Phake - PHP Mocking Framework Documentation

Release 4.0.0

Mike Lively <m@digitalsandwich.com>

Apr 12, 2021

Contents

CHAPTER 1

Introduction to Phake

Phake is a mocking framework for PHP. It allows for the creation of objects that mimic a 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. In the example below I have created a new class called `CardGame` that implements the basic functionality for a card game:

```
class CardGame
{
    private DealerStrategy   $dealerStrategy;
    private CardCollection   $deck;
    private PlayerCollection $players;

    public function CardGame(DealerStrategy $dealerStrategy, CardCollection $deck,␣
↪PlayerCollection $players)
    {
        $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 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 my 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:

```
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();
        $players = new PlayerCollection([$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 `DealerStrategy::shuffle()`.

This is the type of problem that mock 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. If this test were re-written to use Phake, it would become:

```
class CardGameTest2 extends PHPUnit\Framework\TestCase
{
    public function testDealCards()
    {
        $dealer  = Phake::mock(DealerStrategy::class);
        $deck    = Phake::mock(CardCollection::class);
        $players = Phake::mock(PlayerCollection::class);

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

(continues on next page)

(continued from previous page)

```
        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 benefit is that 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.

CHAPTER 2

Getting Started

Phake 4.0 depends on PHP 7.1 or greater. It has no dependency on PHPUnit and should be usable with any version of PHPUnit compatible with your PHP version.

2.1 Composer Install

Phake can be installed via Composer. You will typically want to install Phake as a development requirement. To do so you can add the following to your `composer.json` file:

```
{
    // ..
    "require-dev": {
        "phake/phake": "^4.0"
    }
    // ..
}
```

Once this is added to `composer.json` you can run `composer update phake/phake`

2.2 Install from Source

You can also clone a copy of Phake from the Phake GitHub repository. 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.

2.3 Support

If you think you have found a bug or an issue with Phake, please feel free to open up an issue on the Phake Issue Tracker

CHAPTER 3

Creating Mocks

The `Phake::mock()` method is how you create new test doubles in Phake. You pass in the class name of what you would like to mock.

```
$mock = Phake::mock(ClassToMock::class);
```

The `$mock` variable is now an instance of a generated class that inherits from `ClassToMock` with hooks that allow you to force functions to return known values. By default, all methods on a mock object will return the type specified in the return type or null. This behavior can be overridden on a per method and even per parameter basis. This will be covered in depth in *Method Stubbing*.

The mock will also record all calls made to this class so that you can later verify that specific methods were called with the proper parameters. This will be covered in depth in *Method Verification*.

In addition to classes you can also mock interfaces directly. This is done in much the same way as a class name, you simply pass the interface name as the first parameter to `Phake::mock()`.

```
$mock = Phake::mock(InterfaceToMock::class);
```

You can also pass an array of interface names to `Phake::mock()` that also contains up to 1 class name. This allows for easier mocking of a dependency that is required to implement multiple interfaces.

```
$mock = Phake::mock([ Interface1::class, Interface2::class ]);
```

3.1 Partial Mocks

When testing legacy code, you may find that a better default behavior for the methods is to actually call the original method. This can be accomplished by stubbing each of the methods to return `thenCallParent()`. You can learn more about this in *Calling the Parent*.

While this is certainly possible, 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.

```
class MyClass
{
    private $value;

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

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

Using `Phake::partialMock()` 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::partialMock()`.

```
class MyClassTest extends PHPUnit\Framework\TestCase
{
    public function testCallingParent()
    {
        $mock = Phake::partialMock(MyClass::class, 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 for is not doing too much.

3.2 Calling Private and Protected Methods on Mocks

It is possible to invoke protected and private methods on your mocks using Phake. When you mock a class, the mocked version will retain the same visibility on each of its functions as you would have had on your original class. However, using `Phake::makeVisible()` and `Phake::makeStaticsVisible()` you can allow direct invocation of instance methods and static methods accordingly. Both of these methods accept a mock object as its only parameter and returns a proxy class that you can invoke the methods on. Method calls on these proxies will still return whatever value was previously stubbed for that method call. So if you intend on the original method being called and you aren't using *Partial Mocks*, then you can just enable calling-the-parent for that method call using the `thenCallParent()` answer. This is all discussed in greater depth in method-stubbing and *Answers*.

```
class MyClass
{
    private function foo()
    {
    }
```

(continues on next page)

(continued from previous page)

```
    private static function bar()
    {
    }
}
```

Given the class above, you can invoke both private methods with the code below.

```
$mock = Phake::mock(MyClass::class);

Phake::makeVisible($mock)->foo();

Phake::makeStaticsVisible($mock)->bar();
```

As you can see above when using the static variant you still call the method as though it were an instance method. The other thing to take note of is that there is no modification done on $mock itself. If you use `Phake::makeVisible()` you will only be able to make those private and protected calls off of the return of that method itself.

The best use case for this feature of Phake is if you have private or protected calls that are nested deep inside of public methods. Generally speaking you would always just test from your class's public interface. However these large legacy classes often require a significant amount of setup within fixtures to allow for calling those private and protected methods. If you are only intending on refactoring the private and protected method then using `Phake::makeVisible()` removes the need for these complex fixtures.

