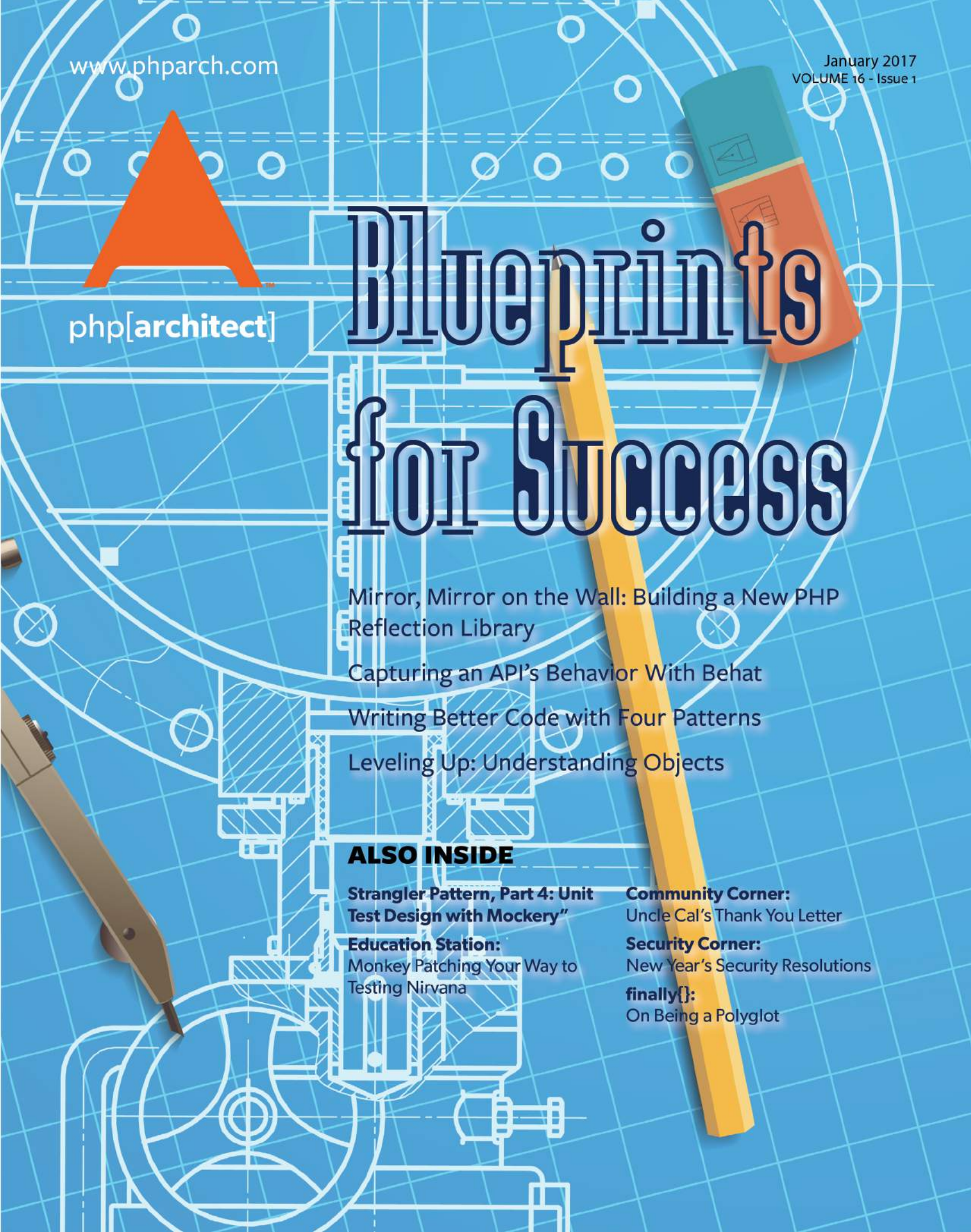
www.phparch.com

January 2017
VOLUME 16 - Issue 1



php[architect]

Blueprints for Success

Mirror, Mirror on the Wall: Building a New PHP Reflection Library

Capturing an API's Behavior With Behat

Writing Better Code with Four Patterns

Leveling Up: Understanding Objects

ALSO INSIDE

Strangler Pattern, Part 4: Unit Test Design with Mockery"

