Mocks Aren't Stubs

Martin Fowler

The term 'Mock Objects' has become a popular one to describe special case objects that mimic real objects for testing. Most language environments now have frameworks that make it easy to create mock objects. What's often not realized, however, is that mock objects are but one form of special case test object, one that enables a different style of testing. In this article I'll explain how mock objects work, how they encourage testing based on behavior verification, and how the community around them uses them to develop a different style of testing.

Last significant update: 02 Jan 07

Contents

I first came across the term "mock object" a few years ago in the XP community. Since then I've run into mock objects more and more. Partly this is because many of the leading developers of mock objects have been colleagues of mine at ThoughtWorks at various times. Partly it's because I see them more and more in the XP-influenced testing literature.

But as often as not I see mock objects described poorly. In particular I see them often confused with stubs - a common helper to testing environments. I understand this confusion - I saw them as similar for a while too, but conversations with the mock developers have steadily allowed a little mock understanding to penetrate my tortoiseshell cranium.

This difference is actually two separate differences. On the one hand there is a difference in how test results are verified: a distinction between state verification and behavior verification. On the other hand is a whole different philosophy to the way testing and design play together, which I term here as the classical and mockist styles of Test Driven Development.

(In the earlier version of this essay I had realized there was a difference, but combined the two differences together. Since then my understanding has improved, and as a result it's time to update this essay. If you haven't read the previous essay you can ignore my growing pains, I've written this essay as if the old version doesn't exist. But if you are familiar with the old version you may find it helpful to note that I've broken the old dichotomy of state based testing and interaction based testing into the state/behavior verification dichotomy and the classical/mockist TDD dichotomy. I've also adjusted my vocabulary to match that of the Gerard Meszaros's upcoming xUnit patterns book.)

Regular Tests

I'll begin by illustrating the two styles with a simple example. (The example is in Java, but the principles make sense with any object-oriented language.) We want to take an order object and fill it from a warehouse object. The order is very simple, with only one product and a quantity. The warehouse holds inventories of different products. When we ask an order to fill itself from a warehouse there are two possible responses. If there's enough product in the warehouse to fill the order, the order becomes filled and the warehouse's amount of the product is reduced by the appropriate amount. If there isn't enough product in the warehouse then the order isn't filled and nothing happens in the warehouse.

These two behaviors imply a couple of tests, these look like pretty conventional JUnit tests.

```
public class OrderStateTester extends TestCase {
  private static String TALISKER = "talisker";
  private static String HIGHLAND_PARK = "Highland Park";
  private Warehouse warehouse = new WarehouseImpl();

  protected void setUp() throws Exception {
    warehouse.add(TALISKER, 50);
    warehouse.add(HIGHLAND_PARK, 25);
  }
  public void testOrderIsFilledIfEnoughInWarehouse() {
    Order order = new Order(TALISKER, 50);
    order.fill(warehouse);
```

```
    assertTrue(order.isFilled());
    assertEquals(0, warehouse.getInventory(TALISKER));
  }
  public void testOrderDoesNotRemoveIfNotEnough() {
    Order order = new Order(TALISKER, 51);
    order.fill(warehouse);
    assertFalse(order.isFilled());
    assertEquals(50, warehouse.getInventory(TALISKER));
  }
```

xUnit tests follow a typical four phase sequence: setup, exercise, verify, teardown. In this case the setup phase is done partly in the setUp method (setting up the warehouse) and partly in the test method (setting up the order). The call to `order.fill` is the exercise phase. This is where the object is prodded to do the thing that we want to test. The assert statements are then the verification stage, checking to see if the exercised method carried out its task correctly. In this case there's no explicit teardown phase, the garbage collector does this for us implicitly.

During setup there are two kinds of object that we are putting together. Order is the class that we are testing, but for `Order.fill` to work we also need an instance of Warehouse. In this situation Order is the object that we are focused on testing. Testing-oriented people like to use terms like object-under-test or system-under-test to name such a thing. Either term is an ugly mouthful to say, but as it's a widely accepted term I'll hold my nose and use it. Following Meszaros I'll use System Under Test, or rather the abbreviation SUT.