Consider this really poor object oriented code. The cleanRowContent() function does some basic text processing such as stripping html tags, cleaning up links, etc. It turns out that the original version of this method is written in a very unperformant manner and I have been tasked with rewriting it.

```
class MyReallyTerribleOldClass
{
    public function __construct(Database $db)
    {
        //...
    }

    public function doWayTooMuch($data)
    {
        $result = $this->db->query($this->getQueryForData($data))

        $rows = array();
        while ($row = $this->db->fetch($result))
        {
            $rows[] = $this->cleanRowContent($row);
        }

        return $rows;
    }

    private function cleanRowContent($row)
    {
        //...
    }

    private function getQueryForData($data)
    {
        //...
```

(continues on next page)

(continued from previous page)

```
    }
}
```

If I was about to make changes to cleanRowContent and wanted to make sure I didn't break previous functionality, in order to do so with the traditional fixture I would have to write a test similar to the following:

```
class Test extends PHPUnit\Framework\TestCase
{
    public function testProcessRow()
    {
        $dbRow = [ 'id' => '1', 'content' => 'Text to be processed with <b>tags␣
↪stripped</b>' ];
        $expectedValue = [ [ 'id' => 1', 'content' => 'Text to be processed with tags␣
↪stripped' ] ];

        $db = Phake::mock(Database::class);
        $result = Phake::mock(DatabaseResult::class);
        $oldClass = new MyReallyTerribleOldClass($db);

        Phake::when($db)->query->thenReturn($result);

        Phake::when($db)->fetch->thenReturn($dbRow)->thenReturn(null);

        $data = $oldClass->doWayTooMuch(array());

        $this->assertEquals($expectedValue, $data);
    }
}
```

Using test helpers or PHPUnit data providers I could reuse this test to make sure I fully cover the various logic paths and use cases for the cleanRowContent(). However this test is doing alot of work to just set up this scenario. Whenever your test is hitting code not relevant to your test in increases the test's fragility. Here is how you could test the same code using `Phake::makeVisible()`.

```
class Test extends PHPUnit\Framework\TestCase
{
    public function testProcessRow()
    {
        $dbRow = [ 'id' => '1', 'content' => 'Text to be processed with <b>tags␣
↪stripped</b>' ];
        $expectedValue = [ [ 'id' => 1', 'content' => 'Text to be processed with tags␣
↪stripped' ] ];

        $oldClass = new Phake::partialMock(MyReallyTerribleOldClass::class);

        $data = Phake::makeVisible($oldClass)->cleanRowContent($dbRow);
        $this->assertEquals($expectedValue, $data);
    }
}
```

As you can see the test is significantly simpler. One final note, if you find yourself using this strategy on newly written code, it could be a code smell indicitive of a class or public method doing too much. It is very reasonable to argue that in my example, the `cleanRowContent()` method should be a class in and of itself or possibly a method on a string manipulation type of class that my class then calls out to. This is a better design and also a much easier to test design.

CHAPTER 4

Method Stubbing

The `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.

Imagine I was in the process of building the next great online shopping cart. The first thing any good shopping cart allows is to be able to add items. The most important thing I want to know from the shopping cart is how much money in merchandise is in there. So, I need to make myself a ShoppingCart class. I also am going to need some class to define my items. I am more worried about the money right now and because of that I am keenly aware that any item in a shopping cart is going to have a price. So I will just create an interface to represent those items called Item. Now take a minute to marvel at the creativity of those names. Great, now check out the initial definitions for my objects.

```
/**
 * An item that is going to make me rich.
 */
interface Item
{
    public function getPrice()
}

/**
 * A customer's cart that will contain items that are going to make me rich.
 */
class ShoppingCart
{
    private array $items = [];

    /**
     * Adds an item to the customer's order
     */
    public function addItem(Item $item): void
    {
        $this->items[] = $item;
    }
```

(continues on next page)

(continued from previous page)

```
    /**
     * Returns the current sub total of the customer's order
     */
    public function getSubTotal()
    {
    }
}
```

So, I am furiously coding away at this fantastic new `ShoppingCart` class when I realize, I am doing it wrong! You see, a few years ago I went to this conference with a bunch of other geeky people to talk about how to make quality software. I am supposed to be writing unit tests. Here I am, a solid thirteen lines (not counting comments) of code into my awe inspiring new software and I haven't written a single test. I tell myself, "There's no better time to change than right now!" So I decide to start testing. After looking at the options I decide PHPUnit with this sweet new mock library called Phake is the way to go.

My first test is going to be for the currently unimplemented `ShoppingCart::getSubTotal()` method. I already have a pretty good idea of what this function is going to need to do. It will need to look at all of the items in the cart, retrieve their price, add it all together and return the result. So, in my test I know I am going to need a fixture that sets up a shopping cart with a few items added. Then I am going to need a test that calls `ShoppingCart::getSubTotal()` and asserts that it returns a value equal to the price of the items I added to the cart. One catch though, I don't have any concrete instances of an `Item`. I wasn't even planning on doing any of that until tomorrow. I really want to just focus on the `ShoppingCart` class. Never fear, this is why I decided to use Phake. I remember reading about how it will allow me to quickly create instance of my classes and interfaces that I can set up stubs for so that method calls return predictable values. This project is all coming together and I am really excited.

```
class ShoppingCartTest extends PHPUnit\Framework\TestCase
{
    public function testGetSub()
    {
        $item1 = Phake::mock(Item::class);
        $item2 = Phake::mock(Item::class);
        $item3 = Phake::mock(Item::class);

        Phake::when($item1)->getPrice()->thenReturn(100);
        Phake::when($item2)->getPrice()->thenReturn(200);
        Phake::when($item3)->getPrice()->thenReturn(300);

        $shoppingCart = new ShoppingCart();
        $shoppingCart->addItem($item1);
        $shoppingCart->addItem($item2);
        $shoppingCart->addItem($item3);

        $this->assertEquals(600, $shoppingCart->getSubTotal());
    }
}
```

My test here shows a very basic use of Phake for creating method stubs. I am creating three different mock implementations of the `Item` class. Then for each of those item classes, I am creating a stub using `Phake::when()` that will return 100, 200, and 300 respectively. I know my method that I am getting ready to implement will need to call those methods in order to calculate the total cost of the order.

My test is written so now it is time to see how it fails. I run it with phpunit and see the output below:

```
$ phpunit ExampleTests/ShoppingCartTest.php
PHPUnit 9.5.4 by Sebastian Bergmann and contributors.

```

(continues on next page)

(continued from previous page)

```
F                                                                   1 / 1 (100%)

Time: 00:00.012, Memory: 4.00 MB

There was 1 failure:

1) ShoppingCartTest::testGetSub
Failed asserting that null matches expected 600.

/home/mikel/Documents/Projects/Phake/tests/ShoppingCartTest.php:69

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Now that I have a working (and I by working I mean breaking!) test it is time to look at the code necessary to make the test pass.