Education Station:
Monkey Patching Your Way to Testing Nirvana

Community Corner:
Uncle Cal's Thank You Letter

Security Corner:
New Year's Security Resolutions

finally{}:
On Being a Polyglot

PHP[TEK] 2017

The Premier PHP Conference
12th Annual Edition

May 24-26 — ATLANTA

Keynote Speakers:

Gemma Anible
WonderProxy

Keith Adams
Slack

Alena Holligan
Treehouse

Larry Garfield
Platform.sh

Mitch Trale
PucaTrade

Sponsored By:

php women

Open Sourcing Mental Illness — Symbolica Than Fear

Save $200 on tickets
Buy ***before*** Feb 18th

tek.phparch.com

php[architect]

CONTENTS

JANUARY 2017
Volume 16 - Issue 1

Blueprints for Success

Features

Columns

Subscriptions
Print, digital, and corporate subscriptions are available. Visit https://www.phparch.com/magazine to subscribe or email contact@phparch.com for more information.

Advertising
To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

Managing Partners
Kevin Bruce, Oscar Merida, Sandy Smith

php[**architect**] is published twelve times a year by:
musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Strangler Pattern, Part 4: Unit Test Design with Mockery

Edward Barnard

When your PHP code must work through other classes, functions, APIs, and databases, those dependencies become a formidable challenge to writing your unit tests. You may find yourself spending an hour getting structures set up for a three-line test. Things can easily get out of hand.

In this article, we introduce Mockery, a drop-in replacement for PHPUnit's built-in mocking library. We introduce and demonstrate strategies to use in keeping your unit test development sane. We cover spies, mocks, and expectations.

We are using the *Strangler Pattern* as our guide for scaling-out our web application at InboxDollars. Our approach is to offload some of the processing. We're moving some of the workload away from the member-facing web servers to a back-end system and planning for a 10X increase in member views.

The *Strangler Pattern* allows us to scale out with a hybrid architecture. Our monolithic web application continues to run as is. We are designing our new back-end processing resources as microservices. We put this together as a distributed messaging system called *BATS (Batch System)* using RabbitMQ[1]. We chose RabbitMQ based on developer advice and because it has enterprise-level support.

Let's write some code; remember, we're doing Test-driven Development (TDD).

Being able to start and stop things seems like a good place to begin. But first, we need the things to start and stop. Those things run by receiving messages via RabbitMQ. Thus, we need to write the code which receives messages. To receive messages, we need to write the code which connects to RabbitMQ which works with exchanges and queues. To receive messages we need to configure the exchanges, create the queues, and bind the queue to the exchange.

To have a message to receive, we first need to put the message into the exchange. For RabbitMQ to be able to route the message from the exchange to the queue, our design requires a consumer first to define the queue.

Where do we start? That's where Mockery[2] comes in. With Mockery, it doesn't matter where you start or what you attack next. You can start anywhere!

Mockery creates mock objects. Wikipedia[3] does not help much:

> *Mock objects are simulated objects that mimic the behavior of real objects in controlled ways.*

While true, it doesn't explain the magic. With mock objects, you can focus on one piece of development and ignore all of those other issues. In any real-world project we will always have countless things to deal with at once, the moment our code first hits production. The ability to start anywhere with Mockery, and to focus on one thing at a time, is quite magical.

Since we are writing code that works with RabbitMQ, we need to deal with connections to RabbitMQ, communication channels, exchanges, queues, routing keys, bindings, and so on. At first glance, this means we can't test *any* code until we write *all* the code.

With Mockery, it's easy to simply *mock* (simulate or fake) everything else and focus on writing the code for one thing. Simply pick a spot and start developing your tests and code.

In Part Three: The Rhythm of Test-Driven Development[4] we used *Learning Tests* to learn how to use a third-party library in whatever way we intend to use it. We need to learn both Mockery and RabbitMQ. We won't develop the learning tests here, but you should note:

1 *RabbitMQ:* https://www.rabbitmq.com

2 *Mockery:* https://github.com/padraic/mockery

3 *Wikipedia:* https://en.wikipedia.org/wiki/Mock_object

4 *Part Three: The Rhythm of Test-Driven Development:* https://www.phparch.com/magazine/

- RabbitMQ has an excellent set of tutorials online: http://www.rabbitmq.com/getstarted.html
- Mockery, likewise, has excellent documentation online: http://docs.mockery.io/en/latest/
- Jeffrey Way's *Laravel Testing Decoded*[5] has a chapter on Mockery. It's the best Mockery tutorial I've ever encountered, and you don't need to know anything about Laravel to follow it. The book as a whole is about TDD using Laravel

Julie Andrews showed the way half a century ago in the movie *Sound of Music*: Let's start at the very beginning—a very good place to start.

