int arg1, String arg2) throws CalculatorException{
    myCalculateCalls.inc();
    myCalculateParameter0Values.addActual(new Integer(arg0));
    myCalculateParameter1Values.addActual(new Integer(arg1));
    myCalculateParameter2Values.addActual(arg2);
    Object nextReturnValue = myActualCalculateReturnValues.getNext();
    if (nextReturnValue instanceof ExceptionalReturnValue &&
((ExceptionalReturnValue)nextReturnValue).getException() instanceof CalculatorException)
        throw (CalculatorException)((ExceptionalReturnValue)nextReturnValue).getException();
    if (nextReturnValue instanceof ExceptionalReturnValue &&
((ExceptionalReturnValue)nextReturnValue).getException() instanceof RuntimeException)
        throw (RuntimeException)((ExceptionalReturnValue)nextReturnValue).getException();
    return ((Integer) nextReturnValue).intValue();
  }
  public void setupExceptionCalculate(Throwable arg){
    myActualCalculateReturnValues.add(new ExceptionalReturnValue(arg));
  }
  public void setupCalculate(int arg){
    myActualCalculateReturnValues.add(new Integer(arg));
  }
  public void verify(){
    myCalculateCalls.verify();
    myCalculateParameter0Values.verify();
    myCalculateParameter1Values.verify();
    myCalculateParameter2Values.verify();
  }
}
```

Listing 2: MockMaker generated class for IntCalculator interface

Both MockMaker and MockObjects suffer from the problem that if your objects' interfaces change then you must keep your mocks updated as well. MockMaker was developed using Java 1.1.7 so at the time there was not much that could be done about this, however a tool called EasyMock [Freese], written by Tammo Freese, makes use of Java's Dynamic Proxies to generate the mock objects on the fly. This solves the problem of hand-cranking all your mock objects as you are forced to do with the bare MockObjects framework and it also solves the problem that MockMaker has of having to maintain your mock object code after it is generated. Method completion is available in your IDE and because EasyMock generates the mocks dynamically, any refactoring of your domain objects is automatically applied to your mocks as well. Another great advantage of EasyMock is that the generation of the mocks is done in the unit test itself. This tightly binds the mock object to the test it is used in and focuses the code on the unit test. The core EasyMock tool can only create mocks for interfaces however there is an extension that allows mocks to be generated for classes as well.

There are three phases in using Easymock as desribed on the EasyMock website [Freese]:

1. Create a Mock Object for the interface we would like to simulate,
2. Record the expected behaviour, and
3. Switch the Mock Object to replay state.

These are shown in the examples below.

```
protected void setUp() {
      mock = createMock(Collaborator.class); // 1
      classUnderTest = new ClassUnderTest();
      classUnderTest.addListener(mock);
   }

   public void testRemoveNonExistingDocument() {
      // 2 (we do not expect anything)
      replay(mock); // 3
      classUnderTest.removeDocument("Does not exist");
   }
```

Listing 3: Unit test using EasyMock [Freese]

In the first example (Listing 3), a mock is created using the static method createMock(), passing the class file as an argument. This mock is now in "record" mode. In this case we do not expect any calls, so all we need to do is switch the mock into "verify" mode by calling the replay() method. If any methods are called on this mock it will throw an AssertionError.
In the next example, we add some expected behaviour and verify that the mock was used correctly with the verify() method.

```
public void testAddDocument() {
      mock.documentAdded("New Document"); // 2
      replay(mock); // 3
      classUnderTest.addDocument("New Document", new byte[0]);
      verify(mock);
   }
```

Listing 4: Unit test using EasyMock with expected behaviour [Freese]

This example only expects the method to be called once. You could add multiple method calls by simply repeating the methods in record mode, but EasyMock provides a shortcut. Simply specify the expected method once and then use the times() method on the object returned by expectLastCall(). So if we expected the documentAdded() method to be called twenty times we could use the following code:

```
mock.documentAdded("New Document");
expectLastCall().times(20);
```

Listing 5: Setting the number of times a method is expected

From these small examples we can see that using EasyMock saves us the trouble of coding our mock objects by hand. There will obviously be a performance hit when running tests that use EasyMock as opposed to a non-dynamically generated mock objects framework. Because the mocks are specified in each unit test they must be generated for each test as each test will want the mock object to expect different things. Another downside is that mocks generated by EasyMock will be less flexible than those created from scratch or using the MockObjects framework, but this is an unavoidable trade-off when using automated tools. If you need your mocks to do something special, then you must create them using MockMaker or from scratch. There is an interesting thread [BFT06] on the EasyMock Yahoo Groups list about the possibility of using EasyMock to generate mocks for final classes. Because EasyMock uses Java's reflection capabilities to generate the mocks from .class files it may be possible to remove the final modifier from the byte code of external libraries thereby allowing you to create mocks for these objects. If this were implemented as an EasyMock plugin it would give EasyMock a powerful advantage over other mock object frameworks when dealing with final classes from external libraries.

Mock objects are a neat way of dealing with an unfortunate reality of programming; that the objects we write will usually need to interact with other objects in the system. Using mocks can not only keep our tests focussed on the things they should be testing, but they can also be a powerful aid to developing strong, cohesive classes that have been designed to interfaces. The few difficulties there are in using mocks are far outweighed by their usefulness in Test Driven Development.

Bibliography

[Astels03]	Astels D. 2003. *Test-driven development: A practical guide*. Prentice Hall PTR
[BFT06]	Burger M, Freese T, Tremblay H. 2006. *Mock a final class using EasyMock 2.0 Class Extension*. Thread on EasyMock group – Yahoo! Groups. Available at http://groups.yahoo.com/group/easymock/message/536.
[Fowler04]	Fowler M. 2004. *Mocks Aren't Stubs*. Available at http://martinfowler.com/articles/mocksArentStubs.html.
[Fowler99]	Fowler M. 1999. *Refactoring: Improving the design of existing code*. Addison Wesley Longman.
[Freese]	Freese T. The EasyMock website is located at http://www.easymock.org.
[Hanson]	Hanson K. *Using Mock Objects in Java*. Tutorial on Java Boutique available at http://javaboutique.internet.com/tutorials/mock_objects/.
[Larman05]	Larman, C. 2005. *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*, 3rd edition. Prentice Hall PTR.
[MFC00]	Mackinnon T., Freeman S., Craig P. 2000. *Endo-Testing: Unit Testing with Mock Objects*. Available at http://www.connextra.com/aboutUs/mockobjects.pdf.
[MFC02]	MockMaker is available at http://mockmaker.sourceforge.net/
[MFC03]	Mackinnon T., Freeman S., Craig P. 2003. MockObjects 0.09 release source code is available at http://sourceforge.net/projects/mockobjects.
[MFC03a]	Mackinnon T., Freeman S., Craig P. 2003. MockObjects 0.09 JavaDoc is available at http://mockobjects.sourceforge.net/javadoc/1.4/
[MH03]	Massol V, Husted T. 2003. *JUnit in action*. Manning Publications Co.