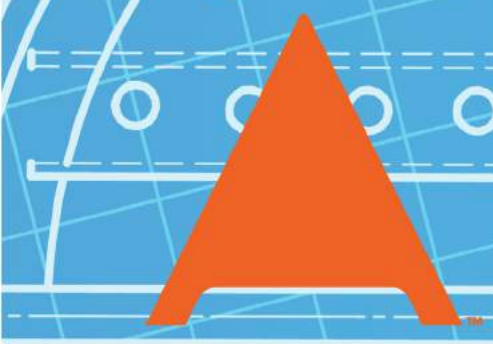
www.phparch.com

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 · Changing Tech Culture

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

Blueprints for Success

Happy New Year, dear reader! Let's get ready to take on the challenges of the next 12 months. As always, a new year brings a sense of a "fresh slate" for our personal and professional goals. This month, we've collected articles that will help you in planning before you dive into your code editor.

When I first started programming professionally, at the turn of the century, writing code and seeing if it would work was paramount. I'm certain many new developers worked the same way. In fact, writing a few lines of code, refreshing your browser, and seeing if it worked—along with the accompanying rush—is what hooked me on this career in the first place. Unfortunately, that's not a sustainable habit, even though managers and others will love you if you're just churning out features and marking tickets resolved. That is, as long as your not causing any collateral damage and creating unnecessary work for your coworkers. At some point, if you haven't already, you'll realize a little planning and learning how others have solved a problem can go a long way. You'll see failure cases you may not have considered, or you'll find an approach that is easier to maintain or scales faster.

This month, our articles offer up various "Blueprints for Success." These may be well-known design patterns, fundamental concepts, or novel applications of a familiar tool. First of, Joseph Maxwell helps in *Writing Better Code with Four Patterns*. Design patterns have been discussed by PHP practitioners for ages now. He takes a look at four useful ones to consider: dependency injection, factory method, strategy, and chain of responsibility. Next, in *Capturing an API's Behavior With Behat*, Michael Heap shows how to use Behat to document how legacy code works. By using "Characterization Tests" you can record exactly how an application works before making changes or adding features. Ed Barnard has *Strangler Pattern, Part 4: Unit Test Design with Mockery* for us this month. In this installment, he writes about using Mockery to handle dependencies in your unit tests. In *Leveling Up*, David Stockton begins a look at Object-oriented programming and design patterns. This one lays the groundwork to help you in *Understanding Objects*.

Also this month, James Titcumb introduces us to the *Better Reflection* library in *Mirror, Mirror on the Wall: Building a New PHP Reflection Library*. If you've ever needed more power than the built-in PHP Reflection library provides, this is worth a look. It's a handy tool to have in your belt for tricky problems.

In our columns this month, Matthew Setter sets you to *Monkey Patching Your Way to Testing Nirvana* in *Education Station*. He looks at using both some reflection and mocking libraries to test PHP's built-in functions. Cal Evans starts the year with *Uncle Cal's Thank You Letter* in *Community Corner*. Chris Cornutt has some *New Year's Security Resolutions* for you to consider in *Security Corner*. And Eli White also has some advice to take hear in *finaly{}: On Being a Polyglot*.

Write For Us

If you would like to contribute, contact us, and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine.

Visit https://phpa.me/write or contact our editorial team at write@phparch.com and get started!

Stay in Touch

Don't miss out on conference, book, and special announcments. Make sure you're connected with us via email, twitter, and facebook.

- Subscribe to our list: http://phpa.me/sub-to-updates
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download this Issue's Code Package:

http://phpa.me/January2017_code

Capturing an API's Behavior With Behat

Michael Heap

Imagine this—you've just joined a fairly well-established company that does millions of dollars in revenue every year. They've built things in a modular and scaleable way with microservices, always using the right tool for the job. There's a mix of half a dozen different languages across the platform, and you can't wait to get stuck in and start contributing.

So, you load up your first project and look for the unit tests to get an understanding of what the service does before you start making changes. Unfortunately for you, there are none—zero, nada. Not a single test for the entire service. Your coworkers tell you it's fine, to start making changes, and they'll notice if you break something. It's down to you to convince them there's a better way to handle this problem.

Why Do You Need Tests?

Whenever you make a change to a code base, the behavior of that service changes. There can be one of two kinds of change: intended or unintended.

Intended changes are fine as you deliberately made a change to the behavior and the code now behaves as you expect it to. You might have fixed a bug or added a new feature. In this situation, you either update the test to match the new behavior or you write a test to cover the new code that you wrote.