> *When you read, you begin with A-B-C. When you sing, you begin with do-re-mi.*

What does this mean when building messaging systems?

1. To process a message, you need to receive the message.
2. To receive the message, the message must have been sent.
3. To send the message, you need to create the message.
4. When creating, sending, and receiving the message, it will calm the chaos if you have a predictable message format.

In other words, we need to begin by creating a *Canonical Data Model:*[6]

> *How can you minimize dependencies when integrating applications that use different data formats? Design a Canonical Data Model that is independent from any specific application. Require each application to produce and consume messages in this common format.*

Sure, all we are *really* doing is passing arrays around as a JSON-encoded string. What we *proclaim* we are doing is constructing messages using a language-independent *Canonical Data Model* which can communicate with any system able to connect to our RabbitMQ server.

Since our development team already uses JSON for data transmission in various contexts, our *Canonical Data Model* is merely formalizing our existing practice.

BATS Message Class

To me, it makes sense to separate metadata from the payload data. For example, suppose we are crediting a member for completing some action on the website:

- The accounting information would be contained in the message payload
- Timestamps, message identifier, routing/origin information, etc., could be metadata

All of our developers are familiar with the "head" and "body" sections of an HTML page. Let's construct a class, `BatsMessage`, which follows a similar idea, able to store "head" (metadata) and "body" (payload) sections. The class can serialize and unserialize its contents to create the JSON-encoded string for communication via RabbitMQ.

New Project

We begin coding by creating a new project. Create the new project `DemoMockery` the same way we did in *Part Three: Rhythm of TDD*

1. Clone the skeleton package:
   `git clone git@github.com:thephpleague/skeleton.git`.
2. Rename the folder: `mv skeleton DemoMockery`.
3. Then, tell PhpStorm to "create a new project from existing files."
4. Run the `prefill.php` script to customize your packages with author name, etc.: `php prefill.php`.
5. Remove the `prefill.php` script as instructed.
6. To install Mockery and its dependencies run:
   `composer require ---dev mockery/mockery`
7. Get a clean run from PHPUnit. In my case, to run it on my Mac, I invoke it via PhpStorm with a special `php.ini` file to load the `xdebug` extension.

Your initial run should report `OK (1 test, 1 assertion)`. Delete `src/SkeletonClass.php` and `tests/ExampleTest.php`, but note the namespace and extended class.

First Test

We begin by making sure we can construct a class of the correct type as in Listing 1.

LISTING 1

```
01. <?php
02. namespace ewbarnard\DemoMockery;
03.
04. class BatsMessageTest extends \PHPUnit_Framework_TestCase
05. {
06.     public function testConstructor() {
07.         $target = new BatsMessage;
08.         static::assertInstanceOf(BatsMessage::class, $target);
09.     }
10. }
```

Run the tests. As expected, we see:

```
PHP Fatal error: Class 'ewbarnard\DemoMockery\BatsMessage' not
found
```

5 *Jeffrey Way's Laravel Testing Decoded:* https://www.amazon.com/dp/B00D8O19O6

6 *Canonical Data Model::* https://www.amazon.com/dp/0321200683/

Create the class:

```
<?php

namespace ewbarnard\DemoMockery;

class BatsMessage {
}
```

We're green, that is, all tests pass: `OK (1 test, 1 assertion)`. I nearly always start out a test suite by checking the constructor. This step ensures I have my tests hooked up correctly. *Always* observe the test failing before making it pass. This guarantees the test really has run.

Explore the API

The `head()` method should return an array, as should `body()`. Write the tests, and write the simplest thing possible to pass the test.

```
public function testHeadReturnsArray() {
   $target = new BatsMessage;
   $head = $target->head();
   static::assertInternalType('array', $head);
}
```

The simplest thing possible to pass the test:

```
public function head() {
   return [];
}
```

The `body()` test and function are nearly identical (at this point). With all tests passing, we can refactor the test to remove duplication (see Listing 2).

