Phake - PHP Mocking Framework

Mike Lively

Phake - PHP Mocking Framework

Mike Lively

Copyright © 2011 Mike Lively

Table of Contents

List of Examples

Introduction to Phake

Phake is a mocking framework for PHP. It allows for the creation of objects that mimic real object in a predictable and controlled manner. This allows you to treat external method calls made by your system under test (SUT) as just another form of input to your SUT and output from your SUT. This is done by stubbing methods that supply indirect input into your test and by verifying parameters to methods that receive indirect output from your test.

In true Las Vegas spirit I am implementing a new framework that allows you to easily create new card games. Most every card game at one point or another needs a dealer. I have created a new class called CardGame that implements the basic functionality for a card game. This example is seen in Example 1, “CardGame class”.

Example 1. CardGame class

```
<?php

class CardGame
{
 private $dealerStrategy;
 private $deck;
 private $players;

 public function CardGame(DealerStrategy $dealerStrategy, CardCollection $deck, Pl
 {
  $this->dealerStrategy = $dealerStrategy;
  $this->deck = $deck;
  $this->players = $players;
 }

 public function dealCards()
 {
  $this->deck->shuffle();
  $this->dealerStrategy->deal($deck, $players);
 }
}

?>
```

If I want to create a new test to ensure that dealCards() works properly, what do I need to test? Everything I read about testing says that I need to establish known input for my test, and then test its output. However, in this case, I don’t have any parameters that are passed into dealCards() nor do I have any return values I can check. I could just run the dealCards() method and make sure I don’t get any errors or exceptions, but that proves little more than my method isn’t blowing up spectacularly. It is apparent that I need to ensure that what I actually need to assert is that the shuffle() and deal() methods are being called. If I want to continue testing this using concrete classes that already exist in me system, I could conjure up one of my implementations of DealerStrategy, CardCollection and PlayerCollection. All of those objects are closer to being true value objects with a testable state. I could feasibly construct instances of those objects, pass them into an instance of CardGame, call dealCards() and then assert the state of those same objects. A test doing this might look something like Example 2, “CardGameTest1 Unit Test”.

Example 2. CardGameTest1 Unit Test

```
<?php

class CardGameTest1 extends PHPUnit_Framework_TestCase
{
 public function testDealCards()
 {
  $dealer = new FiveCardPokerDealer();
  $deck = new StandardDeck();
  $player1 = new Player();
  $player2 = new Player();
  $player3 = new Player();
  $player4 = new Player();
  $players = new PlayerCollection(array($player1, $player2, $player3, $player4);

  $cardGame = new CardGame($dealer, $deck, $players);
  $cardGame->dealCards();

  $this->assertEquals(5, count($player1->getCards()));
  $this->assertEquals(5, count($player2->getCards()));
  $this->assertEquals(5, count($player3->getCards()));
  $this->assertEquals(5, count($player4->getCards()));
 }
}

?>
```

This test isn’t all that bad, it’s not difficult to understand and it does make sure that cards are dealt through making sure that each player has 5 cards. There are at least two significant problems with this test however. The first problem is that there is not any isolation of the SUT which in this case is dealCards(). If something is broken in the FiveCardPokerDealer class, the Player class, or the PlayerCollection class, it will manifest itself here as a broken CardGame class. Thinking about how each of these classes might be implemented, one could easily make the argument that this really tests the FiveCardPokerDealer class much more than the dealCards() method. The second problem is significantly more problematic. It is perfectly feasible that I could remove the call to $this->deck->shuffle() in my SUT and the test I have created will still test just fine. In order to solidify my test I need to introduce logic to ensure that the deck has been shuffled. With the current mindset of using real objects in my tests I could wind up with incredibly complicated logic. I could feasibly add an identifier of some sort to DealerStrategy::shuffle() to mark the deck as shuffled thereby making it checkable state, however that makes my design more fragile as I would have to ensure that identifier was set probably on every implementation of shuffle().

This is the type of problem that mock object frameworks solve. A mock framework such as Phake can be used to create implementations of my DealerStrategy, CardCollection, and PlayerCollection classes. I can then exercise my SUT. Finally, I can verify that the methods that should be called on these objects were called correctly. An example of this usage of Phake would alter the above test case to look like Example 3, “CardGameTest2 Unit Test”.