```
class ShoppingCart
{
    // I am cutting out the already seen code. If you want to see it again look at␣
↪the previous examples!

    /**
     * Returns the current sub total of the customer's order
     */
    public function getSubTotal()
    {
        $total = 0;

        foreach ($this->items as $item)
        {
            $total += $item->getPrice();
        }

        return $total;
    }
}
```

The code here is pretty simple. I am just iterating over the `ShoppingCart::$item` property, calling the `Item::getPrice()` method, and adding them all together. Now when I run phpunit, the tests were successful and I am getting off to a great start with my shopping cart.

```
$ phpunit ExampleTests/ShoppingCartTest.php
PHPUnit 9.5.4 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 00:00.011, Memory: 4.00 MB

OK (1 test, 1 assertion)
```

So, what is Phake doing here? Phake is providing us a predictable implementation of the `Item::getPrice()` method that we can use in our test. It helps me to ensure the when my test breaks I know exactly where it is breaking. I will not have to be worried that a bad implementation of `Item::getPrice()` is breaking my tests.

4.1 How Phake::when() Works

Internally Phake is doing quite a bit when this test runs. The three calls to `Phake::mock()` are creating three new classes that in this case each implement the `Item` interface. These new classes each define implementations of any method defined in the `Item` interface. If `Item` extended another interface, implementations of all of that parent's defined methods would be created as well. Each method being implemented in these new classes does a few different things. The first thing that it does is record the fact that the method was called and stores the parameters that were used to call it. The next significant thing it does is looks at the stub map for that mock object. The stub map is a map that associates answers to method matchers. An answer is what a mocked object will return when it is called. By default, a call to a mock object returns a static answer of NULL. We will discuss answers more in *Answers*. A method matcher has two parts. The first is the method name. The second is an array of arguments. The array of arguments will then contain various constraints that are applied to each argument to see if a given argument will match. The most common constraint is an equality constraint that will match loosely along the same lines as the double equals sign in PHP. We will talk about matchers more in *Method Parameter Matchers*.

When each mock object is initially created, its stub map will be empty. This means that any call to a method on a mock object is going to return a default answer of NULL. If you want your mock object's methods to return something else you must add answers to the stub map. The `Phake::when()` method allows you to map an answer to a method matcher for a given mock object. The mock object you want to add the mapping to is passed as the first parameter to `Phake::when()`. The `Phake::when()` method will then return a proxy that can be used add answers to your mock object's stub map. The answers are added by making method calls on the proxy just as you would on the mock object you are proxying. In the first example above you saw a call to `Phake::when($this->item1)->getPrice()`. The `getPrice()` call here was telling Phake that I am about to define a new answer that will be returned any time `$this->item->getPrice()` is called in my code. The call to `$this->item->getPrice()` returns another object that you can set the answer on using Phake's fluent api. In the example I called `Phake::when($this->item1)->getPrice()->thenReturn(100)`. The `thenReturn()` method will bind a static answer to a matcher for `getPrice()` in the stub map for $this->item1.

4.2 Overwriting Existing Stubs

My shopping cart application is coming right along. I can add items and the total price seems to be accurate. However, while I was playing around with my new cart I noticed a very strange problem. I was playing around with the idea of allowing discounts to be applied to a cart as just additional items that would have a negative price. So while I am playing around with this idea I notice that the math isn't always adding up. If I start with an item that is $100 and then add a discount that is $81.40 I see that the total price isn't adding up to $18.60. This is definitely problematic After doing some further research, I realize I made a silly mistake. I am just using simple floats to calculate the costs. Floats are by nature inaccurate. Once you start using them in mathematical operations they start to show their inadequacy for precision. In keeping with the test driven method of creating code I need to create a unit test this flaw.

```
class ShoppingCartTest extends PHPUnit\Framework\TestCase
{
      public function testGetSub()
      {
          $item1 = Phake::mock(Item::class);
          $item2 = Phake::mock(Item::class);
          $item3 = Phake::mock(Item::class);

          Phake::when($item1)->getPrice()->thenReturn(100);
          Phake::when($item2)->getPrice()->thenReturn(200);
          Phake::when($item3)->getPrice()->thenReturn(300);

          $shoppingCart = new ShoppingCart();
          $shoppingCart->addItem($item1);
```

(continues on next page)

(continued from previous page)

```
            $shoppingCart->addItem($item2);
            $shoppingCart->addItem($item3);

            $this->assertEquals(600, $shoppingCart->getSubTotal());
        }

        public function testGetSubTotalWithPrecision()
        {
            $item1 = Phake::mock(Item::class);
            $item2 = Phake::mock(Item::class);
            $item3 = Phake::mock(Item::class);

            Phake::when($item1)->getPrice()->thenReturn(100);
            Phake::when($item2)->getPrice()->thenReturn(-81.4);
            Phake::when($item3)->getPrice()->thenReturn(20);

            $shoppingCart = new ShoppingCart();
            $shoppingCart->addItem($item1);
            $shoppingCart->addItem($item2);
            $shoppingCart->addItem($item3);

            $this->assertEquals(38.6, $shoppingCart->getSubTotal());
        }
}
```

You can see that I added another test method that uses actual floats for some of the prices as opposed to round numbers. Now when I run my test suite I can see the result.

```
$ phpunit ExampleTests/ShoppingCartTest.php
PHPUnit 9.5.4 by Sebastian Bergmann and contributors.

..                                                                  2 / 2 (100%)

Time: 00:00.012, Memory: 4.00 MB

OK (2 tests, 2 assertions)
```

Once you confirmed that your implementation works, I want to discuss streamlining test cases with you. You will notice that the code in `ShoppingCartTest::testGetSubTotalWithPrecision()` contains almost all duplicate code when compared to `ShoppingCartTest::testGetSub()`. If I were to continue following this pattern of doing things I would eventually have tests that are difficult to maintain. Phake allows you to very easily override stubs. This is very important in helping you to reduce duplication in your tests and leads to tests that will be easier to maintain. To overwrite a previous stub you simply have to redefine it. I am going to change `ShoppingCartTest::testGetSubTotalWithPrecision()` to instead just redefine the `getPrice()` stubs.