An issue arises when you make a change, and it has unintended consequences. Unintended changes aren't okay; they're side effects. If a change is unintended, it shows there has been a misunderstanding somewhere about how the code behaves and has lead to a change in behavior you weren't expecting. In this situation, you can't change the test as it isn't an intended change. Instead, you need to fix your code, so the test passes again.

What if you don't have any tests to tell you about any possible side effects, though? You don't know what the code is supposed to do, only what it currently does. As it turns out, that's a great starting point. You write tests documenting what the service currently does. These are known as characterization tests, and they act as a change protector—protecting code from unintended changes.

About Characterization Tests

Characterization tests capture the actual behavior of a piece of code. They don't check what the code is supposed to do according to the original spec (that'd be a specification test), but they document what the code actually does.

Having these tests helps developers working with legacy code because they can make sure the code still behaves the same way after their changes. The test is not interested in what the rest of the code does, or why it works that way. They only care that their changes didn't change the behavior of other parts of the service unintentionally. In legacy code, different is dangerous. These tests provide an invaluable notification system for change.

So why characterization tests rather than unit tests? For starters, they're easier to write! All you need to do is identify a pinch point and take it from there. In *Working with legacy code*[1], Michael Feathers defines a pinch point as an area of code which provides a narrow interface to a larger piece of code. By writing tests using that narrow interface we can capture all of the behavior it provides while writing the minimum amount of tests needed. The system ends up being a black box. You have no idea about how it works internally, only that given a set of specific inputs it should return a specific output. This is the behavior we're looking to protect. As you'll see in this article, the tests document existing behavior rather than defining desired behavior. The tool set you use is identical.

1 *Working with legacy code:* https://www.amazon.com/dp/0131177052

Writing Your First Characterization Test

You've inherited a project that takes numbers from a user and performs mathematical operations on them. After a little exploration, you find most of the logic is contained in a class that implements the interface in Listing 1.

LISTING 1

```
01. <?php
02. interface Calculator {
03.     public function add (float $n);
04.     public function subtract (float $n);
05.     public function multiply (float $n);
06.     public function divide (float $n);
07.     public function result () : float;
08.
09.     public static function FromExpression(
10.         string $expression
11.     );
12. }
```

In the code base, you can see some usages of it which look like this:

```
$c = new BasicCalculator();
$c->add(1);
$c->add(5);
$c->subtract(3);
echo $c->result();
```

You can clone the source code and tests used in this article from https://github.com/mheap/characterisation-calculator-demo

You could attempt to write unit tests for every method: one for add, one for subtract, etc. Depending on the implementation of the class, this could be quite tough. Instead, we can consider the unit under test to be the entire calculator class itself. This is perfect as it creates a natural boundary you can use to test. All you need to do is create an instance and run some calculations. The `Calculator` interface is our pinch point, and the `BasicCalculator` implementation is what we'll test.

It's easy to start writing a few exploratory tests for basic behaviors such as 1+1, 2+2, and 10-5, so start there. First, create a new test called `testX` and write some code using your class.

```
class BasicCalculatorTest extends PHPUnit_Framework_TestCase {
   public function testX(){
      $c = new SimpleCalculator();
      $c->add(1);
      $c->add(1);
      $this->assertEquals(null, $c->result());
   }
}
```

Notice when you try and assert that the result of 1+1 is null? You know from the interface that it's expected to return a float, so we're expecting this test to fail.

```
1) BasicCalculatorTest::testX
Failed asserting that 2.0 matches expected null.
```

As expected, it fails and gives us an error message. This message is important as it contains the actual result returned by the class. Now you know what the test is intending to show: one plus one equals two. Although you expected that result, you had no idea what the actual implementation was going to return. Writing this test proves how the implementation behaves. Go ahead and update your test with the information you now have.

```
public function testOnePlusOneEqualsTwo(){
   $c = new BasicCalculator();
   $c->add(1);
   $c->add(1);
   $this->assertEquals(2.0, $c->result());
}
```

By starting with a test named `testX` you remove any biases towards what you're expecting to receive from the implementation. You're testing something, but you're not quite sure what it is yet. Once you have the result, update the test name to reflect the behavior then move on to the next few tests (2+2 and 10-5). Remember, you shouldn't have any expectation around the results. You're asking the implementation how it behaves, not telling it how you expect it to behave.