So for this test I need the SUT (`Order`) and one collaborator (`warehouse`). I need the warehouse for two reasons: one is to get the tested behavior to work at all (since `Order.fill` calls warehouse's methods) and secondly I need it for verification (since one of the results of Order.fill is a potential change to the state of the warehouse). As we explore this topic further you'll see there we'll make a lot of the distinction between SUT and collaborators. (In the earlier version of this article I referred to the SUT as the primary object and collaborators and secondary objects)

This style of testing uses **state verification**: which means that we determine whether the exercised method worked correctly by examining the state of the SUT and its collaborators after the method was exercised. As we'll see, mock objects enable a different approach to verification.

Tests with Mock Objects

Now I'll take the same behavior and uses mock objects. For this code I'm using the jMock library for defining mocks. jMock is a java mock object library. There are other mock object libraries out there, but this one is an up to date library written by the originators of the technique, so it makes a good one to start with.

```
public class OrderInteractionTester extends MockObjectTestCase {
  private static String TALISKER = "Talisker";

  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new Order(TALISKER, 50);
    Mock warehouseMock = new Mock(Warehouse.class);
```

```
    //setup - expectations
    warehouseMock.expects(once()).method("hasInventory")
      .with(eq(TALISKER),eq(50))
      .will(returnValue(true));
    warehouseMock.expects(once()).method("remove")
      .with(eq(TALISKER), eq(50))
      .after("hasInventory");

    //exercise
    order.fill((Warehouse) warehouseMock.proxy());

    //verify
    warehouseMock.verify();
    assertTrue(order.isFilled());
  }

  public void testFillingDoesNotRemoveIfNotEnoughInStock() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);

    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy());

    assertFalse(order.isFilled());
  }
```

Concentrate on `testFillingRemovesInventoryIfInStock` first, as I've taken a couple of shortcuts with the later test.

To begin with, the setup phase is very different. For a start it's divided into two parts: data and expectations. The data part sets up the objects we are interested in working with, in that sense it's similar to the traditional setup. The difference is in the objects that are created. The SUT is the same - an order. However the collaborator isn't a warehouse object, instead it's a mock warehouse - technically an instance of the class `Mock`.

The second part of the setup creates expectations on the mock object.The expectations indicate which methods should be called on the mocks when the SUT is exercised.

Once all the expectations are in place I exercise the SUT. After the exercise I then do verification, which has two aspects. I run asserts against the SUT - much as before. However I also verify the mocks - checking that they were called according to their expectations.

The key difference here is how we verify that the order did the right thing in its interaction with the warehouse. With state verification we do this by asserts against the warehouse's state. Mocks use **behavior verification**, where we instead check to see if the order made the correct calls on the warehouse. We do this check by telling the mock what to expect during setup and asking the mock to verify itself during verification. Only the order is checked using asserts, and if the the method doesn't change the state of the order there's no asserts at all.

In the second test I do a couple of different things. Firstly I create the mock differently, using using the `mock` method in MockObjectTestCase rather than the constructor. This is a convenience method in the jMock library that means that I don't need to explicitly call verify later on, any mock created with the convenience method is automatically verified at the end of the test. I could have done this is the first test

too, but I wanted to show the verification more explicitly to show how testing with mocks works.

The second different thing in the second test case is that I've relaxed the constraints on the expectation by using `withAnyArguments`. The reason for this is that the first test checks that the number is passed to the warehouse, so the second test need not repeat that element of the test. If the logic of the order needs to be changed later, then only one test will fail, easing the effort of migrating the tests. As it turns out I could have left `withAnyArguments` out entirely, as that is the default.