```
class ShoppingCartTest extends PHPUnit\Framework\TestCase
{
    private ShoppingCart $shoppingCart;
    private Item $item1;
    private Item $item2;
    private Item $item3;

    public function setUp(): void
    {
        $this->item1 = Phake::mock(Item::class);
```

(continues on next page)

(continued from previous page)

```
        $this->item2 = Phake::mock(Item::class);
        $this->item3 = Phake::mock(Item::class);

        Phake::when($this->item1)->getPrice()->thenReturn(100);
        Phake::when($this->item2)->getPrice()->thenReturn(200);
        Phake::when($this->item3)->getPrice()->thenReturn(300);

        $this->shoppingCart = new ShoppingCart();
        $this->shoppingCart->addItem($this->item1);
        $this->shoppingCart->addItem($this->item2);
        $this->shoppingCart->addItem($this->item3);
    }

    public function testGetSub()
    {
        $this->assertEquals(600, $this->shoppingCart->getSubTotal());
    }

    public function testGetSubTotalWithPrecision()
    {
        Phake::when($this->item1)->getPrice()->thenReturn(100);
        Phake::when($this->item2)->getPrice()->thenReturn(-81.4);
        Phake::when($this->item3)->getPrice()->thenReturn(20);

        $this->assertEquals(38.6, $this->shoppingCart->getSubTotal());
    }
}
```

If you rerun this test you will get the same results shown in before. The test itself is much simpler though there is much less unnecessary duplication. The reason this works is because the stub map I was referring to in *How Phake::when() Works* isn't really a map at all. It is more of a stack in reality. When a new matcher and answer pair is added to a mock object, it is added to the top of the stack. Then whenever a stub method is called, the stack is checked from the top down to find the first matcher that matches the method that was called. So, when I created the additional stubs for the various `Item::getPrice()` calls, I was just adding additional matchers to the top of the stack that would always get matched first by virtue of the parameters all being the same.

4.3 Resetting A Mock's Stubs

If overriding a stub does not work for your particular case and you would rather start over with all default stubs then you can use `Phake::reset()` and `Phake::resetStatic()`. These will remove all stubs from a mock and also empty out all recorded calls against a mock. `Phake::reset()` will do this for instance methods on the mock and `Phake::resetStatic()` will do this for all static methods on the mock.

```
public function testResettingStubMapper()
{
    $mock = Phake::mock(PhakeTest_MockedClass::class);
    Phake::when($mock)->foo()->thenReturn(42);

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

    Phake::reset($mock);
    //$mock->foo() now returns the default stub which in this case is null
    $this->assertNull($mock->foo());
}
```

(continues on next page)

(continued from previous page)

```
public function testResettingCallRecorder()
{
    $mock = Phake::mock(PhakeTest_MockedClass::class);
    $mock->foo();

    //Will work as normal
    Phake::verify($mock)->foo();

    Phake::reset($mock);

    //Will now throw an error that foo was not called
    Phake::verify($mock)->foo();
}
```

4.4 Stubbing Multiple Calls

Another benefit of the stub mapping in Phake is that it allows you to very easily stub multiple calls to the same method that use different parameters. In my shopping cart I have decided to add some functionality that will allow me to easily add multiple products that are a part of a group to the shopping cart. To facilitate this I have decided to create a new class called `ItemGroup`. The `ItemGroup` object will be constructed with an array of `Items`. It will have a method on the class that will add all of the items in the group to the given cart and then the total price of items in the cart will be returned.

It should be noted that earlier I decided to make a small change to the `ShoppingCart::addItem()` method to have it return the total price of items in the cart. I figured that this would be nice api level functionality to make working with the system a little bit easier. I would like to take advantage of that change with this code. Here's a stub of the functionality I am considering.

```
/**
 * A group of items that can be added to a cart all at the same time
 */
class ItemGroup
{
    /**
     * @param array $items an array of Item objects
     */
    public function __construct(array $items)
    {
    }

    /**
     * @param ShoppingCart $cart
     * @return money The new total value of the cart
     */
    public function addItemsToCart(ShoppingCart $cart)
    {
    }
}
```

The next test I am going to write now is going to be focusing on this new `ItemGroup::addItemsToCart()` method. In my test's `setUp()` method I'll create a new instance of `ItemGroup` which will require one or more `Item` implementations. I'll use mocks for those. Then the actual test case I am going to start with will be a test to assert that `ItemGroup::addItemsToCart()` returns the new shopping cart value. I already know that I am

going to need to get this value by looking at the last return value from calls to `ShoppingCart::addItem()`. To allow for checking this I will mock `ShoppingCart` and create three stubs for `ShoppingCart::addItem()`. Each stub will be for a call with a different `Item`.

```
class ItemGroupTest extends PHPUnit\Framework\TestCase
{
    private ItemGroup $itemGroup;
    private Item $item1;
    private Item $item2;
    private Item $item3;

    public function setUp(): void
    {
        $this->item1 = Phake::mock(Item::class);
        $this->item2 = Phake::mock(Item::class);
        $this->item3 = Phake::mock(Item::class);

        $this->itemGroup = new ItemGroup([ $this->item1, $this->item2, $this->item3␣
↪]);
    }

    public function testAddItemsToCart()
    {
        $cart = Phake::mock(ShoppingCart::class);
        Phake::when($cart)->addItem($this->item1)->thenReturn(10);
        Phake::when($cart)->addItem($this->item2)->thenReturn(20);
        Phake::when($cart)->addItem($this->item3)->thenReturn(30);

        $totalCost = $this->itemGroup->addItemsToCart($cart);
        $this->assertEquals(30, $totalCost);
    }
}
```

In this example the `ShoppingCart::addItem()` method is being stubbed three times. Each time it is being stubbed with a different parameter being passed to `addItem()`. This a good example of how parameters are also checked whenever Phake looks at a mock object's stub map for answers. The default behavior of argument matching is again a loose equality check. Similar to how you would use the double equals operator in PHP. The other options for argument matching are discussed further in *Method Parameter Matchers*.

4.5 Stubbing Consecutive Calls

The previous test was a great example for how you can make multiple stubs for a single method however in reality it is not the best way for that particular test to be written. What if the `Item` objects in an `ItemGroup` aren't stored in the order they were passed in? I am needlessly binding my test to the order in which objects are stored. Phake provides the ability to map multiple answers to the same stub. This is done simply by chaining the answers together. I could rewrite the test from the previous chapter to utilize this feature of Phake.