As well as the standard constructor, the calculator has another public interface which parses a string as a mathematical expression. You should also write some characterization tests for this interface to ensure you've covered the calculator's entire public API. Again, you start by calling it `testX` and asserting that the result is null.

```
public function testX(){
   $c = BasicCalculator::FromExpression("10 / 2");
   $this->assertEquals(null, $c->result());
}
```

Once you've run the test, you can update the assertion and test name with the actual behavior.

```
1) BasicCalculatorTest::testX
Failed asserting that 5.0 matches expected null.
```

Then update it to:

```
public function testFromExpressionTenDivTwoEqualsFive(){
   $c = BasicCalculator::FromExpression("10 / 2");
   $this->assertEquals(5.0, $c->result());
}
```

While writing some exploratory tests, you might notice the calculator doesn't apply the rules of multiplication before addition correctly. For example, while we'd expect this test to pass, it actually fails as it returns 14 rather than 11 as expected.

```
public function testX(){
    $c = BasicCalculator::FromExpression("3 + 4 * 2");
    $this->assertEquals(11.0, $c->result());
}
```

As wrong as it feels to ignore it, you can't change this behavior. You're just documenting the way things currently are. If you write a test which adds one and one together and your method returns twelve you can't say "that's wrong" and fix it. Instead, you write your test to capture the existing behavior and move on, writing more tests as documentation.

Does that sound a bit crazy to you? Sadly, this kind of thing happens all the time. Once you make code publicly available, you make a commitment that its behavior won't change without prior notification. Once the code has been made public, the original requirements are forgotten. "Right" and "wrong" don't matter anymore, there is only "what is."

By writing these tests, you're capturing behavior, not changing it. It may look broken, but someone else may be relying on that broken implementation. If you fix the implementation to match the specification, it may break their code. Once the code is released, it becomes the specification. If the bug you've uncovered is a critical production issue, then raise the issue with your team for evaluation. Fixing it might be the right call, especially if it affects security. If it's something that hasn't caused any issues so far, the correct solution might just be to leave it alone.

Testing HTTP APIs

Let's get back to this service you've inherited at your new job. It's a service that accepts a username and an API key and returns whether it's valid or not. Everyone interacts with it via HTTPS, meaning you have the ideal pinch point to start writing tests against. So long as the service behaves identically to people calling the public HTTP API, it doesn't matter how the internals of the service works. You could even rewrite it in a whole new language if you need!

Though the service will likely be a more complicated than this in real life, the behavior will probably be similar to the code in Listing 2.

Reading the code, we can see there are several different paths through the code. Working out all of the different branches that a user can take and making a note of them is the most important part of writing characterization tests. In this case, there are five routes:

1. Missing username
2. Missing API key
3. Invalid username
4. Invalid API key
5. Successful request

It's relatively trivial to test all of those conditions by hand, but as the service grows in complexity, the time it takes will increase. Also, you can't run manual tests as a part of a continuous integration system. Let's look at writing these tests using Behat[2], a Behavior-driven Development (BDD) framework for PHP.

2 *Behat:* http://behat.org

LISTING 2

```php
<?php

function output($d) {
    echo json_encode($d); die;
}

function error($m, $code=400){
    $codes = [
        400 => 'Bad Request',
        404 => 'Not found'
    ];

    header('HTTP/1.1 '.$code.' '.$codes[$code]);
    output(['error' => $m]);
}

$username = $_GET['username']
         ?? error('username is required');
$key = $_GET['key'] ?? error('key is required');

$validUsers = [
    "michael" => "kangar00s",
    "oscar" => "phparch16"
];

$validKey = $validUsers[$username]
         ?? error('could not find user', 404);

if ($key !== $validKey){
    error('invalid apikey');
}

output(['success' => 'true',
        'user' => ['name' => $username]]);
```

About Behat

Behat is intended to be used as a Behavior-driven Design tool that happens to be able to verify those designs by running code. A typical Behat story, which uses the Gherkin[3] syntax, looks like the following:

```
Given I have 1 "Red Bucket" in my basket
And I have 2 "Large Spades" in my basket
When I add 1 "Red Bucket" in my basket
Then I should see 2 "Red Buckets" in my basket
And I should see 2 "Large Spades" in my basket
```

The test follows a `Given`, `When`, `Then` flow to set up existing state, perform an action, and assert that what you expect happened. Each of those human readable lines will be backed by code which describes what actions to perform.

