# Mock Roles, not Objects

Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes
ThoughtWorks UK
Berkshire House, 168-173 High Holborn
London WC1V 7AA
{sfreeman, npryce, tmackinnon, jwalnes} @thoughtworks.com

## ABSTRACT
Mock Objects is an extension to Test-Driven Development that supports good Object-Oriented design by guiding the discovery of a coherent system of types within a code base. It turns out to be less interesting as a technique for isolating tests from third-party libraries than is widely thought. This paper describes the process of using Mock Objects with an extended example and reports best and worst practices gained from experience of applying the process. It also introduces jMock, a Java framework that embodies our collective experience.

## Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques, Object-Oriented design methods

## General Terms
Design, Verification.

## Keywords
Test-Driven Development, Mock Objects, Java..

## 1. INTRODUCTION
Mock Objects is misnamed. It is really a technique for identifying types in a system based on the roles that objects play.

In [10] we introduced the concept of *Mock Objects* as a technique to support Test-Driven Development. We stated that it encouraged better structured tests and, more importantly, improved domain code by preserving encapsulation, reducing dependencies and clarifying the interactions between classes. This paper describes how we have refined and adjusted the technique based on our experience since then. In particular, we now understand that the most important benefit of Mock Objects is what we originally called "interface discovery". We have also reimplemented our framework to support dynamic generation of Mock Objects, based on this experience.

The rest of this section establishes our understanding of Test-Driven Development and good practice in Object-Oriented Programming, and then introduces the Mock Object concept. The rest of the paper introduces Need-Driven Development, as expressed using Mock Objects, and shows a worked example. Then we discuss our experiences of developing with Mock Objects and describe how we applied these to jMock, our Mock Object framework.



### 1.1 Test-Driven Development
In Test-Driven Development (TDD), programmers write tests, called *Programmer Tests*, for a unit of code before they write the code itself [1]. Writing tests is a *design* activity, it specifies each requirement in the form of an executable example that can be shown to work. As the code base grows, the programmers refactor it [4], improving its design by removing duplication and clarifying its intent. These refactorings can be made with confidence because the test-first approach, by definition, guarantees a very high degree of test coverage to catch mistakes.

This changes design from a process of *invention*, where the developer thinks hard about what a unit of code should do and then implements it, to a process of *discovery*, where the developer adds small increments of functionality and then extracts structure from the working code.

Using TDD has many benefits but the most relevant is that it directs the programmer to think about the design of code from its intended use, rather than from its implementation. TDD also tends to produce simpler code because it focuses on immediate requirements rather than future-proofing and because the emphasis on refactoring allows developers to fix design weaknesses as their understanding of the domain improves.

### 1.2 Object-Oriented Programming
A running Object-Oriented (OO) program is a web of objects that collaborate by sending messages to each other. As described by Beck and Cunningham [2], "no object is an island. … All objects stand in relationship to others, on whom they rely for services and control". The visible behaviour of each object is defined in terms of how it sends messages and returns results in response to receiving messages.

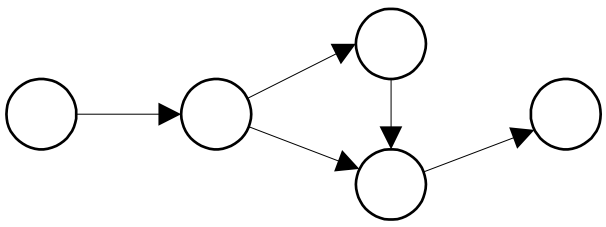

**Figure 1. A Web of Collaborating Objects**

The benefit of OO is that it defines a unit of modularity which is internally coherent but has minimal coupling to the rest of the system. This makes it easy to modify software by changing how

objects are composed together into an application. To achieve this flexibility in practice, objects in a well-designed system should only send messages to their immediate neighbours, otherwise known as the Law of Demeter [15].

Note that the immediate neighbours of an object do not include objects whose references are returned from a call to another object. Programmers should avoid writing code that looks like:

```
dog.getBody().getTail().wag();
```

colloquially known as a "Train Wreck". This is bad because this one line depends on the interfaces and implied structure of three different objects. This style laces structural dependencies between unrelated objects throughout a code base. The solution is described by the heuristic "Tell, Don't Ask" [7], so we rewrite our example as:

```
dog.expressHappiness();
```

and let the implementation of the dog decide what this means.

Given an object that talks only to its immediate neighbours, we can describe it in terms of the services it provides and the services it requires from those neighbours. We call those required services *outgoing interfaces* because that is how the object calls *out* to other objects.

## 1.3 Test-Driven Development of Object Oriented Programs

If we concentrate on an object's external interactions, we can test it by calling one of its services and tracking the resulting interactions with its neighbours. If we are programming test-first, we can define those tests in terms of outgoing interfaces (which might not yet exist) because that's how we can tell whether an action has succeeded.

For example, we decide that `dog.expressHappiness()` has succeeded when its implementation has called `body.wagTail()`. This is a design decision that we make when developing the `dog` object about how to implement one of its services (note that we're still avoiding a Train Wreck by not asking the body about its implementation of a tail).

If the `DogBody` object does not yet have a `wagTail()` method, this test has identified a new requirement that it must fulfil. We don't want to stop now and implement the new feature, because that would be a distraction from the current task and because the implementation of `wagTail()` might trigger an unpredictably long chain of further implementations. Instead we provide a false implementation of the `DogBody` object that pretends to implement the method. Now we can instrument that false object to see if `wagTail()` is actually called when testing `expressHappiness()`.

To summarise, we test an object by replacing its neighbours with objects that *test* that they are called as expected and *stub* any behaviour that the caller requires. These replacements are called mock objects. We call the technique of TDD with mock objects, *Mock Objects.*

# 2. MOCK OBJECTS AND NEED-DRIVEN DEVELOPMENT

*Mock Objects* changes the focus of TDD from thinking about the changes in state of an object to thinking about its interactions with other objects. We use Mock Objects to let us write the code under test *as if* it had everything it needs from its environment. This process shows us what an object's environment *should* be so we can then provide it.