LISTING 2

```
01. <?php namespace ewbarnard\DemoMockery;
02.
03. class BatsMessageTest
04.    extends \PHPUnit_Framework_TestCase
05. {
06.    /** @var BatsMessage */
07.    protected $target;
08.
09.    public function setUp() {
10.       $this->target = new BatsMessage;
11.    }
12.
13.    public function testConstructor() {
14.       static::assertInstanceOf(
15.          BatsMessage::class, $this->target
16.       );
17.    }
18.
19.    public function testHeadReturnsArray() {
20.       $head = $this->target->head();
21.       static::assertInternalType('array', $head);
22.    }
23.
24.    public function testBodyReturnsArray() {
25.       $body = $this->target->body();
26.       static::assertInternalType('array', $body);
27.    }
28. }
```

Now you can better see why I always begin a test suite with the `testConstructor()` test. It's my safety net ensuring I didn't break anything when refactoring the test setup.

We continue to develop and flesh out the `BatsMessage` class. It's very rapid, and we won't show it here.

What's the point? `BatsMessage` is a small class. It doesn't do much except store and serialize a couple of arrays. Does it merit writing an entire suite of unit tests?

Yes it does! We are building a distributed messaging system. The message itself is, obviously, central to everything. The message structure even has a pattern name, *Canonical Data Model*. So, yes, this class does merit the unit-test treatment.

The unit tests also serve as "executable documentation." If anyone needs to integrate with our BATS system, they can examine the `BatsMessage` test suite to easily understand the developer's original intent. TDD mean *all* intended capabilities are demonstrated by the test suite. Finally, if anything happens which breaks the `BatsMessage` class, this test suite will inform us the next time it's run.

Four-Phase Test

xUnit Test Patterns: Refactoring Test Code[7] by Gerard Meszaros (p. 358) explains:

> *How do we structure our test logic to make what we are testing obvious? We structure each test with four distinct parts executed in sequence:*
>
> 1. *Setup*
> 2. *Exercise*
> 3. *Verify*
> 4. *Teardown*

Meszaros continues:

> *We should avoid the temptation to test as much functionality as possible in a single Test Method [p. 348] because that can result in Obscure Tests [p. 186]. In fact, it is preferable to have many small Single-Condition Tests [p. 45]. Using comments to mark the phases of a Four-Phase Test is a good source of self-discipline, in that it makes it very obvious when our tests are not Single-Condition Tests.*
>
> *It will be self-evident if we have multiple exercise SUT (system under test) phases separated by result verification phases or if we have interspersed fixture setup and exercise SUT phases. Sure, these tests may work—but they will provide less Defect Localization [p. 22] than if we have a bunch of independent Single-Condition Tests.*

7 *xUnit Test Patterns: Refactoring Test Code:* https://www.amazon.com/dp/0131495054

The sequence hasn't been obvious in our examples thus far, but, if you *know* the pattern you can see this pattern present in all tests.

Spies And Mockery

Real code has interdependencies. Sure, when "we start at the very beginning," we have no dependencies. That's easy. But, later code *does* have dependencies. To continue writing tests to exercise our code we use spies and mock objects. We'll look at the code first and think backward to how we might have tested it. And then, don't worry, we *will* test it.

For this article we are focusing on a single interdependency, namely the connection between the CakePHP 3 framework and our BATS code. The connection (that is, the interdependency) is in `BatsCommon`*. BatsCommon is an abstract base class containing most of the "glue" code between BATS and RabbitMQ. We're not showing any of the "glue" here.*

What is a spy? A spy allows you to observe or verify the internal state of the SUT (system under test). In our example, the spy will be a child class which spies on its parent class. The child (spy) class is test code, and the parent (real) class is production code.

What is a mock object? A mock object also lets you observe or verify the internal state of the SUT. The difference is the mock object lets you to set up your expectations before the test, and the mock object verifies that each expectation was met. The Spy, by contrast, opens up a backdoor into your production code, allowing your tests to poke around as needed.

CakePHP 3 supports "verbose" output[8] which is only sent to the console when `--verbose` was specified on the command line, with its `Shell::verbose()` method.