Example 3. CardGameTest2 Unit Test

```
<?php

class CardGameTest2 extends PHPUnit_Framework_TestCase
{
 public function testDealCards()
 {
  $dealer = Phake::mock('DealerStrategy');
  $deck = Phake::mock('CardCollection');
  $players = Phake::mock('PlayerCollection');

  $cardGame = new CardGame($dealer, $deck, $players);
  $cardGame->dealCards();

  Phake::verify($deck)->shuffle();
  Phake::verify($dealer)->deal($deck, $players);
 }
}

?>
```

There are three benefits of using mock objects that can be seen through this example. The first is the brittleness of the fixture is reduced. In our previous example you see that I have to construct a full object graph based on the dependencies of all of the classes involved. I am fortunate in the first example that there are only 4 classes involved. In real world problems and especially long lived, legacy code the object graphs can be much, much larger. When using mock objects you typically only have to worry about the direct dependencies of your SUT. Specifically, direct dependencies required to instantiate the dependencies of the class under test, the parameters passed to the method under test (direct dependencies,) and the values returned by additional method calls within the method under test (indirect dependencies.)

The second benefit is the test is only testing the SUT. If this test fails due to a change in anything but the interfaces of the classes involved, the change would have had to been made in either the constructor of CardGame, or the dealCards() method itself. Obviously, if an interface change is made (such as removing the shuffle()) method, then I would have a scenario where the changed code is outside of this class. However, provided the removal of that method was intentional, I will know that this code needs to be addressed as it is depending on a method that no longer exists.

The third benefit is that I have truer verification and assertions of the outcome of exercising my SUT. In this case for instance, I can be sure that if the call to shuffle() is removed, this test will fail. It also does it in a way that keeps the code necessary to assert your final state simple and concise. This makes my test overall much easier to understand and maintain. There is still one flaw with this example however. There is nothing here to ensure that shuffle() is called before deal() it is quite possible for someone to mistakenly reverse the order of these two calls. The Phake framework does have the ability to track call order to make this test even more bullet proof via the Phake::inOrder() method. I will go over this in more detail later. In the next chapter I will go over how you can install Phake and show you some of the basic usage.

Chapter 1. Getting Started

Phake depends on PHP 5.2 or greater. It has no dependency on PHPUnit and should be usable with any version of PHPUnit so long as the PHP version is 5.2 or greater.

Phake can be installed via the digitalsandwich pear channel.

Example 1.1. Installing Phake Via Pear

```
pear channel-discover pear.digitalsandwich.com
pear install digitalsandwich/Phake
```

Once Phake is installed, so long as the pear directory (usually something like /usr/share/php) is in your include_path, you can simply include “Phake.php” in your code and you will be able to utilize the functionality of Phake.

For those that you like to live more on the edge, you can also clone a copy of Phake from the Phake GitHub repository at git://github.com/mlively/Phake.git. Every attempt is made to keep the master branch stable and this should be usable for those that immediately need features before they get released or in the event that you enjoy the bleeding edge. Always remember, until something goes into a rc state, there is always a chance that the functionality may change. However as an early adopter that uses GitHub, you can have a chance to mold the software as it is built.

Chapter 2. Method Verification

Basic Method Verification

Method call verification (or mocking) is accomplished via the Phake::verify() method. Phake::verify() accepts a mock object generated from Phake::mock() as its first object. The most basic way to use this method can be seen below.

Example 2.1. Simple Verification

```
<?php
class PhakeTest1 extends PHPUnit_Framework_TestCase
{
  public function testBasicVerify()
  {
    $mock = Phake::mock('MyClass');

    $mock->foo();

    Phake::verify($mock)->foo();
  }
}
?>
```

The Phake::verify() call here, verifies that the method foo() has been called once (and only once) with no parameters on the object $mock. A very important thing to note here that is a departure from most (if not all) other PHP mocking frameworks is that you want to verify the method call AFTER the method call takes place. Other mocking frameworks such as the one built into PHPUnit depend on you setting the expectations of what will get called prior to running the system under test.

Phake strives to allow you to follow the four phases of a unit test as layed out in xUnit Test Patterns: setup, exercise, verify, and teardown. The setup phase of a test using Phake for mocking will now include calls to Phake::mock() for each class you want to mock. The exercise portion of your code will remain the same. The verify section of your code will include calls to Phake::verify(). The exercise and teardown phases will remain unchanged.

Verifying Method Parameters

Verifying method parameters using Phake is very simple yet can be very flexible. There are a wealth of options for matching parameters that is discussed later on in Method Parameter Matchers.

Verifying Multiple Invocations

A common need for mock objects is the ability to variable multiple invocations on that object. Phake allows you to use Phake::verify() multiple times on the same object. A notable difference between Phake and PHPUnit's mocking framework is the ability to mock multiple invocations of the same method with no regard for call sequences. The PHPUnit mocking test below would fail for this reason.