## 2.1 Need-Driven Development

A core principle of Lean Development is that value should be pulled into existence from demand, rather than pushed from implementation: "The effect of 'pull' is that production is not based on forecast; commitment is delayed until demand is present to indicate what the customer really wants." [16].

This is the flow of programming with Mock Objects. By testing an object in isolation, the programmer is forced to consider an object's interactions with its collaborators in the abstract, possibly before those collaborators exist. TDD with Mock Objects guides interface design by the services that an object *requires*, not just those it *provides*. This process results in a system of narrow interfaces each of which defines a role in an interaction between objects, rather than wide interfaces that describe all the features provided by a class. We call this approach *Need-Driven Development*.

For example, Figure 2 depicts a test of object A. To fulfil the needs of A, we discover that it needs a service S. While testing A we mock the responsibilities of S without defining a concrete implementation.

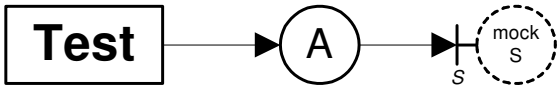


**Figure 2. Interface Discovery**

Once we have implemented A to satisfy its requirements we can switch focus and implement an object that performs the role of S. This is shown as object B in Figure 3. This process will then discover services required by B, which we again mock out until we have finished our implementation of B.

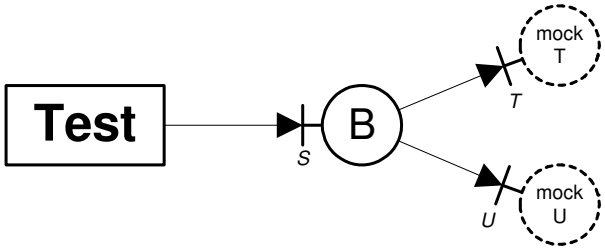


**Figure 3. Iterative Interface Discovery**

We continue this process until we reach a layer that implements real functionality in terms of the system runtime or external libraries.

The end result is that our application is structured as a composition of objects that communicate through narrowly defined role interfaces (Figure 4). As another writer put it, "From each according to his abilities, to each according to his needs!" [11].

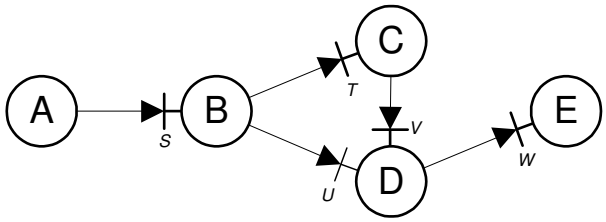


**Figure 4. A Web of Objects Collaborating Through Roles**

Our experience is that systems we produce this way tend towards very flat class hierarchies. This avoids well-known problems, such as the Fragile Base Class [12], which make systems harder to understand and modify.

This process is similar to traditional Top-Down Development, in which the programmer starts at the highest level of abstraction and proceeds, layer by layer, to fill in the detail. The intention is that each layer of code is written in a coherent terminology, defined in terms of the next level of abstraction. This is difficult to achieve in practice because the most important decisions have to be taken early and it is hard to avoid duplication across lower level components. TDD mitigates this by including Refactoring in its process.

Programming from the Bottom-Up has different risks. All the authors have had the experience of developing a supporting class in isolation, as part of a larger task, only to find that the result was not right because we had misunderstood something.

We find that Need-Driven Development helps us stay focussed on the requirements in hand and to develop coherent objects.

# 3. A WORKED EXAMPLE

To illustrate the technique, we will work through an example. Consider a component that caches key-based retrievals from an object loading framework. The instances become invalid a given time after they've been loaded, so sometimes we want to force a reload.

With Mock Objects we use a common structure, identified in [10], for programmer tests.

1. Create the test fixture including any mock objects
2. Define expectations and stubs on the mock objects
3. Invoke the method to be tested
4. Verify expectations and assert any postconditions

This makes the tests easier to read.

## 3.1 An Object Loader

Our first programmer test should be a simple success case, to load and return objects that are not in the cache. In the case, we *expect* to call the loader exactly once with each key, and we need to check that the right value is returned from the cache. Using the jMock framework, described in detail later, we can write out a JUnit [9] test for this (we have left out instance creation for brevity. `KEY` and `VALUE` are constants in the test case, not part of the jMock framework).

```
public class TimedCacheTest {
  public void testLoadsObjectThatIsNotCached() {
    // we expect to call load
    // exactly once with the key,
    // this will return the given value
    mockLoader.expect(once())
      .method("load").with( eq(KEY) )
      .will(returnValue(VALUE));

    mockLoader.expect(once())
      .method("load").with( eq(KEY2) )
      .will(returnValue(VALUE2));

    assertSame( "should be first object",
                VALUE, cache.lookup(KEY) );
    assertSame( "should be second object",
                VALUE2, cache.lookup(KEY2) );
    mockLoader.verify();
  }
}
```

jMock uses reflection to match methods by name and parameters. The jMock syntax for defining expectation is unusual, the first expectation is equivalent to:

```
expectation = mockLoader.expect(once());
expectation.method("load");
expectation.with( eq(KEY) );
expectation.will(returnValue(VALUE));
```

We daisy-chain these calls to make the tests more compact and readable; this is discussed later.

The test implies that the Cache has relationships with something that represents an object loader.

```
  TimedCache cache = new TimedCache (
    (ObjectLoader)mockLoader;
  );
```

The test says that we should call the Object Loader exactly once for each key to get a value. We call `verify()` on the mock object at the end of the test to check that our expectations have been met. An implementation that passes the test would be:

```
public class TimedCache {
  private ObjectLoader loader;
  // constructor
  public Object lookup(Object key) {
    return loader.load(key);
  }
}
```

### *3.1.1 Discovering a New Interface*

What the test actually checks is that the Cache talks correctly to any object that implements `ObjectLoader`; the tests for the real Object Loader will be elsewhere. To write the test, all we need is an empty interface called `ObjectLoader` so we can construct the mock object. To pass the test, all we need is a `load()` method that can accept a key. We have discovered the need for a type:

```
public interface ObjectLoader {
    Object load(Object theKey);
}
```

We have minimised the dependency between our cache and whatever object loading framework we eventually use. The mock object framework also has the advantage that we avoid difficulties with complex setup or changeable data at this level of testing. This leaves us free to think about the relationships between objects, rather than how to get the test infrastructure to work.

## 3.2 Introducing Caching

The next test case is to look up a value twice and not have it loaded the second time. We *expect* to call the loader exactly once with a given key and return the found value. Our second test is:

```
public void testCachedObjectsAreNotReloaded() {
  mockLoader.expect(once())
    .method("load").with( eq(KEY) )
    .will( returnValue(VALUE) );

  assertSame( "loaded object",
              VALUE, cache.lookup(KEY) );
  assertSame( "cached object",
              VALUE, cache.lookup(KEY) );
}
```

We have left out the calls to `verify()` which in practice are handled automatically by the `MockObjectTestCase` class. This test, of course, fails with a message:

```
DynamicMockError: mockObjectLoader: no match found
Invoked: load(<key>)
in:
expected once and has been invoked:
  load( eq(<key>) ), returns <value>
```

This tells us that we have called `load()` a second, unexpected time with the key. The lines after "in:" describe the interactions we expect to have with the Object Loader during the test. We can pass this test by adding a hash map to the Cache; we need a hash map, rather than just a value field, so that the first test will still pass.

```
public Object lookup(Object key) {
  Object value = cachedValues.get(key);
  if (value == null) {
    value = loader.load(key);
    cachedValues.put(key, value);
  }
  return value;
}
```

There is, of course, an implication here that we cannot load null values which we will treat as a requirement. In this case we would also add tests to show what happens when a value is missing.

### *3.2.1 Testing interactions*

By concentrating on the interactions between objects, rather than their state, we can show that the cache does not have to go back to the loader once a value has been retrieved; it calls `lookup()` twice, but fails if `load()` is called more than once.

We also benefit from failing at the right time when the error occurs, rather than at the end of the test. The stack trace takes us to the `load()` within the second `lookup()`, and the failure message tells what has happened and what should have.

## 3.3 Introducing Time

We have a requirement for time-dependent behaviour. We do not want programmer tests to use system time because that makes them subject to non-deterministic failures and timing pauses will slow them down, so we introduce a `Clock` object that returns `Timestamp` objects. We don't want to think too hard just yet about what it means for a value to expire, so we defer that decision to a `ReloadPolicy` object.

This requirement changes the premise of the previous cache hit test, so we'll adapt and rename it. The test is now to look up a value and then look it up again within its lifetime. We *expect* to get a timestamp twice, once for the first load and once for the second look up; we *expect* to call the loader exactly once with the given key and return the found value; and, we *expect* to compare the two timestamps to make sure that the cache is still valid.

For brevity, we will leave out the instantiation of the timestamp objects `loadTime` and `fetchTime`. The test is now:

```
public void
testReturnsCachedObjectWithinTimeout() {
  mockClock.expect(atLeastOnce())
    .method("getCurrentTime").withNoArguments()
    .will( returnValues(loadTime, fetchTime) );

  mockLoader.expect(once())
    .method("load").with( eq(KEY) )
    .will( returnValue(VALUE) );

  mockReloadPolicy.expect(atLeastOnce())
     .method("shouldReload")
       .with( eq(loadTime), eq(fetchTime) )
     .will( returnValue(false) );

  assertSame( "should be loaded object",
              VALUE, cache.lookup(KEY) );
  assertSame( "should be cached object",
              VALUE, cache.lookup(KEY) );
}
```

Again, given the lifetime requirements, we pass the Clock and Reload Policy in to the constructor.

```
TimedCache cache = new TimedCache(
  (ObjectLoader)mockLoader,
  (Clock)mockClock,
  (ReloadPolicy)mockReloadPolicy
);
```

This test, of course, fails with message:

```
AssertionFailedError:
  mockClock: expected method was not invoked:
   expected at least once:
    getCurrentTime(no arguments),
      returns <loadTime>, then returns <fetchTime>
```

This failure was caught during the `verify()` and shows that we need to introduce timing behaviour to the Cache. The next change to `TimedCache` is a bit larger. We add a simple `TimestampedValue` class to hold a timestamp/value pair, and `loadObject()` loads the requested object and inserts it with the current time as a `TimestampedValue` into `cachedValues`.

```
private class TimestampedValue {
  public final Object value;
  public final Timestamp loadTime;
}

public Object lookup(Object theKey) {
  TimestampedValue found =
     (TimestampedValue) cachedValues.get(theKey);

  if( found == null ||
      reloadPolicy.shouldReload(
        found.loadTime, clock.getCurrentTime() )
  {
    found = loadObject(theKey);
  }
  return found.value;
}
```

### 3.3.1 Programming by Composition

You might notice that everything that the `TimedCache` needs is passed into it, either in the constructor or with the method call. This is more or less forced on the programmer by the need to substitute the neighbouring objects with mock implementations. We believe that this is a strength, because it pushes the design towards small, focussed objects that interact only with known collaborators. It also encourages the programmer to create types to represent abstract concepts in the system, such as the `ReloadPolicy`, which gives a clearer separation of concerns in the code.

### 3.3.2 Programming in the Abstract

The test now also checks that the Cache finds the current time twice, once for each lookup, and routes those values correctly to the reload policy. We don't yet have to define what we mean by time or how a value goes stale, we're just concerned with the essential flow of the method. Everything to do with external time is abstracted away into interfaces that we have not yet implemented, just as we abstracted away the object loading infrastructure. This code treats the timestamps as opaque types, so we can use dummy implementations. This leaves us free to concentrate on getting the core caching behaviour right.

## 3.4 Introducing Sequence

We are also concerned that the timestamp for an object is not set before it's loaded into the cache. That is, we *expect* to retrieve the current time after we load the object. We can adjust the test to enforce this sequence.

```
public void
testReturnsCachedObjectWithinTimeout() {
  mockLoader.expect(once())
    .method("load").with( eq(KEY) )
    .will( returnValue(VALUE) );

  mockClock.expect(atLeastOnce())
    .after(mockLoader, "load")
    .method("getCurrentTime").withNoArguments()
    .will( returnValues(loadTime, fetchTime) );

  mockReloadPolicy.expect(atLeastOnce())
     .method("shouldReload")
       .with( eq(loadTime), eq(fetchTime) )
     .will( returnValue(false) );

  assertSame( "should be loaded object",
              VALUE, cache.lookup(KEY) );
  assertSame( "should be cached object",
              VALUE, cache.lookup(KEY) );
}
```

The `after()` clause matches on the identity of an invocation, in this case in a different object. That identity can be set in an `id()` clause with a default, as here, of the method name. This fails, because our implementation of `loadObject()` retrieves the current time into a variable before loading the object, with the message:

```
DynamicMockError: mockClock: no match found
Invoked: getCurrentTime()
in:
  expected at least once:
    getCurrentTime(no arguments),
     after load on mockObjectLoader,
      returns <loadTime>, then returns <fetchTime>
```

This message tells us that we have an invocation of `getCurrentTime()`, but we're actually looking for an invocation of `getCurrentTime()` that occurs after an invocation of `load()`, which is not the same thing. We fix the implementation by moving the call to `Clock`.

### 3.4.1 Varying Levels of Precision

This test now specifies an extra relationship to say that the clock should not be queried until an object has been loaded. This is possible because we're testing interactions between objects rather than final state, so we can catch events at the time they happen. Our use of mock implementations of all the neighbouring objects means that we have somewhere to attach those additional assertions.

On the other hand, we don't care if the `ReloadPolicy` is called more than once, as long as it has the right parameters; it will always return the same result. This means we can weaken its requirement from being called *exactly* once to being called *at least* once. Similarly, jMock can also soften the requirements on the parameters for a mock object using a technique we call Constraints; this is described later.

## 3.5 Introducing a Timeout

Finally, we want to check that a stale value will actually be refreshed from the loader. In this case, we *expect* that the loader will be called twice with the same key and return two different objects. In addition, the reload policy will request a reload, and we expect that the clock will return an extra timestamp for the additional load.

```
public void
testReloadsCachedObjectAfterTimeout() {
  mockClock.expect(times(3))
    .method("getCurrentTime").withNoArguments()
    .will( returnValues(loadTime, fetchTime,
                        reloadTime) );

  mockLoader.expect(times(2))
    .method("load").with( eq(KEY) )
    .will( returnValues(VALUE, NEW_VALUE) );

  mockReloadPolicy.expect(atLeastOnce())
     .method("shouldReload")
       .with( eq(loadTime), eq(fetchTime) )
     .will( returnValue(true) );

  assertSame( "should be loaded object",
              VALUE, cache.lookup(KEY) );
  assertSame( "should be reloaded object",
              NEW_VALUE, cache.lookup(KEY) );
}
```

The existing implementation passes this test. In this case, we might experiment by breaking the code to make sure that this is because the code is correct rather than because the test is incomplete.

As before, this test exercises a timeout without having to wait because we have abstracted out the timing aspects of the Cache. We can force the reload by returning a different value from the `ReloadPolicy`.

## 3.6 Writing Tests Backwards

In practice we have noticed that we write the tests in a different order, one that follows our thinking during TDD.

1. Identify the object we are testing and write the method call, with any required parameters
2. Write expectations to describe the services the object requires from the rest of the system
3. Represent those services as mock objects
4. Create the rest of the context in which the test will execute
5. Define any postconditions
6. Verify the mocks.

As a result of following the "Tell, Don't Ask" principle, we often don't have any postconditions to assert (step 5). This is surprising to programmers who are not thinking about how their objects communicate.

These steps are shown in the example below :

```
public void testReturnsNullIfLoaderNotReady() {

  Mock mockLoader = mock(ObjectLoader.class); // 3
  mockLoader.expect(never())                  // 2
   .method("load").with( eq(KEY) )

  mockLoader.stub()                           // 4
   .method("isReady").withNoArguments()
   .will( returnValue(false) );
  TimedCache cache =
    new TimedCache((ObjectLoader)mockLoader); // 4

  Object result = cache.lookup(KEY);          // 1

  assertNull( "should not have a KEY",        // 5
              result );
  mockLoader.verify();                        // 6
}
```

It is particularly important to make sure that you are clear about the object that you are testing and its role (step 1) as we have often observed this to be a source of confusion when people are having trouble writing tests. Once they have clarified this, it becomes straightforward to proceed from step 2.

## 3.7 Summary

Working through this example has shown how programmers can drive the discovery of object roles by concentrating on the interactions between objects, not their state. Writing tests provides a framework to think about functionality, Mock Objects provides a framework for making assertions about those relationships and for simulating responses.

Programmers can concentrate on the task in hand, assuming that the infrastructure they need will be available because they can build it later. The need to pass mock objects into the target code leads to a object-oriented style based on composition rather than inheritance. All this encourages designs with good separation of concerns and modularity.

Mock Objects also allows programmers to make their tests only as precise as they need to be. The example showed both a more precise assertion, that one invocation must follow another, and a less precise assertion, that a call may be made more than once. The jMock Constraint framework is discussed later.

One flaw with this example is that the requirements for the `TimedCache` itself have not been driven by a higher-level client, as would normally be the case.

# 4. MOCK OBJECTS IN PRACTICE

Between us, the authors have been working with Mock Objects on a wide range of projects for over 5 years. We have also corresponded with other developers who have been using the technique. The longest project was 4 years, and the largest team was 15 developers. We have used it with Java, C#, Ruby, Python, and Javascript, with application scale ranging from enterprise-level to hand-held.

Mock Objects is a design aid, but is no substitute for skilled developers. Our experience is that mock-based tests quickly become too complicated when the system design is weak. The use of mock objects amplifies problems such as tight coupling and misallocated responsibilities. One response to such difficulties is to stop using Mock Objects, but we believe that it is better to use this as a motivator for improving the design. This section describes some of the heuristics that we have found to be helpful.

## 4.1 Only Mock Types You Own

Mock Objects is a design technique so programmers should only write mocks for types that they can change. Otherwise they cannot change the design to respond to requirements that arise from the process. Programmers should not write mocks for fixed types, such as those defined by the runtime or external libraries. Instead they should write thin wrappers to implement the application abstractions in terms of the underlying infrastructure. Those wrappers will have been defined as part of a need-driven test.

We have found this to be a powerful insight to help programmers understand the technique. It restores the pre-eminence of the design in the use of Mock Objects, which has often been overshadowed by its use for testing interactions with third-party libraries.

## 4.2 Don't use getters

The trigger for our original discovery of the technique was when John Nolan set the challenge of writing code without getters. Getters expose implementation, which increases coupling between objects and allows responsibilities to be left in the wrong module. Avoiding getters forces an emphasis on object behaviour, rather than state, which is one of the characteristics of Responsibility-Driven Design.

## 4.3 Be explicit about things that should not happen

A test is a specification of required behaviour and is often read long after the original programmer wrote the test. There are some conditions that are not made clear when they are simply left out of the test. A specification that a method *should not* be called, is not the same as a specification that doesn't mention the method at all. In the latter case, it's not clear to other readers whether a call to the method is an error. We often write tests that specify that methods should not be called, even where not necessary, just to make our intentions clear.

## 4.4 Specify as little as possible in a test

When testing with Mock Objects it is important to find the right balance between an accurate specification of a unit's required behaviour and a flexible test that allows easy evolution of the code base. One of the risks with TDD is that tests become "brittle", that is they fail when a programmer makes unrelated changes to the application code. They have been over-specified to

check features that are an artefact of the implementation, not an expression of some requirement in the object. A test suite that contains a lot of brittle tests will slow down development and inhibit refactoring.

The solution is to re-examine the code and see if either the specification should be weakened, or the object structure is wrong and should be changed. Following Einstein, a specification should be as precise as possible, but not more precise.

## 4.5 Don't use mocks to test boundary objects

If an object has no relationships to other objects in the system, it does not need to be tested with mock objects. A test for such an object only needs to make assertions about values returned from its methods. Typically, these objects store data, perform independent calculations or represent atomic values. While this may seem an obvious thing to say, we have encountered people trying to use mock objects where they don't actually need to.

## 4.6 Don't add behaviour

Mock objects are still stubs and should not add any additional complexity to the test environment, their behaviour should be obvious [10]. We find that an urge to start adding real behaviour to a mock object is usually a symptom of misplaced responsibilities.

A common example of this is when one mock has to interpret its input to return another mock, perhaps by parsing an event message. This introduces a risk of testing the test infrastructure rather than the target code.

This problem is avoided in jMock because its invocation matching infrastructure allows the test to specify expected behaviour. For example:

```
mock.expect(once())
   .method("retrieve").with(eq(KEY1))
   .willReturn(VALUE1);

mock.expect(once())
   .method("retrieve").with(eq(KEY2))
   .willReturn(VALUE2);
```

## 4.7 Only mock your immediate neighbours

An object that has to navigate a network of objects in its implementation is likely to be brittle because it has too many dependencies. One symptom of this is tests that are complex to set up and difficult to read because they have to construct a similar network of mock objects. Unit tests work best when they focus on testing one thing at a time and only setting expectations on objects that are nearest neighbours.

The solution might be to check that you are testing the right object, or to introduce a role to bridge between the object and its surroundings.

## 4.8 Too Many Mocks

A similar problem arises when a test has to pass too many mock objects to the target code, even if they are all immediate neighbours. Again, the tests is likely to be complex to set up and hard to read. Again the solution might be to change misaligned responsibilities, or to introduce an intermediate role. Alternatively, it is possible that the object under test is too large and should be broken up into smaller objects that will be more focussed and easier to test.

## 4.9 Instantiating new objects

It is impossible to test interactions with an object that is created within the target code, including interactions with its constructor. The only solution is to intervene in the creation of the object, either by passing an instance in or by wrapping the call to `new`.

We have found several useful ways of approaching this problem. To pass an instance in, the programmer can either add a parameter to the constructor or the relevant method of the object under test, depending on the relationship between the two objects. To wrap instance creation, the test can either pass in a factory object or add a factory method to the object under test.

The advantage of a factory object is that the test can set expectations on the arguments used to create a new instance. The disadvantage is that this requires a new type. The factory object often represents a useful concept in the domain, such as the `Clock` in our example.

A factory method simply returns a new instance of the type, but can be overridden in a subclass of the target object for testing to return a mock implementation. This is a pragmatic solution which less heavyweight than creating a factory type, and may be effective as an interim implementation.

Some developers propose using techniques such as Aspect Oriented Programming or manipulating class loaders to replace real objects. This is useful for removing external dependencies but does not help to improve the design of the code base.

# 5. MISCONCEPTIONS ABOUT MOCKS

What we mean by "Mock Objects" is often different from what other people mean, and different from what we used to mean. In particular,

## 5.1 Mocks are just Stubs

Stubs are dummy implementations of production code that return canned results. Mock Objects act as stubs, but also include assertions to instrument the interactions of the target object with its neighbours.

## 5.2 Mock Objects should only be used at the boundaries of the system

We believe the opposite, that Mock Objects are most useful when used to drive the design of the code under test. This implies that they are most useful *within* the system where the interfaces can be changed. Mocks and stubs can still be useful for testing interactions with third-party code, especially for avoiding test dependencies, but for us this is a secondary aspect to the technique.

## 5.3 Gather state during the test and assert against it afterwards.

Some implementations set values when methods are called on the Mock Object and then check them at the end of the test. A special case of this is the *Self Shunt* pattern [3] in which the test class implements a Mock itself.

```
public class TimedClassTest
  implements ObjectLoader
{
  final Object RESULT = new Object();
  final Object KEY = new Object();
  int loadCallCount = 0;
  Object lookupKey;

  // ObjectLoader method
  public Object lookup(Object key) {
    loadCallCount++;
    lookupKey = key;
    return LOOKUP_RESULT;
  }

  public testReturnsCachedObjectWithinTimeout() {
    // set up the rest of the test...
    assertSame( "loaded object",
                RESULT, cache.lookup(KEY) );
    assertSame( "cached object",
                RESULT, cache.lookup(KEY) );

    assertEquals("lookup key", KEY, lookupKey);
    assertEquals("load call count",
                 1, loadCallCount);
  }
}
```

This is straightforward and self-contained but has two obvious disadvantages. First, any failures occur after the fact rather than at the time of the error, whereas putting the assertions into the Mock means that the test will fail at the point where the extra call to `load()` happens. Our experience is that immediate failures are easier to understand and fix than *post-hoc* assertions. Second, this approach splits the implementation of the assertion across the test code, raising its intellectual overhead. Our strongest objection, however, is that this approach does not focus the interactions between the object under test and its neighbours, which we believe is key to writing composable, orthogonal code. As the author says, a Self Shunt is likely to be a placeholder implementation as it does not scale well.

## 5.4 Testing using Mock Objects duplicates the code.

Some uses of Mock Objects set up behaviour that shadows the target code exactly, which makes the tests brittle. This is particularly common in tests that mock third-party libraries. The problem here is that the mock objects are not being used to drive the design, but to work with someone else's. At some level, mock objects *should* shadow a scenario for the target code, but only because the design of that code should be driven by the test. Complex mock setup for a test is actually a hint that there is a missing object in the design.

## 5.5 Mock Objects inhibits refactoring because many tests break together.

Some programmers prefer to test clusters of objects so they can refactor code within that cluster without changing the tests. This approach, however, has disadvantages because each test depends on more objects than for Mock Object-based testing. First, a change to a core class because of a new requirement may force changes to multiple tests, especially to test data which is not as amenable to refactoring as code. Second, finding the error when a test does fail can be more complex because the link between the tests and the failing code is less direct; at its worst, this might even require a debugger. Our experience is that Mock Object-based test failures are more focussed and more self-explanatory, reducing the turnaround on code changes.

## 5.6 Using Strings For Method Names is Fragile

Our dynamic mock frameworks look up methods by name using strings. These are not recognised and changed by refactoring development environments when the mocked method is renamed, so related tests will break. Some programmers believe that constantly being forced to repair tests will slow refactoring too much. In practice, types tend to be used more locally in a Mock Object-driven code base, so fewer tests break than might be expected, and those test break cleanly so that the required change is obvious. There is some extra overhead, but we believe it is worth paying for the greatly increased flexibility of the way we can specify expectations.

# 6. JMOCK: A TOOL FOR NEED-DRIVEN DEVELOPMENT

jMock is an open source framework that provides a convenient and expressive API for mocking interfaces, specifying expected invocations and stubbing invoked behaviour. jMock encapsulates the lessons we have learned during the last few years of using mock objects in a test driven process.

The test-driven process, especially when used with pair programming [18], has a rhythm that gives feedback and maintains motivation. The rhythm is broken if the programmers must stop writing the test to write support code.

The first mock object library had this problem: programmers who discovered an interface while writing a test had to stop and write its mock implementation. The jMock API uses dynamic code generation to create mock implementations on the fly at runtime and does everything it can (within the limitations of the Java language) to support programmers when writing and, later, reading expectations.

The main entry point to the jMock API is `MockObjectTestCase`, a class that extends `JUnit`'s `TestCase` with support for using mock objects. `MockObjectTestCase` provides methods that make expectations easy to read and helps the programmer avoid mistakes by automatically verifying mock objects at the end of the test.

Mock objects are created by the `mock(...)` method, which takes a Class object representing an interface type and returns a Mock object that implements that interface. The Mock object can then be cast to the mocked type and passed to the domain code under test.

```
class TimedCacheTest
  extends MockObjectTestCase
{
  Mock mockLoader = mock(ObjectLoader.class);
  TimedCache cache = new TimedCache (
    (ObjectLoader)mockLoader );
  ...
}
```

The Mock object returned from the `mock(...)` method provides methods for setting up expectations.

## 6.1 Defining Expectations

jMock is especially designed for writing tests that are both run and *read* as a form of documentation. Most of the jMock API is concerned with defining readable syntactic sugar for defining expectations. This goal has led to an API that is quite unconventional when compared to typical Java designs because it tries to implement a domain specific embedded language [6] hosted in Java. In particular, the API deliberately breaks the Law of Demeter and does not name methods as verbs in the imperative mood.

An expectation is specified in multiple clauses. The first clause states whether we want to expect or stub an invocation. jMock treats a stub as a degenerate form of expectation that does not actually have to occur. However, the distinction between stubs and expectations is so important to the programmer that jMock makes the distinction obvious in test code.

Subsequent clauses define which method invocations on the mock are tested by the expectation (matching rules), define the stubbed behaviour for matching methods, and optionally identify the expectation so that it can be referenced in the matching rules of subsequent expectations. An expectation contains multiple matching rules and matches invocations that pass all of its rules.

Each clause of an expectation is represented in test code by a method call to an API interface. Each method returns a reference to an interface with which the programmer can define the next clause, which will return another interface for the following clause, and so on. The chain of calls that defines an entire expectation is started by calling `expect()` or `stub()` on the mock itself.

```
mock.expect(expectation)
   .method(method name)
   .with(argument constraints)
   .after(id of prior invocation)
   .match(other matching rule)
   .will(stubbed behaviour)
   .id(id of this invocation);

mock.stub().method(method name)...
```

The names of the chained methods in a statement that sets up the expectation make the expectation easy to understand. The daisy-chain API style ensures that all expectations are specified in a consistent order: expectation or stub, method name, arguments, ordering and other matching rules, stubbed behaviour, identifier. This makes it easier to work with tests that are written by different people.

When used with auto-completion in a development tool, the API acts like a "wizard", guiding the programmer step by step through the task of defining an expectation.

## 6.2 Flexible and Precise Specifications

To avoid the problems of over specification described above, jMock lets the programmer specify expected method calls as constraints that must be met, rather than actual values. Constraints are used to test argument values and even method names. This allows the programmer to ignore aspects of an object's interactions that are unrelated to the functionality being tested.

Constraints are usually used to specify allowable argument values. For example, we can test that a string contains an expected substring while ignoring unimportant details of formatting and punctuation. Although the most common case is that arguments are compared to expected values, the constraints make explicit whether the comparison is actually for equivalence (the `equals` method) or identity (the == operator). It is also common to ignore parameters altogether, which can be specified with the `IS_ANYTHING` constraint.

Constraints are created by "sugar" methods in the `MockObjectTestCase`. The `with` method of the expectation builder interface defines argument constraints. The expectation below specifies that the `pipeFile` method must be called once with two arguments, one of which is equal to the expected `fileName` and the other of which is the `mockPipeline` object.

```
mock.expect(once())
   .method("pipeFile")
   .with(eq(fileName),same(mockPipeline))
   .will( returnValue(fileContent) );
```

It is often useful to match more than just parameter values. For example, it is often useful to match against subsets of an object's methods, such as all Java Bean property getters. In this case, jMock lets the programmer specify a constraint over method names. Along with a mechanism to create default results, this allows us to ignore unrelated aspects of an object's interface and concentrate only on the interesting aspects for the test.

```
mock.stub().method(startingWith("get"))
   .withNoArguments()
   .will(returnADefaultValue);
```

jMock lets the user specify more complex matching rules such as constraints on the order of calls to a mock, or even the order of calls on different mocks. In general, ordering constraints are not necessary, and should be used with care because they can make tests too brittle. jMock minimises this risk by letting the user specify partial orderings between individual invocations. We demonstrated ordering when we introduced sequence in the example.

jMock forces users to specify argument constraints to ensure that tests can easily be read as documentation. We have found that users prefer the resulting clarity, despite the extra typing involved, because it helps them avoid subtle errors.

## 6.3 Extensibility

Although jMock provides a large library of constraints and matching rules, it cannot cover every scenario that a programmer might need. In fact creating constraints specific to your problem domain improves the clarity of your tests. For this reason matching rules and constraints are extensible. Programmers can define their own rules or constraints that seamlessly extend the jMock syntax.

For example, objects that fire events will create a new event object each time an event occurs. To match against an event from a specific object we can write a custom constraint that compares the source of the event to an expected source:

```
mock.expect(once())
   .method("actionPerformed")
   .with(anActionEventFrom(quitButton));
```

jMock is primarily designed to support Need-Driven Development. As such, the API may be less applicable in other scenarios. Users have asked us to modify jMock to help them

perform integration testing, do procedural programming, avoid refactoring poorly designed code, and mock concrete classes, but we have politely declined. No API can be all things to all people, but jMock contains many useful constructs for testing in general, whether or not you do Need-Driven Development. Therefore, jMock has a layered design: the jMock API is "syntactic sugar" implemented in terms of a core object-oriented framework that can be used to create other testing APIs. A description of these core APIs is beyond the scope of this paper but can be found on the jMock website.

## 6.4 Built To Fail

jMock is designed to produce informative messages so that it is easy to diagnose what caused a test failure. Mock objects are named so that the programmer can easily relate failure messages to the implementation of the test and the target code. The core objects that are composed to specify expectations can provide descriptions that combine to produce a clear failure message.

By default, a mock object is named after the type that it mocks. It is often more useful to use the name to describe the role of that mock within the test. In this case, a mock can be named explicitly by passing the name to the mock's constructor.

```
namedMock = mock(MockedType.class,"namedMock");
```

We have discovered a number of other testing techniques that contribute to good error messages, such as *Self Describing Values* and *Dummy Objects*. A Self Describing Value is one that describes its role in the test when printed as part of an error message. For example, a string that is used as a file name should have value such as `"OPENED-FILE-NAME"` instead of a realistic file name, such as `"invoice.xml"`. A Dummy Object is an object that is passed between objects but not actually invoked during the test. A test uses a dummy object in to specify expectations or assertions that verify that the object under test collaborates correctly with its neighbours. The jMock API includes convenience functions for creating self-describing dummy objects.

```
Timestamp loadTime =
  (Timestamp)newDummy(Timestamp.class,"loadTime");
```

Dummy Objects allow the programmer to defer design decisions about the definition of a type and how the type is instantiated.

# 7. RELATED WORK

Responsibility Driven Design [19] is acknowledged as a useful approach to the design of object oriented software. Need-Driven Development is a technique for doing Responsibility-Driven Design test-first. Mock Objects helps the user discover and design roles and responsibilities from the act of writing tests.

The original `mockobjects.com` library [10] provided a low level library for specifying and verifying expectations in hand written mocks. Having to take time out to create mock implementation of interfaces interrupted the rhythm of the Test Driven Development cycle and resulted in extra work when interfaces changed. To mitigate this, the project provided mock implementations of many of the common JDK and J2EE interfaces. This was impractical to complete and focused on using mock objects for testing rather than as a design tool.

MockMaker [14] automatically generated the source code for mock objects from user defined interface definitions at build time. This encouraged the use of mock objects as a design aid and reduced the interruption to the rhythm of programming. A drawback was that it complicated the build process and the generated mock objects were hard to customise.

EasyMock [5] generates mock objects at runtime through dynamic code generation. It features a “record and playback” style API. Test code defines expected calls by making calls onto a mock object while it is in “record mode” and then puts the mock object into “playback mode” before invoking the object under test. The mock object then verifies that it receives the same calls with the same arguments as those that it recorded. This provided an API that was very easy for new users and that worked well with refactoring tools. However, the simple way of defining expectations often results in over-specified, brittle tests.

DynaMock [13] also generates mock objects at run time. The API is designed to be read as a specification of required behaviour. However, the API is inflexible and hard to extend by users.

Some projects use Aspect Oriented Programming [8] or byte code manipulation to redirect calls on application objects to mock objects during tests. This approach can be useful if you need to test code that runs against inflexible third party APIs. However, this approach is just a testing technique and inhibits useful feedback into the design process.

# 8. FURTHER WORK

Our plans for jMock are to improve the API to work well with automatic refactoring tools and code completion whilst maintaining the flexibility and expressiveness of the current API.

We plan to port jMock to other languages, including C#, and dynamic languages such as Ruby and Python. Much of the development effort of jMock was spent on exploring how to define a usable domain specific language in Java. A design goal of the porting efforts will be to maintain the expressiveness of the API while supporting local language idioms.

An issue with the Mock Objects technique is maintaining and checking consistency between different tests. A test that uses a Mock Object verifies that the object under test makes an expected sequence of outgoing calls. However, this does not test that all objects that use the same interface use that interface in a consistent way, or are consistent with implementers of that interface. We currently address this issue by integration testing and running acceptance tests end-to-end. This catches integration errors but identifying the cause of an error is complicated. We are currently working on an API for testing consistency among clients and implementers of an interface by explicitly describing protocols between objects.

# 9. CONCLUSIONS

Since our previous paper on the topic, we have found that our basic concepts still hold up in daily use across multiple projects. Our understanding of the technique has deepened, in particular we now have a much stronger bias towards using Mock Objects for design, rather than just testing. We now understand its role in driving good design from requirements, and its technical limitations. We have embodied our experience in jMock, a new

generation of Mock Object framework that we believe gives us the expressiveness we need to support Need-Driven Development.

## 10. ACKNOWLEDGMENTS

Our thanks to Martin Fowler, John Fuller, Nick Hines, Dan North, Rebecca Parsons, Imperial College, our colleagues at ThoughtWorks, and members of the eXtreme Tuesday Club.

## 11. REFERENCES

[1] Astels, D. *Test-Driven Development: A Practical Guide*, Prentice-Hall, 2003.

[2] Beck, K. and Cunningham, W. A Laboratory For Teaching Object-Oriented Thinking. In *SIGPLAN Notices (OOPLSA'89)*, *24*, 10, October 1989.

[3] Feathers, M. The "Self-Shunt" Unit Testing Pattern, May 2001. Available at: http://www.objectmentor.com/resources/articles/SelfShunPtrn.pdf

[4] Fowler, M. et al. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Reading, MA, 1999.

[5] Freese, T. EasyMock 2003. Available at: http://www.easymock.org

[6] Hudak, P. *Building domain-specific embedded languages*. ACM Computing Surveys, 28(4es), December 1996.

[7] Hunt, A. and Thomas, D. *Tell, Don't Ask*, May 1998. Available at: http://www.pragmaticprogrammer.com/ppllc/papers/1998_05.html

[8] Kiczales, G., *et al*. Aspect-Oriented Programming, In *proceedings of the European Conference on Object-Oriented Programming (ECOOP),* Finland. Springer-Verlag LNCS 1241. June 1997.

[9] JUnit. 2004. Available at http://www.junit.org

[10] Mackinnon, T., Freeman, S., Craig, P. Endo-testing: unit testing with mock objects. In *Extreme Programming Examined*, Addison-Wesley, Boston, MA. 2001. 287-301.

[11] Marx, K. *Critique of the Gotha Program*, 1874.

[12] Mikhajlov, L. and Sekerinski, E. A Study of the Fragile Base Class Problem. In E. Jul (Ed.), *ECOOP'98 - Object-Oriented Programming 12th European Conference*, Brussels, Belgium, July 1998, pp 355-382, Lecture Notes in Computer Science 1445, Springer-Verlag,1998.

[13] Massol, V. and Husted, T. *JUnit in Action*, Manning, 2003.

[14] Moore, I. and Cooke, M. MockMaker, 2004. Available at: http://www.mockmaker.org

[15] Lieberherr, K. and Holland, I. Assuring Good Style for Object-Oriented Programs *IEEE Software,* September 1989, 38-48.

[16] Poppendieck, M. Principles of Lean Thinking, In *OOPSLA Onward!*, November 2002.

[17] Sun Microsystems, *Java Messaging Service*, Available at: http://java.sun.com/products/jms

[18] Williams, L. and Kessler, R. *Pair Programming Illuminated.* Addison-Wesley, Reading, MA, 2002. ISBN 0201745763.

[19] Wirfs-Brock, R. and McKean, A. *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley, Reading, MA, 2002.