Using EasyMock

There are a number of mock object libraries out there. One that I come across a fair bit is EasyMock, both in its java and .NET versions. EasyMock also enable behavior verification, but has a couple of differences in style with jMock which are worth discussing. Here are the familiar tests again:

```
public class OrderEasyTester extends TestCase {
  private static String TALISKER = "Talisker";

  private MockControl warehouseControl;
  private Warehouse warehouseMock;

  public void setUp() {
    warehouseControl = MockControl.createControl(Warehouse.class);
    warehouseMock = (Warehouse) warehouseControl.getMock();
  }

  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new Order(TALISKER, 50);

    //setup - expectations
    warehouseMock.hasInventory(TALISKER, 50);
    warehouseControl.setReturnValue(true);
    warehouseMock.remove(TALISKER, 50);
    warehouseControl.replay();

    //exercise
    order.fill(warehouseMock);

    //verify
    warehouseControl.verify();
    assertTrue(order.isFilled());
  }

  public void testFillingDoesNotRemoveIfNotEnoughInStock() {
    Order order = new Order(TALISKER, 51);

    warehouseMock.hasInventory(TALISKER, 51);
    warehouseControl.setReturnValue(false);
    warehouseControl.replay();

    order.fill((Warehouse) warehouseMock);

    assertFalse(order.isFilled());
    warehouseControl.verify();
  }
}
```

EasyMock uses a record/replay metaphor for setting expectations. For each object you wish to mock you

create a control and mock object. The mock satisfies the interface of the secondary object, the control gives you additional features. To indicate an expectation you call the method, with the arguments you expect on the mock. You follow this with a call to the control if you want a return value. Once you've finished setting expectations you call replay on the control - at which point the mock finishes the recording and is ready to respond to the primary object. Once done you call verify on the control.

It seems that while people are often fazed at first sight by the record/replay metaphor, they quickly get used to it. It has an advantage over the constraints of jMock in that you are making actual method calls to the mock rather than specifying method names in strings. This means you get to use code-completion in your IDE and any refactoring of method names will automatically update the tests. The downside is that you can't have the looser constraints.

The developers of jMock are working on a new version which will use other techniques to allow you use actual method calls.

The Difference Between Mocks and Stubs

When they were first introduced, many people easily confused mock objects with the common testing notion of using stubs. Since then it seems people have better understood the differences (and I hope the earlier version of this paper helped). However to fully understand the way people use mocks it is important to understand mocks and other kinds of test doubles. ("doubles"? Don't worry if this is a new term to you, wait a few paragraphs and all will be clear.)

When you're doing testing like this, you're focusing on one element of the software at a time - hence the common term unit testing. The problem is that to make a single unit work, you often need other units - hence the need for some kind of warehouse in our example.

In the two styles of testing I've shown above, the first case uses a real warehouse object and the second case uses a mock warehouse, which of course isn't a real warehouse object. Using mocks is one way to not use a real warehouse in the test, but there are other forms on unreal objects used in testing like this.

The vocabulary for talking about this soon gets messy - all sorts of words are used: stub, mock, fake, dummy. For this article I'm going to follow the vocabulary of Gerard Meszaros's upcoming book. It's not what everyone uses, but I think it's a good vocabulary and since it's my essay I get to pick which words to use.

Meszaros uses the term **Test Double** as the generic term for any kind of pretend object used in place of a real object for testing purposes. The name comes from the notion of a Stunt Double in movies. (One of his aims was to avoid using any name that was already widely used.) Meszaros then defined four particular kinds of double:

- **Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists.
- **Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example).
- **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls,

such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.
- **Mocks** are what we are talking about here: objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

Of these kinds of doubles, only mocks insist upon behavior verification. The other doubles can, and usually do, use state verification. Mocks actually do behave like other doubles during the exercise phase, as they need to make the SUT believe it's talking with its real collaborators - but mocks differ in how the setup and the verification phases.

To explore test doubles a bit more, we need to extend our example. Many people only use a test double if the real object is awkward to work with. A more common case for a test double would be if we said that we wanted to send an email message if we failed to fill an order. The problem is that we don't want to send actual email messages out to customers during testing. So instead we create a test double of our email system, one that we can control and manipulate.