Example 2.2. Multiple Invocations With PHPUnit Mocks - Bad Example

```
<?php
class MyTest extends PHPUnit_Framework_TestCase
{
  public function testPHPUnitMock()
  {
    $mock = $this->getMock('PhakeTest_MockedClass');

    $mock->expects($this->once())->method('fooWithArgument')
            ->with('foo');

    $mock->expects($this->once())->method('fooWithArgument')
            ->with('bar');

    $mock->fooWithArgument('foo');
    $mock->fooWithArgument('bar');
  }
}
?>
```

The reason this test fails is because by default PHPUnit only allows a singl expectation per method. The way you can fix this is by using the at() matcher. This allows you to specify the index of the invocation you want to match again. So to make the test above work you would have to change it.

Example 2.3. Multiple Invocations With PHPUnit Mocks - Good Example

```
<?php
class MyTest extends PHPUnit_Framework_TestCase
{
  public function testPHPUnitMock()
  {
    $mock = $this->getMock('PhakeTest_MockedClass');

    //NOTICE this is now at() instead of once()
    $mock->expects($this->at(0))->method('fooWithArgument')
            ->with('foo');

    //NOTICE this is now at() instead of once()
    $mock->expects($this->at(1))->method('fooWithArgument')
            ->with('bar');

    $mock->fooWithArgument('foo');
    $mock->fooWithArgument('bar');
  }
}
?>
```

This test will now run as expected. There is still one small problem however and that is that you are now testing not just the invocations but also the order of invocations. Many times the order in which two calls are made really do not matter. If swapping the order of two method calls will not break your application then there is no reason to enforce that code structure through a unit test. Unfortunately, you cannot have multiple invocations of a method in PHPUnit without enforcing call order. In Phake these two notions of call order and multiple invocations are kept completely distinct. Here is the same test written using Phake.

Example 2.4. Multiple Invocations With Phake

```
<?php
class MyTest extends PHPUnit_Framework_TestCase
{
  public function testPHPUnitMock()
  {
    $mock = Phake::mock('PhakeTest_MockedClass');

    $mock->fooWithArgument('foo');
    $mock->fooWithArgument('bar');

    Phake::verify($mock)->fooWithArgument('foo');
    Phake::verify($mock)->fooWithArgument('bar');
  }
}
?>
```

You can switch the calls around in this example as much as you like and the test will still pass. You can mock as many different invocations of the same method as you need.

If you would like to verify the exact same parameters are used on a method multiple times (or they all match the same constraints multiple times) then you can use the verification mode parameter of Phake::verify(). The second parameter to Phake::verify() allows you to specify how many times you expect that method to be called with matching parameters. If no value is specified then the default of one is used. The other options are:

1. Phake::times($n) – Where $n equals the exact number of times you expect the method to be called.
2. Phake::atLeast($n) – Where $n is the minimum number of times you expect the method to be called.
3. Phake::atMost($n) – Where $n is the most number of times you would expect the method to be called.

Here is an example of this in action.

Example 2.5. Multiple Invocations of the Same Call With Phake

```
<?php
class MyTest extends PHPUnit_Framework_TestCase
{
  public function testPHPUnitMock()
  {
    $mock = Phake::mock('PhakeTest_MockedClass');

    $mock->fooWithArgument('foo');
    $mock->fooWithArgument('foo');

    Phake::verify($mock, Phake::times(2))->fooWithArgument('foo');
  }
}
?>
```

Verifying Calls Happen in a Particular Order

Sometimes the desired behavior is that you verify calls happen in a particular order. Say there is a functional reason for the two variants of fooWithArgument() to be called in the order of the original test. You can utilize Phake::inOrder() to ensure the order of your call invocations. Phake::inOrder() takes one or more arguments and errors out in the event that one of the verified calls was invoked out of order. The calls don't have to be in exact sequential order, there can be other calls in between, it just ensures the specified calls themselves are called in order relative to each other. Below is an example Phake test that behaves similarly to the PHPUnit test that utilized at().

Example 2.6. Verifying Multiple Calls in Order

```
<?php
class MyTest extends PHPUnit_Framework_TestCase
{
  public function testPHPUnitMock()
  {
    $mock = Phake::mock('PhakeTest_MockedClass');

    $mock->fooWithArgument('foo');
    $mock->fooWithArgument('bar');

    Phake::inOrder(
      Phake::verify($mock)->fooWithArgument('foo'),
      Phake::verify($mock)->fooWithArgument('bar')
    );
  }
}
?>
```

Verifying No Interaction with a Mock so Far

Occasionally you may want to ensure that no interactions have occurred with a mock object. This can be done by passing your mock object to Phake::verifyNoInteraction($mock). This will not prevent further interaction with your mock, it will simply tell you whether or not any interaction up to that point has happened.