```
class ItemGroupTest extends PHPUnit\Framework\TestCase
{
    private ItemGroup $itemGroup;
    private Item $item1;
    private Item $item2;
    private Item $item3;

    public function setUp(): void
```

(continues on next page)

(continued from previous page)

```
    {
        $this->item1 = Phake::mock(Item::class);
        $this->item2 = Phake::mock(Item::class);
        $this->item3 = Phake::mock(Item::class);

        $this->itemGroup = new ItemGroup([ $this->item1, $this->item2, $this->item3␣
↪]);
    }

    public function testAddItemsToCart()
    {
        $cart = Phake::mock(ShoppingCart::class);
        Phake::when($cart)->addItem(Phake::anyParameters())->thenReturn(10)
            ->thenReturn(20)
            ->thenReturn(30);

        $totalCost = $this->itemGroup->addItemsToCart($cart);
        $this->assertEquals(30, $totalCost);
    }
}
```

You will notice a few of differences between this example and the example in *Stubbing Multiple Calls*. The first difference is that there is only one call to `Phake::when()`. The second difference is that I have chained together three calls to `thenReturn()`. The third difference is instead of passing one of my mock Item objects I have passed the result of the `Phake::anyParameters()` method. This is a special argument matcher in Phake that essentially says match any call to the method regardless of the number of parameters or the value of those parameters. You can learn more about `Phake::anyParameters()` in *Wildcard Parameters*.

So, this single call to `Phake::when()` is saying: "Whenever a call to `$cart->addItem()` is made, regardless of the parameters, return 10 for the first call, 20 for the second call, and 30 for the third call." If you are using consecutive call stubbing and you call the method more times than you have answers set, the last answer will continue to be returned. In this example, if `$cart->addItem()` were called a fourth time, then 30 would be returned again.

4.6 Stubbing Reference Parameters

Occasionally you may run into code that utilizes reference parameters to provide additional output from a method. This is not an uncommon thing to run into with legacy code. Phake provides a custom parameter matcher (these are discussed further in *Method Parameter Matchers*) that allows you to set reference parameters. It can be accessed using `Phake::setReference()`. The only parameter to this matcher is the value you would like to set the reference parameter to provided all other parameters match.

```
interface IValidator
{
    /**
     * @parm array $data Data to validate
     * @parm array &$errors contains all validation errors if the data is not valid
     * @return boolean True when the data is valid
     */
    public function validate(array $data, array &$errors);
}

class ValidationLogger implements IValidator
{
    private $validator;
```

(continues on next page)

(continued from previous page)

```
    private $log;

    public function __construct(IValidator $validator, Logger $log)
    {
        $this->validator = $validator;
        $this->log = $log;
    }

    public function validate(array $data, array &$errors)
    {
        if (!$this->validator->validate($data, $errors))
        {
            foreach ($errors as $error)
            {
                $this->log->info("Validation Error: {$error}");
            }

            return false;
        }

        return true;
    }
}

class ValidationLoggerTest extends PHPUnit\Framework\TestCase
{
    public function testValidate()
    {
        //Mock the dependencies
        $validator = Phake::mock(IValidator::class);
        $log = Phake::mock(Logger::class);
        $data = [ 'data1' => 'value' ];
        $expectedErrors = ['data1 is not valid'];

        //Setup the stubs (Notice the Phake::setReference()
        Phake::when($validator)->validate($data, Phake::setReference(
↪$expectedErrors))->thenReturn(false);

        //Instantiate the SUT
        $validationLogger = new ValidationLogger($validator, $log);

        //verify the validation is false and the message is logged
        $errors = [];
        $this->assertFalse($validationLogger->validate($data, $errors));
        Phake::verify($log)->info('Validation Error: data1 is not valid');
    }
}
```

In the example above, I am testing a new class I have created called `ValidationLogger`. It is a decorator for other implementations of `IValidator` that allows adding logging to any other validator. The `IValidator::validate()` method will always return an array of errors into the second parameter (a reference parameter) provided to the method. These errors are what my logger is responsible for logging. So in order for my test to work properly, I will need to be able to set that second parameter as a part of my stubbing call.

In the call to `Phake::when($validator)->validate()` I have passed a call to `Phake::setReference()` as the second parameter. This is causing the mock implementation of `IValidator` to set `$errors` in `ValidationLogger::validate()` to the array specified by `$expectedErrors`. This

allows me to quickly and easily validate that I am actually logging the errors returned back in the reference parameter.

By default `Phake::setReference()` will always return true regardless of the parameter initially passed in. If you would like to only set a reference parameter when that reference parameter was passed in as a certain value you can use the `when()` modifier. This takes a single parameter matcher as an argument. Below, you will see that the test has been modified to call `when()` on the result of *Phake::setReference()*'. This modification will cause the reference parameter to be set only if the $errors parameter passed to `IValidator::validate()` is initially passed as an empty array.

```
class ValidationLoggerTest extends PHPUnit\Framework\TestCase
{
    public function testValidate()
    {
        //Mock the dependencies
        $validator = Phake::mock(IValidator::class);
        $log = Phake::mock(Logger::class);
        $data = [ 'data1' => 'value' ];
        $expectedErrors = [ 'data1 is not valid' ];

        //Setup the stubs (Notice the Phake::setReference()
        Phake::when($validator)->validate($data, Phake::setReference($expectedErrors)-
↪>when([])->thenReturn(false);

        //Instantiate the SUT
        $validationLogger = new ValidationLogger($validator, $log);

        //verify the validation is false and the message is logged
        $errors = [];
        $this->assertFalse($validationLogger->validate($data, $errors));
        Phake::verify($log)->info('Validation Error: data1 is not valid');
    }
}
```

Please note, when you are using `Phake::setReference()` you still must provide an answer for the stub. If you use this function and your reference parameter is never changed, that is generally the most common reason.

4.7 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.