Most of the BATS library is "plain PHP," that is, not specifically part of the CakePHP ecosystem. It's helpful to hook into CakePHP's console output functions, so I built this into the base class `BatsCommon` (see Listing 3). The caller passes `$this` into the `BatsCommon` constructor which provides us access to `verbose()`.

Consider the list of tests which should have gotten us here:

- If nothing is passed into the constructor, the set of parameters is an empty array.
- Anything passed into the constructor is available as an array key of `$params` property.
- If `caller` is passed in, it is available as `$this->caller`.
- `verbose()` must be called with at least one parameter.
- The second `verbose()` parameter defaults to integer 1.
- When `caller` is passed into the constructor, `verbose` calls it with both parameters.

8 *"verbose" output:* http://phpa.me/cake-console-output

LISTING 3

```
01. <?php
02. namespace ewbarnard\DemoMockery;
03.
04. class BatsCommon
05. {
06.    protected $parms = [];
07.
08.    /** @var mixed */
09.    protected $caller;
10.
11.    public function __construct(array $parms = []) {
12.       $this->parms = $parms;
13.       if (array_key_exists('caller', $parms)) {
14.          $this->caller = $parms['caller'];
15.       }
16.    }
17.
18.    protected function verbose($message, $lines = 1) {
19.       if ($this->caller) {
20.          $this->caller->verbose($message, $lines);
21.       }
22.    }
23. }
```

- When `caller` is not passed into the constructor, `verbose` does *not* call `caller`.

First Test

Our first test, shown in Listing 4, checks the constructor. We'll use `setUp()` and use `testConstructor()` to ensure `setUp()` was indeed executed correctly. This becomes more important as we switch to using a spy.

All tests pass. How do we test that parameters are correctly passed into the constructor, given they are stored in a protected property? We use a *spy*, see Listing 5. Place the spy in the `test` folder, not in the `src` folder. It is *not* part of your production code.

LISTING 4

```
01. <?php
02. namespace ewbarnard\DemoMockery;
03.
04. class BatsCommonTest extends \PHPUnit_Framework_TestCase
05. {
06.
07.    /** @var BatsCommon */
08.    protected $target;
09.
10.    public function setUp() {
11.       $this->target = new BatsCommon();
12.    }
13.
14.    public function testConstructor() {
15.       static::assertInstanceOf(
16.          BatsCommon::class, $this->target
17.       );
18.    }
19. }
```

Test Spy *[Meszaros, p. 538]*

How do we implement Behavior Verification? How can we verify logic independently when it has indirect outputs to other software components?

We use a Test Double to capture the indirect output calls made to another component by the SUT for later verification by the test… A key indication for using a Test Spy is having an Untested Requirement [p. 268] caused by an inability to observe the side effects of invoking methods on the SUT. Test Spies are a natural and intuitive way… that gives the Test Method access to the values recorded during the SUT execution.

Now, let's test the parameter-passing mechanism.

```
public function testParms() {
   $expected = ['a' => 'b', 'c' => 3];
   $this->target = new BatsCommonSpy($expected);
   static::assertSame($expected,
                      $this->target->parms());
}
```

All tests pass. We have duplication; it's time to refactor as shown in Listing 6. Note, it's just as important to refactor the tests and keep *them* as clean as the production code. It's important that future developers be able to understand your tests as quickly and easily as possible.

All tests continue to pass.

Mock Object

You won't be surprised to learn that *mock object* [Meszaros, p. 544] is one of the *xUnit Test Patterns*. Don't be put off by the book's 947 pages. You'll "level up" your unit-testing experience *every* time you read a page or even a paragraph out of the book.

It's important to keep our test cases organized. How do we do that? There's a test pattern for that:

Testcase Class per Fixture *[Meszaros, p. 631]*

How do we organize our Test Methods onto Testcase Classes?

We organize Test Methods into Testcase Classes based on commonality of the test fixture.

As the number of Test Methods grows, we need to decide on which Testcase Class [p. 373] to put each Test Method. Our choice of a test organization strategy affects how easily we can get a "big picture" view of our tests. It also affects our choice of a fixture setup strategy.

Using a Testcase Class per Fixture lets us take advantage of the Implicit Setup [p. 424] mechanism provided by the Test Automation Framework [page 298].