Here we can begin to see the difference between mocks and stubs. If we were writing a test for this mailing behavior, we might write a simple stub like this.

```
public interface MailService {
  public void send (Message msg);
}

public class MailServiceStub implements MailService {
  private List<Message> messages = new ArrayList<Message>();
  public void send (Message msg) {
    messages.add(msg);
  }
  public int numberSent() {
    return messages.size();
  }
}
```

We can then use state verification on the stub like this.

```
class OrderStateTester...
  public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    MailServiceStub mailer = new MailServiceStub();
    order.setMailer(mailer);
    order.fill(warehouse);
    assertEquals(1, mailer.numberSent());
  }
```

Of course this is a very simple test - only that a message has been sent. We've not tested it was send to the right person, or with the right contents, but it will do to illustrate the point.

Using mocks this test would look quite different.

```
class OrderInteractionTester...
  public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);
    Mock mailer = mock(MailService.class);
    order.setMailer((MailService) mailer.proxy());

    mailer.expects(once()).method("send");
```

```
    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy());
  }
}
```

In both cases I'm using a test double instead of the real mail service. There is a a difference in that the stub uses state verification while the mock uses behavior verification.

In order to use state verification on the stub, I need to make some extra methods on the stub to help with verification. As a result the stub implements `MailService` but adds extra test methods.

Mock objects always use behavior verification, a stub can go either way. Meszaros refers to stubs that use behavior verification as a Test Spy. The difference is in how exactly the double runs and verifies and I'll leave that for you to explore on your own.

Classical and Mockist Testing

Now I'm at the point where I can explore the second dichotomy: that between classical and mockist TDD. The big issue here is *when* to use a mock (or other double).

The **classical TDD** style is to use real objects if possible and a double if it's awkward to use the real thing. So a classical TDDer would use a real warehouse and a double for the mail service. The kind of double doesn't really matter that much.

A **mockist TDD** practitioner, however, will always use a mock for any object with interesting behavior. In this case for both the warehouse and the mail service.

Although the various mock frameworks were designed with mockist testing in mind, many classicists find them useful for creating doubles.

An important offshoot of the mockist style is that of Behavior Driven Development (BDD). BDD was originally developed by my colleague Dan North as a technique to better help people learn Test Driven Development by focusing on how TDD operates as a design technique. This led to renaming tests as behaviors to better explore where TDD helps with thinking about what an object needs to do. BDD takes a mockist approach, but it expands on this, both with its naming styles, and with its desire to integrate analysis within its technique. I won't go into this more here, as the only relevance to this article is that BDD is another variation on TDD that tends to use mockist testing. I'll leave it to you to follow the link for more information.

Choosing Between the Differences

In this article I've explained a pair of differences: state or behavior verification / classic or mockist TDD. What are the arguments to bear in mind when making the choices between them? I'll begin with the state versus behavior verification choice.

The first thing to consider is the context. Are we thinking about a easy collaboration, such as order and warehouse, or an awkward one, such as order and mail service?

If it's an easy collaboration then the choice is simple. If I'm a classic TDDer I don't use a mock, stub or any kind of double. I use a real object and state verification. If I'm a mockist TDDer I use a mock and behavior verification. No decisions at all.

If it's an awkward collaboration, then there's no decision if I'm a mockist - I just use mocks and behavior verification. If I'm a classicist then I do have a choice, but it's not a big deal which one to use. Usually classicists will decide on a case by case basis, using the easiest route for each situation.

So as we see, state versus behavior verification is mostly not a big decision. The real issue is between classic and mockist TDD. As it turns out the characteristics of state and behavior verification do affect that discussion, and that's where I'll focus most of my energy.

But before I do, let me throw in an edge case. Occasionally you do run into things that are really hard to use state verification on, even if they aren't awkward collaborations. A great example of this is a cache. The whole point of a cache is that you can't tell from its state whether the cache hit or missed - this is a case where behavior verification would be the wise choice for even a hard core classical TDDer. I'm sure there are other exceptions in both directions.