```
class MyClass
{
    private $value;

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

(continues on next page)

(continued from previous page)

```
    }

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

Using `Phake::partialMock()` 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::partialMock()`.

```
class MyClassTest extends PHPUnit\Framework\TestCase
{
    public function testCallingParent()
    {
        $mock = Phake::partialMock(MyClass::class, 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 for is not doing too much.

4.8 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.

```
class MyClassTest extends PHPUnit\Framework\TestCase
{
    public function testDefaultStubs()
    {
        $mock = Phake::mock(MyClass::class, Phake::ifUnstubbed()->thenReturn(42));

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

4.9 Stubbing Magic Methods

Most magic methods can be stubbed 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 stub a particular invocation of `__call()` you can create a stub for the method you are targetting in the first parameter to `__call()`.

Consider the following class.

```
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.

```
class MagicClassTest extends PHPUnit\Framework\TestCase
{
    public function testMagicCall()
    {
        $mock = Phake::mock(MagicClass::class);

        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()`. The matchers passed to `Phake::whenCallMethod()` will be matched to the method name and array of arguments similar to what you would expect to be passed to a `__call()` method. You can also use Phake::anyParameters() instead.

```
class MagicClassTest extends PHPUnit\Framework\TestCase
{
    public function testMagicCall()
    {
        $mock = Phake::mock(MagicClass::class);

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

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

CHAPTER 5

Method Verification

The `Phake::verify()` method is used to assert that method calls have been made on a mock object that you can create with `Phake::mock()`. `Phake::verify()` accepts the mock object you want to verify calls against. Mock objects in Phake can almost be viewed as a tape recorder. Any time the code you are testing calls a method on an object you create with `Phake::mock()` it is going to record the method that you called along with all of the parameters used to call that method. Then `Phake::verify()` will look at that recording and allow you to assert whether or not a certain call was made.

```
class PhakeTest1 extends PHPUnit\Framework\TestCase
{
    public function testBasicVerify()
    {
        $mock = Phake::mock(MyClass::class);

        $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 laid 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.

5.1 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 are discussed later on in *Method Parameter Matchers*.

5.2 Verifying Multiple Invocations

A common need for mock objects is the ability to have 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.

```
class MyTest extends PHPUnit\Framework\TestCase
{
    public function testPHPUnitMock()
    {
        $mock = $this->getMock(PhakeTest_MockedClass::class);

        $mock->expects($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 single 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.

```
class MyTest extends PHPUnit\Framework\TestCase
{
    public function testPHPUnitMock()
    {
        $mock = $this->getMock(PhakeTest_MockedClass::class);

        //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.

```
class MyTest extends PHPUnit\Framework\TestCase
{
    public function testPHPUnitMock()
    {
        $mock = Phake::mock(PhakeTest_MockedClass::class);

        $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:

- `Phake::times($n)` – Where `$n` equals the exact number of times you expect the method to be called.
- `Phake::atLeast($n)` – Where `$n` is the minimum number of times you expect the method to be called.
- `Phake::atMost($n)` – Where `$n` is the most number of times you would expect the method to be called.
- `Phake::never()` - Same as calling `Phake::times(0)`.

Here is an example of this in action.

```
class MyTest extends PHPUnit\Framework\TestCase
{
    public function testPHPUnitMock()
    {
        $mock = Phake::mock(PhakeTest_MockedClass::class);

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

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

5.3 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()`.

```
class MyTest extends PHPUnit\Framework\TestCase
{
    public function testPHPUnitMock()
    {
        $mock = Phake::mock(PhakeTest_MockedClass::class);

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

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

5.4 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. You can pass multiple arguments to this method to verify no interaction with multiple mock objects.

5.5 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)`. You can pass multiple arguments to this method to verify no further interaction occurs with multiple mock objects.

5.6 Verifying No Unverified Interaction with a Mock

By default any unverified calls to a mock are ignored. That is to say, if a call is made to *$mock->foo()* but *Phake::verify($mock)->foo()* is never used, then no failures are thrown. If you want to be stricter and ensure that all calls have been verified you can call *Phake::verifyNoOtherInteractions($mock)* at the end of your test. This will check and make sure that all calls to your mock have been verified by one or more calls to Phake verify. This method should only be used in those cases where you can clearly say that it is important that your test knows about all calls on a particular object. One useful case for instance could be in testing a method that returns a filtered array.

```
class FilterTest {
    public function testFilteredList()
    {
        $filter = new MyFilter();
        $list = Phake::Mock(MyList::class);

        $filter->addEvenToList([ 1, 2, 3, 4, 5 ], $list);

        Phake::verify($list)->push(2);
```

(continues on next page)

(continued from previous page)

```
        Phake::verify($list)->push(4);

        Phake::verifyNoOtherInteractions($list);
    }
}
```

Without *Phake::verifyNoOtherInteractions($list)* you would have to add additional verifications that *$list->push()* was not called for the odd values in the list. This method should be used only when necessary. Using it in every test is an anti-pattern that will lead to brittle tests.

5.7 Verifying Magic Methods

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 invocation of `__call()` you can verify the actual method call by mocking the method passed in as the first parameter.

Consider the following class.

```
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.