LISTING 5

```
01. <?php
02. namespace ewbarnard\DemoMockery;
03.
04. class BatsCommonSpy extends BatsCommon
05. {
06.    public function parms() {
07.       return $this->parms;
08.    }
09. }
```

LISTING 6

```
01. <?php
02. namespace ewbarnard\DemoMockery;
03.
04. class BatsCommonTest extends \PHPUnit_Framework_TestCase
05. {
06.    /** @var BatsCommonSpy */
07.    protected $target;
08.
09.    public function setUp() {
10.       $this->build();
11.    }
12.
13.    protected function build(array $parms = []) {
14.       $this->target = new BatsCommonSpy($parms);
15.    }
16.
17.    public function testConstructor() {
18.       static::assertInstanceOf(
19.          BatsCommon::class, $this->target
20.       );
21.    }
22.
23.    public function testParms() {
24.       $expected = ['a' => 'b', 'c' => 3];
25.       $this->build($expected);
26.       static::assertSame($expected, $this->target->parms());
27.    }
28. }
```

In other words, when your test setup changes, start a new test class (and file). The *Implicit Setup* mentioned above is simply PHPUnit's built-in `setUp()` function.

Our next test requires we set up a mock object. This is our clue that it's time to create a new test class.

Our list of tests:

- When `caller` is passed into the constructor, `verbose` calls it with both parameters.
- When `caller` is not passed into the constructor, `verbose` does *not* call `caller`.

Given `verbose()` is a protected method, we need to enhance our spy:

```
public function verboseSpy($message, $lines = 1) {
   $this->verbose($message, $lines);
}
```

Our tests depend on some assumptions. Be sure the following tests remain on our list (or have already been written):

- If `caller` is passed in, it is available as `$this->caller`.
- If `caller` is *not* passed in, `$this->caller` is null.
- `verbose()` must be called with at least one parameter.
- The second `verbose()` parameter defaults to integer 1.

One great way to *ensure* those tests aren't forgotten is to write empty tests and mark them incomplete. For example (string split for publication):

```
public function testCallerNull() {
   static::markTestIncomplete('If caller not passed '
      . 'in to constructor, $this->caller remains null');
}
```

When we run the tests we have the reminder:

```
There was 1 incomplete test:

1) ewbarnard\DemoMockery\BatsCommonTest::testCallerNull
If caller not passed in to constructor, $this->caller remains
null

OK, but incomplete, skipped, or risky tests!
Tests: 6, Assertions: 5, Incomplete: 1.
```

We can also mark tests as *skipped*. Use *skipped* when tests should not run given the current environment. For example, a framework's MySQL tests should only run when MySQL is available. Otherwise, they should report themselves as *skipped*.

The second of our tests is easy. If `caller` was not passed to the constructor, `$this->caller` should not be referenced as an object when calling `verbose()`. So, we simply call `verbose()`. If nothing blows up, the test passes.

```
public function testNoVerbose() {
   $this->target->verboseSpy('test');
   static::assertTrue(TRUE, 'test blows up otherwise');
}
```

The test passes. Now we need to create a Mock Object [Meszaros, p. 544]:

> *How do we implement Behavior Verification for indirect outputs of the SUT? How can we verify logic independently when it depends on indirect inputs from other software components?*
>
> *We replace an object on which the SUT depends on with a test-specific object that verifies it is being used correctly by the SUT.*

The *Mockery* package makes it ridiculously easy to replace dependencies *and* verify the dependencies were exercised as expected.

Create a new empty test class as in Listing 7 which correctly uses Mockery.

When you use Mockery, remember to include a `tearDown()` method with `m::close()`. Mockery runs its verifications during `m::close()`. Without that call, you're not testing what you thought you were. I use `Mockery` as `m` as a convenience.