As we delve into the classic/mockist choice, there's lots of factors to consider, so I've broken them out into rough groups.

Driving TDD

Mock objects came out of the XP community, and one of the principal features of XP is its emphasis on Test Driven Development - where a system design is evolved through iteration driven by writing tests.

Thus it's no surprise that the mockists particularly talk about the effect of mockist testing on a design. In particular they advocate a style called need-driven development. With this style you begin developing a story by writing your first test for the outside of your system, making some interface object your SUT. By thinking through the expectations upon the collaborators, you explore the interaction between the SUT and its neighbors - effectively designing the outbound interface of the SUT.

Once you have your first test running, the expectations on the mocks provide a specification for the next step and a starting point for the tests. You turn each expectation into a test on a collaborator and repeat the process working your way into the system one SUT at a time. This style is also referred to as outside-in, which is a very descriptive name for it. It works well with layered systems. You first start by programming the UI using mock layers underneath. Then you write tests for the lower layer, gradually stepping through the system one layer at a time. This is a very structured and controlled approach, one that many people believe is helpful to guide newcomers to OO and TDD.

Classic TDD doesn't provide quite the same guidance. You can do a similar stepping approach, using fake methods instead of mocks. To do this you whenever you need something from a collaborator you just hard-code exactly the response the test requires to make the SUT work. Then one you're green with that you replace the hard coded response with a the proper code.

But classic TDD can do other things too. A common style is middle-out. In this style you take a feature and decide what you need in the domain for this feature to work. You get the domain objects to do what you need and once they are working you layer the UI on top. Doing this you might never need to fake anything. A lot of people like this because it focuses attention on the domain model first, which helps keep domain logic from leaking into the UI.

I should stress that both mockist and classicists do this one story at a time. There is a school of thought that builds application layer by layer, not starting one layer until another is complete. Both classicists and mockists tend to have an agile background and prefer fine-grained iterations. As a result they work feature by feature rather than layer by layer.

Fixture Setup

With classic TDD, you have to create not just the SUT but also all the collaborators that the SUT needs in response to the test. While the example only had a couple of objects, real tests often involve a large amount of secondary objects. Usually these objects are created and torn down with each run of the tests.

Mockist tests, however, only need to create the SUT and mocks for its immediate neighbors. This can avoid some of the involved work in building up complex fixtures (At least in theory. I've come across tales of pretty complex mock setups, but that may be due to not using the tools well.)

In practice, classic testers tend to reuse complex fixtures as much as possible. In the simplest way you do this by putting fixture setup code into the xUnit setup method. More complicated fixtures need to be used by several test classes, so in this case you create special fixture generated classes. I usually call these Object Mothers, based on a naming convention used on an early ThoughtWorks XP project. Using mothers is essential in larger classic testing, but the mothers are additional code that need to be maintained and any changes to the mothers can have significant ripple effects through the tests. There also may be a performance cost in setting up the fixture - although I haven't heard this to be a serious problem when done properly. Most fixture objects are cheap to create, those that aren't are usually doubled.

As a result I've heard both styles accuse the other of being too much work. Mockists say that creating the fixtures is a lot of effort, but classicists say that this is reused but you have to create mocks with every test.

Test Isolation

If you introduce a bug to a system with mockist testing, it will usually cause only tests whose SUT contains the bug to fail. With the classic approach, however, any tests of client objects can also fail, which leads to failures where the buggy object is used as a collaborator in another object's test. As a result a failure in a highly used object causes a ripple of failing tests all across the system.

Mockist testers consider this to be a major issue; it results in a lot of debugging in order to find the root of the error and fix it. However classicists don't express this as a source of problems. Usually the culprit is relatively easy to spot by looking at which tests fail and the developers can tell that other failures are derived from the root fault. Furthermore if you are testing regularly (as you should) then you know the breakage was caused by what you last edited, so it's not difficult to find the fault.