```
class MagicClassTest extends PHPUnit\Framework\TestCase
{
    public function testMagicCall()
    {
        $mock = Phake::mock(MagicClass::class);

        $mock->magicCall();

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

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

```
class MagicClassTest extends PHPUnit\Framework\TestCase
{
    public function testMagicCall()
    {
        $mock = Phake::mock(MagicClass::class);

        $mock->magicCall(42);

        Phake::verifyCallMethodWith('magicCall', [ 42 ])->isCalledOn($mock);
    }
}
```

CHAPTER 6

Mocking Static Methods

Phake can be used to verify as well as stub polymorphic calls to static methods. It is important to note that you cannot verify or stub all static calls. In order for Phake to record or stub a method call, it needs to intercept the call so that it can record it. Consider the following class

```
class StaticCaller
{
    public function callStaticMethod()
    {
        Foo::staticMethod();
    }
}
```

You will not be able to stub or verify the call to Foo::staticMethod() because the call was made directly on the class. This prevents Phake from seeing that the call was made. However, say you have an abstract class that has an abstract static method.

```
abstract class StaticFactory
{
    protected static function factory()
    {
        // ...
    }

    public static function getInstance()
    {
        return static::factory();
    }
}
```

In this case, because the `static::` keyword will cause the called class to be determined at runtime, you will be able to verify and stub calls to StaticFactory::factory(). It is important to note that if self::factory() was called then stubs and verifications would not work, because again the class is determined at compile time with the self:: keyword. The key thing to remember with testing statics using Phake is that you can only test statics that leverage Late Static Binding: http://www.php.net/manual/en/language.oop5.late-static-bindings.php

The key to testing static methods using Phake is that you need to create a "seam" for your static methods. If you are not familiar with that term, a seam is a location where Phake is able to override and intercept calls to your code. The typical seem for Phake is a parameter that allows you to pass your object. Typically you would pass a real object, however during testing you pass in a mock object created by Phake. This is taking advantage of an instance seam.

Thankfully in php now you can do something along the lines of $myVar::myStatic() where if $myVar is a string it resolves as you would think for a static method. The useful piece though is that if $myVar is an object, it will resolve that object down to the class name and use that for the static.

So, the general idea here is that you can take code that is in class Foo:

```
class Foo
{
    public function doSomething()
    {
        // ... code that does stuff ...
        Logger::logData();
    }
}
```

which does not provide a seam for mocking Logger::logData() and provide that seem by changing it to:

```
class Foo
{
    public $logger = 'Logger';
    public function doSomething()
    {
        // ... code that does stuff ...
        $logger = $this->logger;
        $logger::logData($data);
    }
}
```

Now you can mock logData as follows:

```
class FooTest
{
    public function testDoSomething()
    {
        $foo = new Foo();
        $foo->logger = Phake::mock(Logger::class);
        $foo->doSomething();
        Phake::verifyStatic($foo->logger)->logData(Phake::anyParameters());
    }
}
```

Phake has alternative methods to handle interacting with static methods on your mock class. `Phake::mock()` is still used to create the mock class, but the remaining interactions with static methods use more specialized methods. The table below shows the Phake methods that have a separate counterpart for interacting with static calls.

Instance Method	Static Method
`Phake::when()`	`Phake::whenStatic()`
`Phake::verify()`	`Phake::verifyStatic()`
`Phake::verifyCallMethodWith()`	`Phake::verifyStaticCallMethodWith()`
`Phake::whenCallMethodWith()`	`Phake::whenStaticCallMethodWith()`
`Phake::reset()`	`Phake::resetStatic()`

If you are using Phake to stub or verify static methods then you should call `Phake::resetStaticInfo()` in the the `tearDown()` method. This is necessary to reset the stubs and call recorder for the static calls in the event that the mock class gets re-used.

CHAPTER 7

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.

7.1 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.

```
class MyClass
{
    private Logger $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.

```
class MyClassTest extends PHPUnit\Framework\TestCase
{
    public function testProcessSomeDataLogsExceptions()
    {
        $logger = Phake::mock(Logger::class);
        $data = Phake::mock(MyData::class);
        $processor = Phake::mock(MyDataProcessor::class);

        Phake::when($processor)->process($data)->thenThrow(new Exception('My error␣
↪message!'));

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

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

7.2 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.

```
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.

```
class MyClassTest extends PHPUnit\Framework\TestCase
{
    public function testCallingParent()
    {
        $mock = Phake::mock(MyClass::class);
        Phake::when($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.

7.3 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.

7.4 Answer Callbacks

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 callback answer. These do generally increase the complexity of tests and you really should only use them if you won't know what you need to return until call time.

You can specify a callback answer using the thenReturnCallback method. This argument takes a callback or a closure. The callback will be passed the same arguments as were passed to the method being stubbed. This allows you to use them to help determine the answer.

```
class MyClassTest extends PHPUnit\Framework\TestCase
{
    public function testCallback()
    {
        $mock = Phake::mock(MyClass::class);
        Phake::when($mock)->foo->thenReturnCallback(function ($val) { return $val * 2;
↪ });

        $this->assertEquals(42, $mock->foo(21));
    }
}
```

7.5 Custom Answers

You can also create custom answers. 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.

CHAPTER 8

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:

```
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:

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


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

$dealer->deal($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:

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

8.1 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 seamlessly inside of Phake. Here is an example of how the PHPUnit constraints can be used:

```
class TestPHPUnitConstraint extends PHPUnit\Framework\TestCase
{
    public function testDealNumberOfCards()
    {
        $deck = Phake::mock(CardCollection::class);
        $players = Phake::mock(PlayerCollection::class);

        $dealer = Phake::mock(DealerStrategy::class);
        $dealer->deal($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 PHPUnit's documentation on assertions and constraints. Any constraint that can be used with `assertThat()` in PHPUnit can also be used in Phake.

8.2 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.

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

(continues on next page)

(continued from previous page)

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

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

8.3 Wildcard Parameters

Frequently when stubbing methods, you do not really care about matching parameters. Often times matching every parameter for a stub can result in overly brittle tests. If you find yourself in this situation you can use Phake's shorthand stubbing to instruct Phake that a mock should be stubbed on any invocation. You could also use it to verify a method call regardless of parameters. This is not a very common use case but it is possible.

To specify that a given stub or verification method should match any parameters, you call the method you are stubbing or mocking as a property of `Phake::when()` or `Phake::verify()`. The code below will mock any invocation of $obj->foo() regardless of parameters to return bar.

```
class FooTest extends PHPUnit\Framework\TestCase
{
    public function testAddItemsToCart()
    {
        $obj = Phake::mock(MyObject::class);

        Phake::when($obj)->foo->thenReturn('bar');

        $this->assertEquals('bar', $obj->foo());
        $this->assertEquals('bar', $obj->foo('a parameter'));
        $this->assertEquals('bar', $obj->foo('multiple', 'parameters'));
    }
}
```

If you are familiar with `Phake::anyParameters()` then you will recognize that the shorthand functionality is really just short hand of `Phake::anyParameters()`. You can still use `Phake::anyParameters()` but it will likely be deprecated at some point in the future.