LISTING 7

```
01. <?php
02. namespace ewbarnard\DemoMockery;
03.
04. use Mockery as m;
05.
06. class BatsVerboseTest extends \PHPUnit_Framework_
TestCase
07. {
08.    public function tearDown() {
09.       m::close();
10.    }
11. }
```

LISTING 8

```
01. public function testVerbose() {
02.    $caller = m::mock()->makePartial();
03.    $caller->shouldReceive('verbose')
04.           ->once()
05.           ->withArgs(['test', 5])
06.    ;
07.    $parms = ['caller' => $caller];
08.    $target = new BatsCommonSpy($parms);
09.    $target->verboseSpy('test', 5);
10. }
```

All tests pass. Here is what the above code does:

1. `M::mock()` creates a mock object. We don't care about its class. We could have called `m::mock('BatsCommon')` to mock the `BatsCommon` class. PHP must be able to find the class for Mockery to mock it. `makePartial()` tells Mockery to *only* mock those methods named in the upcoming "expectations." Any other method calls pass through to the "real" class being mocked.
2. Sets expectations. This mocked object should get `verbose()` method called exactly once as `verbose('test', 5)`. The test `tearDown()` will verify all expectations were met.
3. Sets the parameter list we will be passing to our `BatsCommon` object constructor.
4. Creates our `BatsCommon` object.
5. Calls `verbose()`. It's a protected method, so we have the spy call the class for us.

The TearDown method calls `m::close()` which verifies that all expectations were met. One thing you should note: if the expectations are *not* met, the PHPUnit output can be confusing. Change the call from `$target->verboseSpy('test', 5);` to `$target->verboseSpy('test', 6);`. In other words, call `verbose()` with the second parameter being 6 rather than 5. PHPUnit spews the following:

```
There was 1 error:

1) ewbarnard\DemoMockery\BatsVerboseTest::testVerbose
BadMethodCallException: Method Mockery_0::verbose() does not
exist on this mock object
```

I found this confusing, given verbose() *does* exist on that mock object. The answer is verbose only "exists" when it is called with precisely the arguments `['test', 5]`. If the method gets called with different arguments, Mockery reports the method does not exist.

We can verify this by removing the `withArgs()` part of the expectation:

```
$caller->shouldReceive('verbose')
      ->once();
```

Now all tests pass. In fact, that's how I debug this situation. I first remove the arguments from the expectation. If the test passes, I know the SUT is not passing the expected arguments. If I am *still* puzzled, I throw an exception at that point in the code which displays the parameter list being used. PHPUnit displays the exception message, and I can usually tell what went wrong.

Either way, the result is likely a bug prevented. However, be careful! It's far too easy to "chase down the rabbit hole," caught up in test set-up dependencies. Remember Will Rogers' advice, "When you find yourself in a hole, stop digging." Step back and consider the situation. You might realize you're on the wrong track. You might decide to "bookmark your location" by marking the test incomplete and then dig in to a different piece of the project.

That's it! Mockery has a lot more capability, but you can go a very long way with `mock()`, `makePartial()`, `shouldReceive()`.

Mocking Protected Methods

Mockery has one more capability I find particularly useful with TDD. It can mock (and therefore allow you to declare expectations for) protected methods. Generally speaking, you should focus testing efforts on your public methods. Sometimes, though, it's best to:

- Quickly write a test which verifies the protected method gets called as expected.
- Move on to the next step.

For example, suppose our code calls `verbose()` based on a certain condition. We can test that our code correctly detects the condition by verifying it calls `verbose()`. Our code is in Listing 9.

All tests pass. Note the differences from our prior mocking example:

- We gave the mock a different variable name. We'll see why below.
- We are mocking the target class, *not* a dependency class. Our dependency is the `verbose()` method *inside* our target class.
- We allow mocking protected methods.