One factor that may be significant here is the granularity of the tests. Since classic tests exercise multiple real objects, you often find a single test as the primary test for a cluster of objects, rather than just one. If that cluster spans many objects, then it can be much harder to find the real source of a bug. What's

happening here is that the tests are too coarse grained.

It's quite likely that mockist tests are less likely to suffer from this problem, because the convention is to mock out all objects beyond the primary, which makes it clear that finer grained tests are needed for collaborators. That said, it's also true that using overly coarse grained tests isn't necessarily a failure of classic testing as a technique, rather a failure to do classic testing properly. A good rule of thumb is to ensure that you separate fine-grained tests for every class. While clusters are sometimes reasonable, they should be limited to only very few objects - no more than half a dozen. In addition, if you find yourself with a debugging problem due to overly coarse-grained tests, you should debug in a test driven way, creating finer grained tests as you go.

In essence classic xunit tests are not just unit tests, but also mini-integration tests. As a result many people like the fact that client tests may catch errors that the main tests for an object may have missed, particularly probing areas where classes interact. Mockist tests lose that quality. In addition you also run the risk that expectations on mockist tests can be incorrect, resulting in unit tests that run green but mask inherent errors.

It's at this point that I should stress that whichever style of test you use, you must combine it with coarser grained acceptance tests that operate across the system as a whole. I've often come across projects which were late in using acceptance tests and regretted it.

Coupling Tests to Implementations

When you write an mockist test, you are testing the outbound calls of the SUT to ensure it talks properly to its suppliers. A classic test only cares about the final state - not how that state was derived. Mockist tests are thus more coupled to the implementation of a method. Changing the nature of calls to collaborators usually cause a mockist test to break.

This coupling leads to a couple of concerns. The most important one is the effect on Test Driven Development. With mockist testing, writing the test makes you think about the implementation of the behavior - indeed mockist testers see this as an advantage. Classicists, however, think that it's important to only think about what happens from the external interface and to leave all consideration of implementation until after you're done writing the test.

Coupling to the implementation also interferes with refactoring, since implementation changes are much more likely to break tests than with classic testing.

This can be worsened by the nature of mock toolkits. Often mock tools specify very specific method calls and parameter matches, even when they aren't relevant to this particular test. One of the aims of the jMock toolkit is to be more flexible in its specification of the expectations to allow expectations to be looser in areas where it doesn't matter, at the cost of using strings that can make refactoring more tricky.

Design Style

One of the most fascinating aspects of these testing styles to me is how they affect design decisions. As I've talked with both types of tester I've become aware of a few differences between the designs that the styles encourage, but I'm sure I'm barely scratching the surface.

I've already mentioned a difference in tackling layers. Mockist testing supports an outside-in approach while developers who prefer a domain model out style tend to prefer classic testing.

On a smaller level I noticed that mockist testers tend to ease away from methods that return values, in favor of methods that act upon a collecting object. Take the example of the behavior of gathering information from a group of objects to create a report string. A common way to do this is to have the reporting method call string returning methods on the various objects and assemble the resulting string in a temporary variable. A mockist tester would be more likely to pass a string buffer into the various objects and get them to add the various strings to the buffer - treating the string buffer as a collecting parameter.

Mockist testers do talk more about avoiding 'train wrecks' - method chains of style of `getThis().getThat().getTheOther()`. Avoiding method chains is also known as following the Law of Demeter. While method chains are a smell, the opposite problem of middle men objects bloated with forwarding methods is also a smell. (I've always felt I'd be more comfortable with the Law of Demeter if it were called the Suggestion of Demeter.)

One of the hardest things for people to understand in OO design is the "Tell Don't Ask" principle, which encourages you to tell an object to do something rather than rip data out of an object to do it in client code. Mockists say that using mockist testing helps promote this and avoid the getter confetti that pervades too much of code these days. Classicists argue that there plenty other ways to do this.

An acknowledged issue with state-based verification is that it can lead to creating query methods only to support verification. It's never comfortable to add methods to the API of an object purely for testing, using behavior verification avoids that problem. The counter-argument to this is that such modification are usually minor in practice.

Mockists favor role interfaces and assert that using this style of testing encourages more role interfaces, since each collaboration is mocked separately and is thus more likely to be turned into a role interface. So in my example above using a string buffer for generating a report, a mockist would be more likely to invent a particular role that makes sense in that domain, which *may* be implemented by a string buffer.

It's important to remember that this difference in design style is key motivator for most mockists. TDD's origins were a desire to get strong automatic regression testing that supported evolutionary design. Along the way its practitioners discovered that writing tests first made a significant improvement to the design process. Mockists have a strong idea of what kind of design is a good design and have developed mock libraries primarily to help people develop this design style.

So should I be a classicist or a mockist?

I find this a difficult question to answer with confidence. Personally I've always been a old fashioned classic TDDer and thus far I don't see any reason to change. I don't see any compelling benefits for mockist TDD, and am concerned about the consequences of coupling tests to implementation.

This has particularly struck me when I've observed a mockist programmer. I really like the fact that while writing the test you focus on the result of the behavior, not how it's done. A mockist is constantly thinking about how the SUT is going to be implemented in order to write the expectations. This feels really unnatural to me.

I also suffer from the disadvantage of not trying mockist TDD on anything more than toys. As I've

learned from Test Driven Development itself, it's often hard to judge a technique without trying it seriously. I do know many good developers who are very happy and convinced mockists. So although I'm still a convinced classicist, I'd rather present both arguments as fairly as I can so you can make your own mind up.

So if mockist testing sounds appealing to you, I'd suggest giving it a try. It's particularly worth trying if you are having problems in some of the areas that mockist TDD is intended to improve. I see two main areas here. One is if you're spending a lot of time debugging when tests fail because they aren't breaking cleanly and telling you where the problem is. (You could also improve this by using classic TDD on finer-grained clusters.) The second area is if your objects don't contain enough behavior, mockist testing may encourage the development team to create more behavior rich objects.

Final Thoughts

As interest in unit testing, the xunit frameworks and Test Driven Development has grown; more and more people are running into mock objects. A lot of the time people learn a bit about the mock object frameworks, without fully understanding the mockist/classical divide that underpins them. Whichever side of that divide you lean on, I think it's useful to understand this difference in views. While you don't have to be a mockist to find the mock frameworks handy, it is useful to understand the thinking that guides many of the design decisions of the software.

The purpose of this article was, and is, to point out these differences and to lay out the trade-offs between them. There is more to mockist thinking than I've had time to go into, particularly its consequences on design style. I hope that in the next few years we'll see more written on this and that will deepen our understanding of the fascinating consequences of writing tests before the code.

Further Reading

For a thorough overview of xunit testing practice, keep an eye out for Gerard Meszaros's forthcoming book (disclaimer: it's in my series). He also maintains a web site with the patterns from the book.

To find out more about TDD, the first place to look is Kent's book.

To find out more about the mockist style of testing, the first place to look is the mockobjects.com where Steve Freeman and Nat Price advocate the mockist point of view with papers and a worthwhile blog. In particular read the excellent OOPSLA paper. For more on Behavior Driven Development, a different offshoot of TDD that is very mockist in style, start with Dan North's introduction.

You can also find out more about these techniques by looking at the tool websites for jMock, nMock, EasyMock, and the .NET EasyMock. (There are other mock tools out there, don't consider this list to be complete.)

XP2000 saw the original mock objects paper, but it's rather outdated now.

Significant Revisions

02 Jan 07: Split the original distinction of state-based versus interaction-based testing into two: state versus behavior verification and classic versus mockist TDD. I also made various vocabulary changes to bring it into line with Gerard Meszaros's book of xunit patterns.

08 Jul 04: First published