8.3.1 Default and Variable Parameters

Wildcards can also come in handy when stubbing or verifying methods with default parameters or variable parameters. In addition to `Phake::anyParameters()`, `Phake::ignoreRemaining()` can be used to instruct Phake to not attempt to match any further parameters.

A good example of where this could be handy is if you are mocking or verifying a method where the first parameter is important to stubbing but maybe the remaining parameters aren't. The code below stubs a factory method where the first parameter sets an item's name, but the remaining parameters are all available as defaults.

```
class MyFactory
{
    public function createItem($name, $color = 'red', $size = 'large')
    {
        //...
    }
```

(continues on next page)

(continued from previous page)

```
}

class MyTest extends PHPUnit\Framework\TestCase
{
    public function testUsingItemFactory()
    {
        $factory = Phake::mock(MyFactory::class);

        $factory->createItem('Item1', 'blue', 'small');

        //Verification below will succeed
        Phake::verify($factory)->createItem('Item1', Phake::ignoreRemaining());
    }
}
```

8.4 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.

```
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 dependency injection to instantiation. So please remember, this is not an example of clean design, simply an example of what you can do with argument capturing.

```
class MyPokerGameTest extends PHPUnit\Framework\TestCase
{
    public function testDealCards()
    {
        $dealer = Phake::mock(MyPokerDealer::class);
        $players = Phake::mock(PlayerCollection::class);

        $cardGame = new MyPokerGame($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.

```
class MyBetterPokerGameTest extends PHPUnit\Framework\TestCase
{
    public function testDealCards()
    {
        $dealer = Phake::mock(MyPokerDealer::class);
        $players = Phake::mock(PlayerCollection::class);

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

        Phake::verify($dealer)->deal(
            Phake::capture($deck)
                ->when($this->isInstanceOf(CardCollection::class)),
            $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.

```
Exception: Expected MyPokerDealer->deal(<captured parameter>,
equal to <object:PlayerCollection>) to be called exactly 1
times, actually called 0 times.
Other Invocations:
  PhakeTest_MockedClass->deal(<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 returned. Argument capturing in no way helps with that goal. In the worst case scenario, you will have some incredibly difficult test failures to diagnose.

Beginning in Phake 2.1 you can also capture all values for a given parameter for every matching invocation. For instance imagine if you have a method `$foo->process($eventManager)` that should send a series of events.

```
class Foo
{
    // ...
    public function process(Request $request, EventManager $eventManager)
    {
       $eventManager->fire(new PreProcessEvent($request));
       // ... do stuff
       $eventManager->fire(new PostProcessEvent($request, $result));
    }
}
```

If you wanted to verify different aspects of the `$eventManager->fire()` calls this would have been very difficult and brittle using standard argument captors. There is now a new method `Phake::captureAll()` that can be used to capture all otherwise matching invocations of method. The variable passed to `Phake::captureAll()` will be set to an array containing all of the values used for that parameter. So with this function the following test can be

written.

```
class FooTest
{
    public function testProcess()
    {
        $foo = new Foo();
        $request = Phake::mock(Request::class);
        $eventManager = Phake::mock(EventManager::class);

        $foo->process($request, $eventManager);

        Phake::verify($eventManager, Phake::atLeast(1))->fire(Phake::captureAll(
↪$events));

        $this->assertInstanceOf(PreProcessEvent::class, $events[0]);
        $this->assertEquals($request, $events[0]->getRequest());

        $this->assertInstanceOf(PostProcessEvent::class, $events[1]);
        $this->assertEquals($request, $events[1]->getRequest());
    }
}
```

8.5 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.

```
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::class);
        $players = Phake::mock(PlayerCollection::class);

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

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

CHAPTER 9

Configuration

There are some options you can use to configure and customize Phake. None of these options are required and Phake can always just be used straight out of the box, however some configuration options are available to provide more convenient integration with PHPUnit and ability to debug your mock objects.

9.1 Setting the Phake Client

While Phake does not have a direct dependency on PHPUnit, there is a PHPUnit specific client that improves error reporting and allows you to utilize strict mode with PHPUnit. Without using the PHPUnit client, any failed verifications will result in an errored test. Generally speaking, with PHPUnit, the error result is reserved for bad tests, not failed tests.

The other issue you would run into when using Phake with PHPUnit without using the PHPUnit Phake client is that any test runs utilizing the –strict flag will fail when an assertion is not recorded. By default Phake does not register assertions with PHPUnit. When the PHPUnit client is used however, the assertions are recorded and –strict mode can be safely used with your tests.

To enable the PHPUnit Phake client, you can register it in your test bootstrap.

```
require_once('Phake.php');
Phake::setClient(Phake::CLIENT_PHPUNIT);
```

9.2 Setting the Mock Class Loader

When generating mock classes, Phake will load them into memory utilizing the PHP `eval()` function. This can make the code inside of mock classes difficult to debug or diagnose when errors occur in this code. Using the `Phake::setMockLoader()` method you can change this behavior to instead dump the generated class to a file and then require that file. This will allow for accurate and easily researchable errors when running tests. This shouldn't typically be required for most users of Phake, however if your are having errors or working on code for Phake itself it can be incredibly useful.

`Phake::setMockLoader()` accepts a single parameter of type `Phake\ClassGenerator\ILoader`. The default behavior is contained in the `Phake\ClassGenerator\EvalLoader` class. If you would instead like to dump the classes to files you can instead use the `Phake\ClassGenerator\FileLoader` class. The constructor accepts a single parameter containing the directory you would like to dump the classes to. The classes will be stored in files with the same name as the generated class.

Below is an example of the code required to dump mock classes into the /tmp folder.

```
require_once('Phake.php');
require_once('Phake/ClassGenerator/FileLoader.php');
Phake::setMockLoader(new Phake\ClassGenerator\FileLoader('/tmp'));
```

CHAPTER 10

Indices and tables

- genindex
- modindex
- search