Verifying No Further Interaction with a Mock

There is a similar method to prevent any future interaction with a mock. This can be done by passing a mock object to Phake::verifyNoFurtherInteraction($mock).

Verifying Magic Methods

Magic methods are commonly used in PHP and the need to be able to seemlessly utilize them is always necessary. Most magic methods can be verified using the method name just like you would any other method. The one exception to this is the __call() method. This method is overwritten on each mock already to allow for the fluent api that Phake utilizes. If you want to verify a particular invoaction of __call() you can verify the actual method call by mocking the method passed in as the first parameter.

Consider the following class.

Example 2.7. A Magic Class

```
<?php
class MagicClass
{
public function __call($method, $args)
{
return '__call';
}
}
?>
```

You could mock an invocation of the __call() method through a userspace call to magicCall() with the following code.

Example 2.8. Implicitly Verify __call()

```
<?php
class MagicClassTest extends PHPUnit_Framework_TestCase
{
  public function testMagicCall()
  {
    $mock = Phake::mock('MagicClass');

    $mock->magicCall();

    Phake::verify($mock)->magicCall();
  }
}
?>
```

If for any reason you need to explicitely verify calls to __call() then you can use Phake::verifyCallMethodWith().

Example 2.9. Explicitly Verify __call()

```
<?php
class MagicClassTest extends PHPUnit_Framework_TestCase
{
  public function testMagicCall()
  {
    $mock = Phake::mock('MagicClass');

    $mock->magicCall();

    Phake::verifyCallMethodWith('magicCall')->isCalledOn($mock);
  }
}
?>
```

This last section is purposely vague as this method of explicitly verifying statics will likely be replaced in the very near future.

Chapter 3. Method Stubbing

Basic Method Stubbing

To Phake::when() method is used to stub methods in Phake. As discussed in the introduction, stubbing allows an object method to be forced to return a particular value given a set of parameters. Similarly to Phake::verify(), Phake::when() accepts a mock object generated from Phake::mock() as its first parameter. A basic example of stubbing can be found below.

Example 3.1. Simple Stubbing

```
<?php
class PhakeTest extends PHPUnit_Framework_TestCase
{
 public function testSimpleStub()
 {
  $mock = Phake::mock('PhakeTest_MockedClass');

  Phake::when($mock)->foo()->thenReturn(42);

  $this->assertEquals(42, $mock->foo());
 }
}
?>
```

The Phake::when() call here ensures that whenever the method foo() is called with no parameters on the object $mock the string ‘bar’ will be returned. If the method you are stubbing normally accepts parameters you can use any of the Method Parameter Matchers that you would also use for Phake::verify(). The various method matchers are discussed later on in Method Parameter Matchers.

Whenever a mock object is created using Phake::mock() all methods by default will simply return null. The reasoning behind this is that generally speaking, each method you test should depend on only what it needs to perform the (hopefully one) responsibility assigned to it. Normally you will have very controlled delegation to other objects. To help with localization of errors in your test it is assumed that you will always want to mock external dependencies to keep them from influencing the results of unit tests dedicated to the behavior of other parts of the system. Another reason for this default behavior is that it provides consistent and predictable behavior regardless of whether you are testing concrete classes, abstract classes, or interfaces. It should be noted that this default behavior for concrete methods in classes is different then the default behavior in PHPUnit. In PHPUnit, you have to explicitly indicate that you are mocking a method, otherwise it will call the actual method code. There are certainly cases where this is useful and this behavior can be achieved in Phake. I will discuss this aspect of Phake later in this chapter.

Overwriting Existing Stubs

Users of Phake will often setup stubs as a part of a common test fixture. (setUp() in PHPUnit.) This helps reduce duplicate code which in turn makes your tests more maintainable. However, it often happens that for all but one test, the stubbing needs to be the same, and then you have a single test that is reponsible for an error condition of some sort. Phake allows you to redefine a previously defined stub. The way that stubbing works internally in Phake is that each stub defined with Phake::when() is kept in what is essentially an array. Then, when a method call is made, this array is iterated through in reverse order and the first stub that matches the current method invocation is used to provide the answer to that call. So, by virtue of redefining a stub using the same parameters, the new stub will always be the first to match. An example of how this works is shown below.

Example 3.2. Redefining Stubs

```
<?php
class PhakeTest extends PHPUnit_Framework_TestCase
{
 public function testRedefineStub()
 {
  $mock = Phake::mock('PhakeTest_MockedClass');

  Phake::when($mock)->foo()->thenReturn(24);
  Phake::when($mock)->foo()->thenReturn(42);

  $this->assertEquals(42, $mock->foo());
 }
}
?>
```

Stubbing Multiple Calls

It is very easy to stub multiple calls to the same method in Phake. If the various calls to a method that are being stubbed utilize different parameters in each call then you can simply create multiple stubs using Phake::when(). An example of this can be seen below.

Example 3.3. Multiple Stubs

```
<?php
class PhakeTest extends PHPUnit_Framework_TestCase
{
 public function testMultipleStubs()
 {
  $mock = Phake::mock('PhakeTest_MockedClass');

  Phake::when($mock)->foo()->thenReturn(24);
  Phake::when($mock)->fooWithReturnValue()->thenReturn(42);

  $this->assertEquals(24, $mock->foo());
  $this->assertEquals(42, $mock->fooWithReturnValue());
 }
}
?>
```

In the above example, I have stubbed two variations of the add() method. The first stub will return 4 when add() is called with the parameters 2 and 2. The second stub will return 8 when add() is called with the parameters 3 and 5.

Stubbing Consecutive Calls

Occasionally you may also want to stub multiple calls to the same method. A very common case for this is a test that involves iterators. When you step through an iterator, you normally expect multiple calls to Iterator::next() to return different elements. To achieve a different result on each call to a stub you can chain the answer portion of Phake::when() together. So your stubbing may start to look like this: Phake::when($itr)->thenReturn(1)->thenReturn(2)->thenReturn(3).

Below is an example that utilizes this functionality to test a custom function named iterator_sum() which is supposed to take all of the values in an iterator and return their sum.

Example 3.4. Multiple Stubs

```
<?php
class PhakeTest extends PHPUnit_Framework_TestCase
{
 public function testConsecutiveCalls()
 {
  $mock = Phake::mock('PhakeTest_MockedClass');

  Phake::when($mock)->foo()->thenReturn(24)->thenReturn(42);

  $this->assertEquals(24, $mock->foo());
  $this->assertEquals(42, $mock->foo());
 }
}
?>
```

Answers

In all of the examples so far, the thenReturn() answer is being used. There are other answers that are remarkably useful writing your tests.

Throwing Exceptions

Exception handling is a common aspect of most object oriented systems that should be tested. The key to being able to test your exception handling is to be able to control the throwing of your exceptions. Phake allows this using the thenThrow() answer. This answer allows you to throw a specific exception from any mocked method. Below is an example of a piece of code that catches an exception from the method foo() and then logs a message with the exception message.

Example 3.5. A class with exception logging

```
<?php
class MyClass
{
 private $logger;

 public function __construct(LOGGER $logger)
 {
  $this->logger = $logger;
 }

 public function processSomeData(MyDataProcessor $processor, MyData $data)
 {
  try
  {
   $processor->process($data);
  }
  catch (Exception $e)
  {
   $this->logger->log($e->getMessage());
  }
 }
}
?>
```

In order to test this we must mock foo() so that it throws an exception when it is called. Then we can verify that log() is called with the appropriate message.

Example 3.6. Using thenThrow() to Throw Exceptions

```
<?php
class MyClassTest extends PHPUnit_Framework_TestCase
{
 public function testProcessSomeDataLogsExceptions()
 {
  $logger = Phake::mock('LOGGER');
  $data = Phake::mock('MyData');
  $processor = Phake::mock('MyDataProcessor');

  Phake::when($processor)->process($data)->thenThrow(new Exception('My error messa

  $sut = new MyClass($logger);
  $sut->processSomeData($processor, $data);

  //This comes from the exception we created above
  Phake::verify($logger)->log('My error message!');
 }
}
?>
```

Calling the Parent

Phake provides the ability to allow calling the actual method of an object on a method by method basis by using the thenCallParent() answer. This will result in the actual method being called. Consider the following class.

Example 3.7. A Test Class

```
<?php
class MyClass
{
 public function foo()
 {
  return '42';
 }
}
?>
```

The thenCallParent() answer can be used here to ensure that the actual method in the class is called resulting in the value 42 being returned from calls to that mocked method.

Example 3.8. Using thenCallParent() to Force a Stub to Call its Parent

```
<?php
class MyClassTest extends PHPUnit_Framework_TestCase
{
 public function testCallingParent()
 {
  $mock = Phake::mock('MyClass');
  Phake::when($mock)->foo()->thenCallParent();

  $this->assertEquals(42, $mock->foo());
 }
}
?>
```

Please avoid using this answer as much as possible especially when testing newly written code. If you find yourself requiring a class to be only partially mocked then that is a code smell for a class that is likely doing too much. An example of when this is being done is why you are testing a class that has a singular method that has a lot of side effects that you want to mock while you allow the other methods to be called as normal. In this case that method that you are desiring to mock should belong to a completely separate class. It is obvious by the very fact that you are able to mock it without needing to mock other messages that it performs a different function.

Even though partial mocking should be avoided with new code, it is often very necessary to allow creating tests while refactoring legacy code, tests involving 3rd party code that can’t be changed, or new tests of already written code that cannot yet be changed. This is precisely the reason why this answer exists and is also why it is not the default answer in Phake.

Capturing a Return Value

Another tool in Phake for testing legacy code is the captureReturnTo() answer. This performs a function similar to argument capturing, however it instead captures what the actual method of a mock object returns to the variable passed as its parameter. Again, this should never be needed if you are testing newly written code. However I have ran across cases several times where legacy code calls protected factory methods and the result of the method call is never exposed. This answer gives you a way to access that variable to ensure that the factory was called and is operating correctly in the context of your method that is being tested.

Custom Answers

While the answers provided in Phake should be able to cover most of the scenarios you will run into when using mocks in your unit tests there may occasionally be times when you need more control over what is returned from your mock methods. When this is the case, you can use a custom answer. All answers in Phake implement the Phake_Stubber_IAnswer interface. This interface defines a single method called getAnswer() that can be used to return what will be returned from a call to the method being stubbed. If you need to get access to how the method you are stubbing was invoked, there is a more complex set of interfaces that can be implemented: Phake_Stubber_Answers_IDelegator and Phake_Stubber_IAnswerDelegate.

Phake_Stubber_Answers_IDelegator extends Phake_Stubber_IAnswer and defines an additional method called processAnswer() that is used to perform processing on the results of getAnswer() prior to passing it on to the stub’s caller. Phake_Stubber_IAnswerDelegate defines an interface that allows you to create a callback that is called to generate the answer from the stub. It defines getCallBack() which allows you to generate a PHP callback based on the object, method, and arguments that a stub was called with. It also defines getArguments() which allows you to generate the arguments that will be passed to the callback based on the method name and arguments the stub was called with.

Partial Mocks

When testing legacy code, if you find that the majority of the methods in the mock are using the thenCallParent() answer, you may find it easier to just use a partial mock in Phake. Phake partial mocks also allow you to call the actual constructor of the class being mocked. They are created using Phake::partialMock(). Like Phake::mock(), the first parameter is the name of the class that you are mocking. However, you can pass additional parameters that will then be passed as the respective parameters to that class’ constructor. The other notable feature of a partial mock in Phake is that its default answer is to pass the call through to the parent as if you were using thenCallParent().

Consider the following class that has a method that simply returns the value passed into the constructor.

Example 3.9. A Test Class

```
<?php
class MyClass
{
 private $value;

 public __construct($value)
 {
  $this->value = $value;
 }

 public function foo()
 {
  return $this->value;
 }
}
?>
```

Using Phake::partMock() you can instantiate a mock object that will allow this object to function as designed while still allowing verification as well as selective stubbing of certain calls. Below is an example that shows the usage of Phake::partMock()

Example 3.10. Using Partial Mocks

```
<?php
class MyClassTest extends PHPUnit_Framework_TestCase
{
 public function testCallingParent()
 {
  $mock = Phake::partialMock('MyClass', 42);

  $this->assertEquals(42, $mock->foo());
 }
}
?>
```

Again, partial mocks should not be used when you are testing new code. If you find yourself using them be sure to inspect your design to make sure that the class you are creating a partial mock is not doing too much.

Setting Default Stubs

You can also change the default stubbing for mocks created with Phake::mock(). This is done by using the second parameter to Phake::mock() in conjunction with the Phake::ifUnstubbed() method. The second parameter to Phake::mock() is reserved for configuring the behavior of an individual mock. Phake::ifUnstubbed() allows you to specify any of the matchers mentioned above as the default answer if any method invocation is not explicitly stubbed. If this configuration directive is not provided then the method will return NULL by default. An example of this can be seen below.

Example 3.11. Setting the default stub

```
<?php
class MyClassTest extends PHPUnit_Framework_TestCase
{
 public function testDefaultStubs()
 {
  $mock = Phake::mock('MyClass', Phake::ifUnstubbed()->thenReturn(42));

  $this->assertEquals(42, $mock->foo());
 }
}
?>
```

Stubbing Magic Methods

The verification of __call() was discussed in the previous chapter. Magic methods can also be stubbed in much the same way. If you want to verify a particular invocation of __call() you can stub the actual method call by mocking the method passed in as the first parameter.

Consider the following class.

Example 3.12. A Magic Class

```
<?php
class MagicClass
{
public function __call($method, $args)
{
return '__call';
}
}
?>
```

You could stub an invocation of the __call() method through a userspace call to magicCall() with the following code.

Example 3.13. Implicitly Stub of __call()

```
<?php
class MagicClassTest extends PHPUnit_Framework_TestCase
{
  public function testMagicCall()
  {
    $mock = Phake::mock('MagicClass');

    Phake::when($mock)->magicCall()->thenReturn(42);

  $this->assertEquals(42, $mock->magicCall());
  }
}
?>
```

If for any reason you need to explicitly stub calls to __call() then you can use Phake::whenCallMethodWith().

Example 3.14. Explicitly Stubbing __call()

```
<?php
class MagicClassTest extends PHPUnit_Framework_TestCase
{
  public function testMagicCall()
  {
    $mock = Phake::mock('MagicClass');

    Phake::whenCallMethodWith('magicCall')->isCalledOn($mock)->thenReturn(42);

  $this->assertEquals(42, $mock->magicCall());
  }
}
?>
```

Chapter 4. Method Parameter Matchers

The verification and stubbing functionality in Phake both rely heavily on parameter matching to help the system understand exactly which calls need to be verified or stubbed. Phake provides several options for setting up parameter matches.

The most common scenario for matching parameters as you use mock objects is matching on equal variables For this reason the default matcher will ensure that the parameter you pass to the mock method is equal (essentially using the '==' notation) to the parameter passed to the actual invocation before validating the call or returning the mocked stub. So going back to the card game demonstration from the introduction. Consider the following interface:

Example 4.1. Dealer Strategy Interface

```
<?php
interface DealerStrategy
{
  public function deal(
      CardCollection $deck,
      PlayerCollection $players);
}

?>
```

Here we have a deal() method that accepts two parameters. If you want to verify that deal() was called, chances are very good that you want to verify the the parameters as well. To do this is as simple as passing those parameters to the deal() method on the Phake::verify($deal) object just as you would if you were calling the actual deal() method itself. Here is a short albeit silly example:

Example 4.2. Example of Default 'Equals' Matching

```
<?php
//I don't have Concrete versions of
// CardCollection or PlayerCollection yet
$deck = Phake::mock('CardCollection');
$players = Phake::mock('PlayerCollection');


$dealer = Phake::mock('DealerStrategy');

$dealer->deal($deck, $players);

Phake::verify($dealer)->deal($deck, $players);
?>
```

In this example, if I were to have accidentally made the call to deal() with a property that was set to null as the first parameter then my test would fail with the following exception:

Example 4.3. Failed Simple Equals Test

```
Expected DealerStrategy->fooWithArgument(equal to
<object:CardCollection>, equal to <object:PlayerCollection>)
to be called exactly 1 times, actually called 0 times.
Other Invocations:
  PhakeTest_MockedClass->fooWithArgument(<null>,
equal to <object:PlayerCollection>)
```

Determining the appropriate method to stub works in exactly the same way.

There may be cases when it is necessary to verify or stub parameters based on something slightly more complex then basic equality. This is what we will talk about next.

Using PHPUnit Matchers

Phake was developed with PHPUnit in mind. It is not dependent on PHPUnit, however if PHPUnit is your testing framework of choice there is some special integration available. Any constraints made available by the PHPUnit framework will work seemlessly inside of Phake. Here is an example of how the PHPUnit constraints can be used:

Example 4.4. Using PHPUnit Matchers

```
<?php
class TestPHPUnitConstraint extends PHPUnit_Framework_TestCase
{
  public function testDealNumberOfCards()
  {
    $deck = Phake::mock('CardCollection');
    $players = Phake::mock('PlayerCollection');

    $dealer = Phake::mock('DealerStrategy');
    $dealer->deal($deck, $players, 11);

    Phake::verify($dealer)
      ->deal($deck, $players, $this->greaterThan(10));
  }
}
?>
```

I have added another parameter to my deal() method that allows me to specify the number of cards to deal to each player. In the test above I wanted to verify that the number passed to this parameter was greater than 10.

For a list of the constraints you have available to you through PHPUnit, I recommend reading the
<a>PHPUnit's documentation on assertions and constraints</a>
. Any constraint that can be used with assertThat() in PHPUnit can also be used in Phake.

Using Hamcrest Matchers

If you do not use PHPUnit, Phake also supports Hamcrest matchers. This is in-line with the Phake's design goal of being usable with any testing framework. Here is a repeat of the PHPUnit example, this time using

SimpleTest and Hamcrest matchers. (Please note I will probably add support for simple test directly at a later date.)

Example 4.5. Using Hamcrest Matchers

```php
<?php
class TestHamcrestMatcher extends UnitTestCase
{
  public function testDealNumberOfCards()
  {
    $deck = Phake::mock('CardCollection');
    $players = Phake::mock('PlayerCollection');

    $dealer = Phake::mock('DealerStrategy');
    $dealer->deal($deck, $players, 11);

    Phake::verify($dealer)->deal($deck, $players, greaterThan(10));
  }
}
?>
```

Parameter Capturing

As you can see there are a variety of methods for verifying that the appropriate parameters are being passed to methods. However, there may be times when the prebuilt constraints and matchers simply do not fit your needs. Perhaps there is method that accepts a complex object where only certain components of the object need to be validated. Parameter capturing will allow you to store the parameter that was used to call your method so that it can be used in assertions later on.

Consider the following example where I have defined a getNumberOfCards() method on the CardCollection interface.

Example 4.6. Card Collection Interface

```php
<?php
interface CardCollection
{
  public function getNumberOfCards();
}
?>
```

I want to create new functionality for a my poker dealer strategy that will check to make sure we are playing with a full deck of 52 cards when the deal() call is made. It would be rather cumbersome to create a copy of a CardCollection implementation that I could be sure would match in an equals scenario. Such a test would look something like this.

Please note, I do not generally advocate this type of design. I prefer a dependency injection versus instantiation. So please remember, this is not an example of clean design, simply an example of what you can do with argument capturing.

Example 4.7. Using Argument Captors

```
<?php
class MyPokerGameTest extends PHPUnit_Framework_TestCase
{
  public function testDealCards()
  {
    $dealer = Phake::mock('MyPokerDealer');
    $players = Phake::mock('PlayerCollection');

    $cardGame = new MyPokerGame($dealer, $players);

    Phake::verify($dealer)->deal(Phake::capture($deck), $players);

    $this->assertEquals(52, $deck->getNumberOfCards());
  }
}
?>
```

You can also capture parameters if they meet a certain condition. For instance, if someone mistakenly passed an array as the first parameter to the deal() method then PHPUnit would fatal error out. This can be protected against by using the the Phake::capture()->when() method. The when() method accepts the same constraints that Phake::verify() accepts. Here is how you could leverage that functionality to bulletproof your captures a little bit.

Example 4.8. Using Argument Captors With Conditions

```
<?php
class MyBetterPokerGameTest extends PHPUnit_Framework_TestCase
{
  public function testDealCards()
  {
    $dealer = Phake::mock('MyPokerDealer');
    $players = Phake::mock('PlayerCollection');

    $cardGame = new MyPokerGame($dealer, $players);

    Phake::verify($dealer)->deal(
      Phake::capture($deck)
        ->when($this->isInstanceOf('CardCollection')),
      $players
    );

    $this->assertEquals(52, $deck->getNumberOfCards());
  }
}
?>
```

This could also be done by using PHPUnit's assertions later on with the captured parameter, however this also has a side effect of better localizing your error. Here is the error you would see if the above test failed.

Example 4.9. Failed Test of Argument Captor

```
Exception: Expected MyPokerDealer->deal(<captured parameter>,
equal to <object:PlayerCollection>) to be called exactly 1
times, actually called 0 times.
Other Invocations:
  PhakeTest_MockedClass->fooWithArgument(<array>,
<object:PlayerCollection>)
```

It should be noted that while it is possible to use argument capturing for stubbing with Phake::when() I would discourage it. When stubbing a method you should only be concerned about making sure an expected value is return and argument capturing in no way helps with that goal. In the worst case scenario you will have some incredibly difficult test failures to diagnose.

Custom Parameter Matchers

An alternative to using argument capturing is creating custom matchers. All parameter matchers implement the interface Phake_Matchers_IArgumentMatcher. You can create custom implementations of this interface. This is especially useful if you find yourself using a similar capturing pattern over and over again. If I were to rewriting the test above using a customer argument matcher it would look something like this.

Example 4.10. Custom Argument Matcher

```
<?php
class FiftyTwoCardDeckMatcher implements Phake_Matchers_IArgumentMatcher
{
  public function matches($argument)
  {
    return ($argument instanceof CardCollection
        && $argument->getNumberOfCards() == 52);
  }

  public function __toString()
  {
    return '<object:CardCollection with 52 cards>';
  }
}

class MyBestPokerGameTest extends PHPUnit_Framework_TestCase
{
  public function testDealCards()
  {
    $dealer = Phake::mock('MyPokerDealer');
    $players = Phake::mock('PlayerCollection');

    $cardGame = new MyPokerGame($dealer, $players);

    Phake::verify($dealer)->deal(new 52CardDeckMatcher(), $players);
  }
}
?>
```