LISTING 9

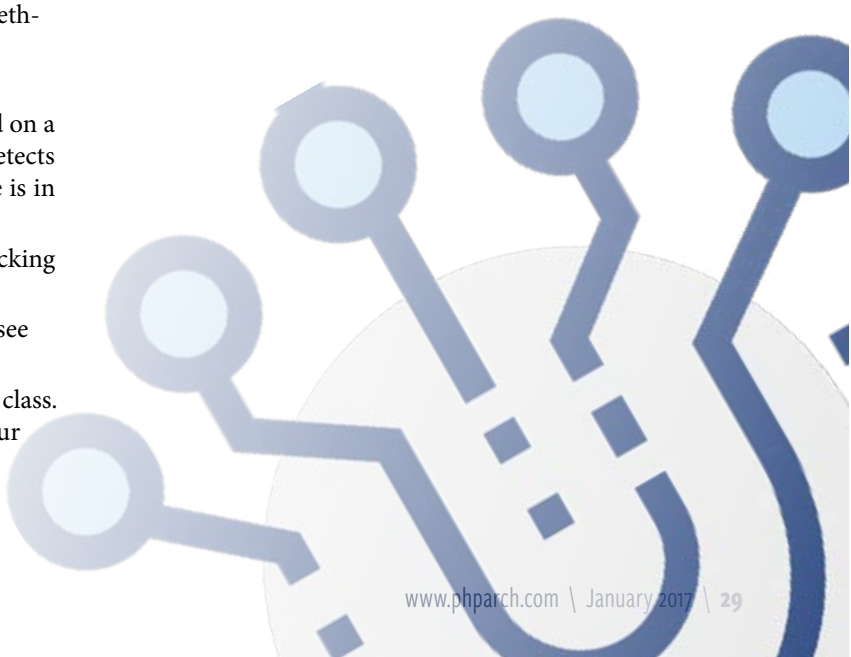
```
01. protected function verbose($message, $lines = 1) {
02.    if ($this->caller) {
03.       $this->caller->verbose($message, $lines);
04.    }
05. }
06.
07. public function doSomething($memberId = 0) {
08.    $memberId = (int)$memberId;
09.    if ($memberId <= 0) {
10.       $this->verbose('Invalid member id encountered');
11.       return;
12.    }
13. }
```

LISTING 10

```
01. public function testDoSomethingZero() {
02.    $mock = m::mock(BatsCommon::class)->makePartial();
03.    $mock->shouldAllowMockingProtectedMethods();
04.    $mock->shouldReceive('verbose')
05.         ->once()
06.         ->withArgs(['Invalid member id encountered'])
07.    ;
08.    /** @var BatsCommon $target */
09.    $target = $mock;
10.    $target->doSomething(0);
11. }
```

- The expectations call looks like the prior setup, except we expect to call with a single argument.
- We flip variables from `$mock` to `$target`. This is so the PhpStorm static analysis tools correctly identify the respective classes involved. `$mock` is a `mock()` result, having the `shouldReceive()` method and friends. We tell PhpStorm `$target` is an instance of our target class, having the `doSomething()` method.

Generally speaking, the need to test protected methods is an indication of design gone wrong. I used to bypass the issue by marking everything public. I've decided it's better to use public/protected as intended.



When you have a lot of complex business logic, that logic needs to go somewhere. When you're doing Test-driven Development and aim to keep your method complexity low, you tend to have a lot of protected methods laying out that business logic.

If you have a lot of protected methods, you may have another class trying to get out. But those protected methods still need to go somewhere. Whether you have two protected methods in class A and two more in class B, you still have four protected methods which *might* play best by being individually tested when you're doing Test-first Development. That won't always be the case, but when it is, I say just mock the protected method and get on with it.

Your most likely alternative is PHPUnit's data provider. You can pass a series of inputs through the public method and verify return values (or use a spy to verify internal state). You can drive your development by adding more and more cases to the data provider: add another use case to the data provider; watch it fail; write the minimum code to make it pass; refactor. That refactor may well involve extracting the new logic to a new protected method.

Summary

There's no way around this: unit tests are tricky because dependencies are tricky to test. Learn the craft and practice, practice, practice. It gets better. It gets smoother. You'll find your judgment more and more reliable. You'll find yourself spending a larger proportion of your time in the red-green-refactor cycle of real development, which can be quite fun. You'll find yourself spending less time debugging in production, which is generally *not* fun.

When you are developing code, your testing target is that *one* method you're working with at the moment. *Everything else*, even that related method ten lines down, is a dependency. Put your target method in "laser focus." If anything else makes it difficult to test, use mocks, spies, and anything else in your arsenal to push those dependencies aside.

Looking ahead, *Part Five: Producer-Consumer Programming in CakePHP/RabbitMQ* brings our case study full circle. We'll see a bit more code, and look at the radically different way of thinking that gets us there.

Ed Barnard began his career with CRAY-1 serial number 20, working with the operating system in assembly language. He's found that at some point code is code. The language and details don't matter. It's the craftsmanship that matters, and that craftsmanship comes from learning and teaching. He does PHP and MySQL for InboxDollars.com. @ewbarnard

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[**architect**]

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe