


Friday, March 6. 2009

The Mockery: An Independent Mock Object and Stub Framework for PHP5

Every testing framework supports Stubs and Mock Objects, well...most of them. No wait, only one really does! SimpleTest. Yes, PHPUnit has a whole chapter on the topic - but let's just say it's implementation is more than a little broken.

The side effect of SimpleTest hogging the only true working Mock Object implementation has been an overwhelming de-emphasis in using Mock Objects in PHP. In a world where Unit Testing has given way to Test-Driven Design (TDD), and TDD shows signs of slowly being replaced by, or at a minimum learning from, Behaviour Driven Development (BDD), this lack of Mock Objects just won't do. PHP needs something concrete in the area.

To that end, and right on the back of my previous MutateMe release, I put my considerable and legendary talents (which exist somewhere between my overweening ego, and my dictionary of GTA IV phone numbers) to work to create Mockery. Unfortunately my conscience forces me to acknowledge the invaluable input from Travis Swicegood (who recently released one of my favourite books on git!) who was deeply involved in the discussions and prototype code around Mockery's predecessor, PHPMock, and therefore deserves a lot of blam...er...credit 

You can download the initial PEAR package release of 0.1.0alpha from <http://dev.phpspec.org/Mockery-0.1.0alpha.tgz> or pull from git using the clone URL on the Github.com page at <http://github.com/padraic/mockery>.

Update #1: Support queries and/or comments and suggestions are also welcome to the Mockery mailing list at <http://groups.google.com/group/mockery>.

What Is Mockery?


Mockery is an independent Mock Object and Stub framework. It's not tailored specifically to any one testing framework, rather it's an entirely separate framework with a discrete API. The idea is that you can use this framework within PHPUnit, or SimpleTest, or anything else really, without being forced to rely on the built-in support (if any) that test framework provides.

Mockery was designed specifically to implement a form of Domain Specific Language (DSL). It makes extensive use of a fluent interface mixed with methods to approximate plain English. A mock object expectation could look like:

```
$mock->shouldReceive('create')->atLeast()->once()->atMost()->twice()->withAnyArgs()->andReturn(new stdClass);
```

There's a lot packed into this single statement, and the natural flow of the DSL (there are no nested methods - only a linear fluent interface) is a major component in allowing your unit tests to remain readable. The API grammar itself is actually tiny - it's how you mix and match them that makes things interesting and useful.

Being an independent framework, support for Mockery is non-existent in testing frameworks, but that's not even necessary! Mockery will throw an exception of type `Mockery_ExpectationException` whenever the Mock Object is used in an unexpected fashion. Most testing frameworks natively report uncaught exceptions as


Errors and print the exception's message to the screen - so the only step needed is to ensure all Mocks are validated at the end of a unit test. Naturally, that would be a nice step to automate within testing frameworks but that's not essential (maybe someone will hear my cry in the wilderness and adopt us though ).

What are these Mock Objects and Stubs?

Both Mock Objects and Stubs are variants of a Test Double. A Test Double is basically a fake object used to replace another real object in the test. Consider a simple scenario where a Model has a dependency on a Data Mapper, which in turn has a dependency on a database. You have two broad options:

1. Use a real database which means you are really testing the Model, the Data Mapper, and the DBMS as part of an integration test. This creates problems - databases are slow, need to be managed and reset, and why should the Data Mapper or even the database schema even exist yet?
2. Replace the Data Mapper with a fake - a Stub or Mock Object which implements an identical interface, but returns canned predictable responses with a negligible performance hit.

Many people will identify with the first option - there are enough arguments over whether to use a real database when testing to realise it's a common approach. Fewer will identify with the second, since it has normally (outside of SimpleTest) involved creating special subclasses or mini-stubs.

The purpose of Mockery is not to pass judgement on any practice however - that's my personal domain .

Mockery supports both Mock Objects and Stubs as equal citizens.

The difference between Mocks and Stubs is easier to grasp. A Stub is a static object - it's created to return canned responses to all method calls. There's nothing more complicated to learn - it's that simple.

A Mock Object is a more interactive agent - it's dynamic since you can literally program it to expect specific methods, with a specific order or number of calls, and return canned data depending on the preceding conditions. Essentially - it allows you to clearly define what interactions you expect the Mock Object to experience during a test. And then validate the Mock to see if your expectations were correct (pass) or incorrect (fail!). Yep - Mock Objects carry verifiable assertions about what should happen in a test to the Mock.

This side of Mock Objects, asserting its expected behaviour, let's you design objects in isolation by Mocking their dependencies - even if those dependencies do not yet exist. This has led their charge into mainstream TDD and BDD as a form of API exploration. As you add Mocks, you are basically designing a class's interface to its dependencies. The concept of Mock Objects as a design tool is now fairly common if a relatively new idea.

Stubbing With Mockery

Mockery support Stubs in a fairly simple fashion. I remember Travis and I had this argument so many times I lost count :), so I divorced the idea of a Stub from a Mock to give Stubs a separate and ultra-simple

implementation.

As an aside, Mockery offer two primary starting methods:

1. Mockery::mock()
2. mockery()

Both are interchangeable. For simplicity, and because Travis sold me on the idea, I prefer the function method. The function method is also more anonymous so you're not constantly creating Stubs using a mock() method (to the confusion of whoever reads your tests). To create a Stub, simply call mockery() and provide it with a class name you want to create, and an array of public methods and their return values. Obviously, the class name cannot exist - if the class does already exist you'll actually receive a Mock Object (but luckily that too will be confined to a Stub so you won't notice anything different so it still qualifies as a Stub!):

```
$stub = mockery('Foo', array('print'=>'I am Foo!'));
```

The new object acts just like a class defined as:


```
class Foo {  
    public function print() {  
        return 'I am Foo!';  
    }  
}
```

Either way, the new object will have one thing for certain - it will be of type Foo, sufficient to pass any class hinting you use in your source code. You can set the configuration method array to return any scalar value or object you want, including other Stubs or even Mock Objects if you're really feeling fiendishly clever.

With the Stub created, you can proceed to pass it into other objects expecting the interface and return values you configured for that test.

Mock Objects with Mockery

Since Mock Objects are more interactive, they offer a broader API. Like Stubs, they always inherit the class type of the class being mocked, and you can also set return values. Unlike Stubs, you can also set expectations!

So one day, this guy who looks like a Klingon, walks into a web development shop. Sounds like a bad joke, eh? 

. He wants us to design a Ship Control System but gets nervous when asked about Weapons, Propulsion and Power. Left without a clue about these additional system, he hands over some interface requirements to stick to, and we start with one preliminary test (applying TDD).

```
class BirdOfPreyTest extends PHPUnit_Framework_TestCase {  
    public function testWarpFactor8() {  
        $power = mockery('KlingonPower');  
        $ship = new KlingonBOP($power);  
    }  
}
```

```

// set Mock Object expectations
$power->shouldReceive('checkForWarp')->once()->with(8)->andReturn(true);
$power->shouldReceive('deduct')->with(80)->once();
// perform action required
$success = $ship->gotoWarp(8);
// add any necessary assertions?
$this->assertTrue($success);
// verify mock expectations
mockery_verify();
}
}

```

Since the class KlingonPower doesn't really exist, we use a Mock Object to figure out how interactions should occur. From those interactions, as the example suggests, it's a simple matter to actually go and implement the real thing! But since this test focuses on the Bird Of Prey control system, no need to run off yet. We don't have full specs for a Power system, so we continue using Mocks to explore what its API could be.

Let's implement the KlingonBOP class to make this test pass:

```

class KlingonBOP {
    protected $power = null;
    public function __construct(KlingonPower $power) {
        $this->power = $power;
    }
    public function gotoWarp($factor) {
        if ($this->power->checkForWarp($factor)) {
            $this->power->deduct($factor * 10);
            return true;
        }
        return false;
    }
}

```

As you can see, using the Mock Object guided us more clearly towards the API of a possible Power system class without requiring we actually implement it. A Stub would also have fit, but would not have provided the API behaviour and expectations we used when implementing the KlingonBOP class.

As for feedback on failures, here some example output from PHPUnit as an example. Reporting details should be improved in future iterations.

```

PHPUnit 3.3.14 by Sebastian Bergmann.
.E
Time: 0 seconds
There was 1 error:
1) testAddWithFailingMock(AdditionTest)
Mockery_ExpectationException: method get() called incorrect number of times;
expected call 2 times but received 1

```

This is a really simple example, but hopefully it offers a small taste for the usefulness of both Mock Objects and Stubs.

The Mockery API

Here's the run down of the current starting points to create a Stub or Mock Object:

```
mockery( 'Foo' );
```

Create a new Mock Object with the class name and class type of 'Foo'.

```
mockery( 'Foo', array( 'set'=>null, 'get'=>foo' ) );
```

Create a new Stub with the class name and type of 'Foo' defining two public methods and their return values.

```
mockery( 'Existing_Class' );
```

Create a new Mock Object with a unique class name but a class type of 'Existing_Class'. The unique class name is unimportant, but used to generate a Mock which is a subclass of any existing Class, Abstract Class or Interface.

```
mockery( 'Existing_Class', 'Custom_Name' );
```

Create a new Mock Object with a class name of 'Custom_Name' but a class type of 'Existing_Class'.

```
mockery( 'Existing_Class', array( 'set'=>null, 'get'=>foo' ) );
```

Create a new Stub with a unique class name but class type of 'Existing_Class' which effectively acts as a Stub (internally this is a variation on a Mock Object but that detail isn't that important).

Additional tweaks and mixes of options will follow in the next release shortly.

```
mockery_verify()
```

Verify all currently unverified Mock Objects - this also exists as a separate method on every Mock Object for more granularity if needed.

The Mockery Mock Object Expectations API

Every Mock Object expectation starts with the `shouldReceive()` method. This is also the only non-namespaced method added to all objects, and is therefore a reserved method name. Other methods added dynamically to Mock Objects (or Stubs) are prefixed with "mockery_" to minimise the contamination of the object space with methods you might collide with one day.

A `shouldReceive()` call returns an instance of `Mockery_Expectations` which accepts expectations for the given method via a fluent interface, i.e. sequential method calls. These methods include:

```
never()
```

Expect the method to never be called. Same as `times(0)`.

```
zeroOrMoreTimes()
```

Expect the method to be called zero or more times.

```
once()
```

Expect the method to be called exactly once. Same as `times(1)`.

```
twice()
```

Expect the method to be called exactly twice. Same as `times(2)`.

```
times(x)
```

Expect the method to be called x times.

`atLeast()`

Set a minimum of times to expect the method to be called by appending one of `once()`, `twice()`, or `times()`.

`atMost()`

Set a maximum of times to expect the method to be called by appending one of `once()`, `twice()`, or `times()`.

`with(x[, y...])`

Expect the list of arguments to be passed to the method.

`withAnyArgs()`

Expect any arguments including no arguments to be passed to the method.

`withNoArgs()`

Expect no arguments to be passed to the method.

`withArgsMatching(x[, y...])`

Expect arguments which, when cast to `String`, match the given list of Regular Expressions.

`andReturn(x[, y...])`

Set a return value to be returned from the method call. Multiple parameters set an ordered sequence of returns. In both cases, the last listed value is returned indefinitely to all subsequent calls once any preceding listed values are used.

`andThrow(exception[, message])`

Set an Exception class to throw including an optional message.


```
ordered( )
```

Set an expectation method to be called in a specific order. Each use of ordered() on a Mock Object determines the order, but any method expectations omitting this call remain free to be called at any time.

Conclusion

This is an initial alpha release, so while its hopefully sufficiently feature complete to be useful, its main objective is to illicit feedback! If you want new features, API changes, fixes, or have any ideas or comments whatsoever - please post a comment. You can also fork the git repository on Github.com if you want to make fixes so I can pull them back into my own repo for later distribution in a stable release.

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 00:38

Really cool! Just getting into TDD, and have been having a hard time figuring out how to test things properly. Mocks make a lot of sense now 

Anonymous on Mar 6 2009, 02:20


Because of minimal current support for mocking, I'm an advocate of self-shunting.

Like:


```
class BirdOfPreyTest extends PHPUnit_Framework_TestCase implements KlingonPowerInterface {  
/ ... /
```

I think an easy solution for mocking will reduce the need for doing this. Anonymous on Mar 6 2009, 09:19


> He wants us to design a Ship Control
> System but gets nervous when asked
> about Weapons, Propulsion and Power.

Wow, Ship Control Systems can be built with PHP? Well, that'll certainly fill my weekends. 

Anonymous on Mar 6 2009, 09:20

The funny thing is that I abandoned SimpleTest because of its flawed Mock objects implementation 

. Back then generated classes did not extend original ones, so mock could not be passed to a typehinted method. Anonymous on Mar 6 2009, 09:41

Sure, will come in useful if you want a job at NASA in the next decade 

. If Klingons use PHP, surely NASA will! 

Anonymous on Mar 6

2009, 12:08

You can rest assured that problem doesn't exist in Mockery - you can mock any class, abstract class or interface, and it's type will be preserved by the Mock Object's generated class through inheritance. This also applies to Stubs (which is why stubbing an existing class creates a Stub-constrained Mock Object - an internals technicality which merely exists to ensure Stubs have an accurate type too). Anonymous on Mar 6 2009, 12:11

My comment was more about you praising SimpleTest's mock implementation and saying PHPUnit's one was inferior to it. By the way, in what way do you think PHPUnit mocks are broken? And, for broader approach, could you post a comparison table maybe between PHPUnit, SimpleTest and Mockery? Anonymous on Mar 6 2009, 17:16

I tried using PHPUnit's mocks a few days ago (3.3.14 I think is my version of PHPUnit) - I'll try to be more clear about the problems in a better note later, but it was like trying to shove a round peg into a square hole - it seemed to make sense, but little worked as expected. I think one of its main problems is that the API doesn't connect linearly in the same way as Mockery or even SimpleTest - the nested method approach causes problems.

I'm being vague because I can't explain it right offhand



. I took to shunting in PHPUnit instead. Anonymous on Mar 6 2009, 17:42

Usually it's as easy as:

```
$mock = $this->getMock('ClassName');
$mock->expects($this->once())
->method('getSomething')
->with($this->equalTo('a'))
->will($this->throwException(new Exception('wrong!')));
```

Before returnCallback() was introduced, some hacks were required to get conditional returns working, but it shouldn't be an issue anymore.

Mocking a static class is a bit harder and requires some support from SUT, but no testing frameworks support mocking a static anyway. Anonymous on Mar 6 2009, 20:38

The main problem I see with PHPUnit mocks is that the defaults when constructing them are bad. They will do unintuitive things like run the original object's constructor. Fortunately, that behavior can be turned off. Anonymous on Mar 6 2009, 21:55

Bruce, I'm afraid I can't give you a concrete example because it was quite some time since I ran screaming from PHPUnit.

The basic problem, alluded to by Padraic's "trying to shove a square peg into a round hole", is that PHPUnit's Mock implementation is very sensitive to the number and sequence of calls to the mocked method. One has to define them very carefully or the test will fail. But more than half the time, I couldn't care less what the sequence of calls is, I just want to have a mocked method return 'bar' whenever it receives 'foo' as an argument. If any other arguments were made, with PHPUnit I had to explicitly define them and say in what order they should occur. The code for setting up some Mocks was longer than the code I wanted to test and I still couldn't get it to do everything I wanted.

Simpletest's setReturnValue() and setReturnValueAt() methods are so much simpler. Anonymous on Mar 7 2009, 13:45

Thanks David for expanding on the problem I had trouble explaining myself - as you can guess I'm not a fan and my avoidance of PHPUnit's Mocks has left me forgetful of the problems that made me avoid it.

I'm hoping Mockery does its bit in the space, and avoids some the problems noted by everyone above, even if it only ends up being a reaction by the frameworks to finally fix their problems so PHP can get out of its rut and adopt Mock Objects/Stubs with more enthusiasm. Anonymous on Mar 7 2009, 18:36

I guess I'm not the only one to experience this and I hope what I said more or less sums it up for you, Padraic. I decided that, since Simpletest's continued development seemed to have stalled and PHPUnit seemed to be gaining currency in the wider community, I really should take a serious look at it. What really surprised me at the time was that I hit the limitations within a couple of days of starting using it.

I had more than a few WTF moments in that short time!


I mentioned this to Sebastian Bergmann when he was in Melbourne and he expressed some (what I took to be) tongue-in-cheek contempt for my position. I would have liked to talk to him about it a bit more but I'm given to understand that he's not responsible for the Mock objects implementation. I'm also not a developer by trade and left it to the serious chaps to do the talking! Anonymous on Mar 8 2009, 09:56

To be fair to Sebastian, I'm pretty sure he wants to get Mock Objects improved in PHPUnit and since it is outside his circle of development responsibility (or was when it was added), I don't blame him for its shortcomings - problems in PHPUnit are largely the community's fault - it's an open source project and there is an opportunity for anyone to dig into PHPUnit to fix things.

Main reason I took the Mockery route was to improve the situation, but without the failings (I also never liked the nested style API and I needed to add additional features like the upcoming Mock Object Recording and reusable defaults) or limitations of the original.

Also I'm a proponent of the "GNU Way" - smaller specialised components which can be mixed and matched add more value than tightly integrated tools. Since I use both PHPUnit and SimpleTest (and #5 on my project list - PHPSpec), integrated tools offer me zero value - I need cross framework tools that can be used anywhere by anyone without specific knowledge needed for each frameworks individual featureset. Anonymous on Mar 8 2009, 14:40

It is good to see someone working on a new mock object implementation again. Are you planning on adding more exhaustive parameter matching? It seems having only equality and regex would be pretty limiting.


Also, (and I say this "tongue in cheek") your example has multiple mock assertions...I thought you liked one assertion per test 

Anonymous on Mar 8 2009, 19:27


Current release is alpha so there is a lot of smaller features (and some bigger!) to add including:

- recordable mock objects
- overridable defaults
- broader equality for arrays/objects
- support for "partial" mocks


Those are the ones most needed before a stable release.

Assertions don't apply to Mocks . They are replacement/exploratory tools before you get to the assertion itself (practice with mocks has a few uses beyond just verifying). Anonymous on Mar 8 2009, 23:41

[quote]
I just want to have a mocked method return 'bar' whenever it receives 'foo' as an argument
[/quote]

You talked to the right person then 
I have had this problem myself and here's the solution: <http://weirdan.livejournal.com/69776.html>

PS: PÃ¼draic, you have absolutely awful comment system here. Anonymous on Mar 9 2009, 01:31

Complain to the S9Y developers? 
Sorry, but with around 350 posts archived here from years of blogging it's difficult to migrate all that. It's a project for the future I intend doing. Anonymous on Mar 9 2009, 10:54