It turns out the same flow is perfect for testing APIs too. To test your API, you can use the `datasift/behat-extension`[4] package. This extension pulls in Behat for you and provides some extra step definitions to help test HTTP APIs. This is what my `composer.json` looks like:

3 *Gherkin:* https://cucumber.io/docs/reference

4 `datasift/behat-extension`: https://packagist.org/packages/datasift/behat-extension

```
{
    "name": "mheap/auth-service",
    "require-dev": {
        "datasift/behat-extension": "4.*"
    }
}
```

After running `composer install`, it's time to bootstrap Behat and write your first test. You'll need to create a `behat.yml` file in your project's root folder with the following contents to register the extension with Behat. The `base_url` provided is where Behat will look for your application (in this case, I'm running the app on port 8080 using PHP's built in web server):

```
default:
 extensions:
   DataSift\BehatExtension:
     base_url: http://localhost:8080/

 suites:
   default:
     contexts:
       -
'DataSift\BehatExtension\Context\RestContext'
```

Start the built-in service by running `php -S localhost:8080` in the same folder as your `index.php` that bootstraps the application. You can test that this is working by visiting `http://localhost:8080`—you should see the error `{"error":"username is required"}`. Let's write a test for this error condition!

Your First Behat Test

Create a folder named `features` in the same directory as `behat.yml`. This is where Behat will look for tests to run. Inside `features`, create a file called `auth.feature`. This is where your tests will live. Edit `auth.feature` and add the following content:

```
Feature: Auth service
  Scenario: No Username
    When I make a "GET" request to "/"
    Then the response status code should be "400"
    And the "error" property equals "username is \required"
```

Here, you're testing the auth services when no username is provided. When you make a GET request, you expect to receive an HTTP code of 400 with an error that the username is required. You can run this test by executing `./vendor/bin/behat`.

```
$ ./vendor/bin/behat Feature: Auth service

Scenario: No Username When I make a "GET" request to "/"
Then the response status code should be "400" And the
"error" property equals "username is required"

1 scenario (1 passed) 3 steps (3 passed)
```

Behat just made a real HTTP request to your service and verified the response it received contained the correct HTTP code and error message. Work your way through the remaining branches by adding a new scenario underneath the auth feature for each branch. To help you out, you can find my final set of tests in Listing 3.

LISTING 3

```
01. Feature: Auth service
02.
03.   Scenario: No Username
04.     When I make a "GET" request to "/"
05.     Then the response status code should be "400"
06.     And the "error" property equals "username is required"
07.
08.   Scenario: No API Key
09.     When I make a "GET" request to "/?username=foo"
10.     Then the response status code should be "400"
11.     And the "error" property equals "key is required"
12.
13.   Scenario: Invalid Username
14.     When I make a "GET" request to "/?username=bananas&key=foo"
15.     Then the response status code should be "404"
16.     And the "error" property equals "could not find user"
17.   Scenario: Invalid API Key
18.     When I make a "GET" request to "/?username=michael&key=foo"
19.     Then the response status code should be "400"
20.     And the "error" property equals "invalid apikey"
21.
22.   Scenario: Successful login
23.     When I make a "GET" request to "/?username=michael&key=kangar00s"
24.     Then the response status code should be "200"
25.     And the "success" property equals "true"
```

With this set of tests, we can be sure that if the auth service behavior changes (both intentionally and unintentionally) we will be able to catch the change in an automated fashion and make a decision about if the change should be approved or denied.

If you wanted to test multiple values for a single scenario, you could use Behat's table driven test support to achieve your goal. You'll need to change your **Scenario** to a **Scenario Outline** and add an **Examples** table with the required data:

```
Scenario Outline: Successful login
  When I make a "GET" request to "/?username=<name>&key=<key>"
  Then the response status code should be "200"
  And the "success" property equals "true"
  And the "user.name" property equals "<name>"
  Examples:
    | name    | key       |
    | michael | kangar00s |
    | oscar   | phparch16 |
```

Behat will then run this scenario once per row in the `Examples` table. This is a great way to test different input values without repeating the test boilerplate each time.

Working With Databases

As well as providing steps to test RESTful APIs, the Behat extension contains many more useful contexts. One of my most used ones is the `DatabaseContext`. If you register the `DatabaseContext` in `behat.yml` and provide the connection

credentials, your database state can be reset before every scenario. This can be combined with the `RestContext` to test a database driven API. See Listing 4 for an example of how to configure the `DatabaseContext`.

There's not much to the `DatabaseContext`. If you register the context and provide the required details, it'll do all of its work in the background without you needing to do anything else. It currently supports MySQL and SQLite, but as it's all PDO under the hood. It should easily work with any other database you need.

Encapsulating Behavior

While the above tests are functional, they don't represent our intentions. Although we are currently making a `GET` request to `"/"` at the moment, what we're actually trying to do is log in as a user. If we changed the endpoint from `"/"` to `"/check"` all of our tests would need updating in the future. Let's write a custom step to encapsulate that behavior now.

Currently, we load in the `RestContext` and `DatabaseContext` from the DataSift Behat extension. As we're going to be writing a custom step, we need to add the code to our own Behat context.

By convention, any steps for the current test suite should live in a class named `FeatureContext` which lives inside `features/bootstrap`. Create `features/bootstrap/FeatureContext.php` now with the following contents:

```
<?php

use Behat\Behat\Context\Context;
use Behat\Behat\Tester\Exception\PendingException;

class FeatureContext implements Context {
}
```

Next, you'll need to register the context with Behat by editing `behat.yml`. Add `FeatureContext` to the top of your list of available contexts like so:

```
suites:
    default:
        contexts:
            - FeatureContext
            - 'DataSift\BehatExtension\Context\RestContext'
            - 'DataSift\BehatExtension\Context\DatabaseContext'
```

`FeatureContext` should now be available to all of our tests. It's now time to change our feature file to use the language we'd like. Open up `features/auth.feature` and replace the line:

```
When I make a "GET" request to "/"
```

with the line:

```
When I log in as "michael" with the password "demo"
```

Here, we're removing the implementation details and exposing our intent instead. We don't care that it's a `GET` request, we care that we're trying to log in as a user.

Save your changes then run Behat again with `./vendor/bin/behat`. It should say the "default suite has undefined steps." Enter the number which corresponds to

LISTING 4

```
default:
  extensions:
    DataSift\BehatExtension:
      base_url: http://localhost:8080/
      database:
        driver: mysql
        dbname: authmanager
        host: 127.0.0.1
        port: ~
        username: travis
        schema: /path/to/schema.sql
        data: /path/to/data.sql

  suites:
    default:
      contexts:
          - 'DataSift\BehatExtension\Context\RestContext'
          - 'DataSift\BehatExtension\Context\DatabaseContext'
```

LISTING 5

```
<?php
use Behat\Behat\Context\Context;
use Behat\Behat\Tester\Exception\PendingException;
use Behat\Behat\Hook\Scope\BeforeScenarioScope;

class FeatureContext implements Context {

    private $restContext;

    /** @BeforeScenario */
    public function gatherContexts(
        BeforeScenarioScope $scope
    ) {
        $environment = $scope->getEnvironment();
        $this->restContext = $environment->getContext(
            'DataSift\BehatExtension\Context\RestContext'
        );
    }

    /**
     * @When I log in as ":user" with the password ":pass"
     */
    public function iLogInAsWithThePassword($user,
                                            $passw)  {
        $url = "/?";
        if ($user) {
            $url .= '&username='.$user;
        }
        if ($pass) {
            $url .= '&key='.$pass;
        }

        return $this->restContext->iRequest("GET", $url);
    }
}
```

`FeatureContext` and watch as Behat generates an example step for us. Copy and paste the generated definition into the `FeatureContext` class and run your test again. This time you should get a TODO message, saying you should write the pending definition.

Now, let's implement our new step! As we just want to delegate to another class, there's a lot of boilerplate to set up so that our context can access the `RestContext`. Take a look at the entire `FeatureContext` in Listing 5 before we examine it in depth.

A lot is going on there, so let's step through it piece by piece. The first thing we do is import all of the objects we'll need at the top. Next, we define a `gatherContexts` function that needs to run before every scenario. This is how we can get access to the `RestContext` and call methods on it. Finally, we implement our new `When I login` step by constructing the URL we need to call and delegating to the existing `RestContext`. You might notice in the annotation for the `iLogInAsWithThePassword` function, I added quotes around the arguments. This allows us to provide empty arguments and test that the application behaves as intended as well as passing valid and invalid arguments.

You can test the empty argument support by removing the username and password from the first test we edited to generate our step so the username and password parameters are empty strings. Once you have your first test passing, update the rest of your tests to use the new step language. You can also update your table driven test so that the `When` statement contains:

```
When I log in as "" with the password ""
```

Then, run your tests again.

Congratulations! You just wrote your first acceptance tests for an existing service and encapsulated the behavior in a custom step, so it's easy to update your tests should the underlying code change in the future.

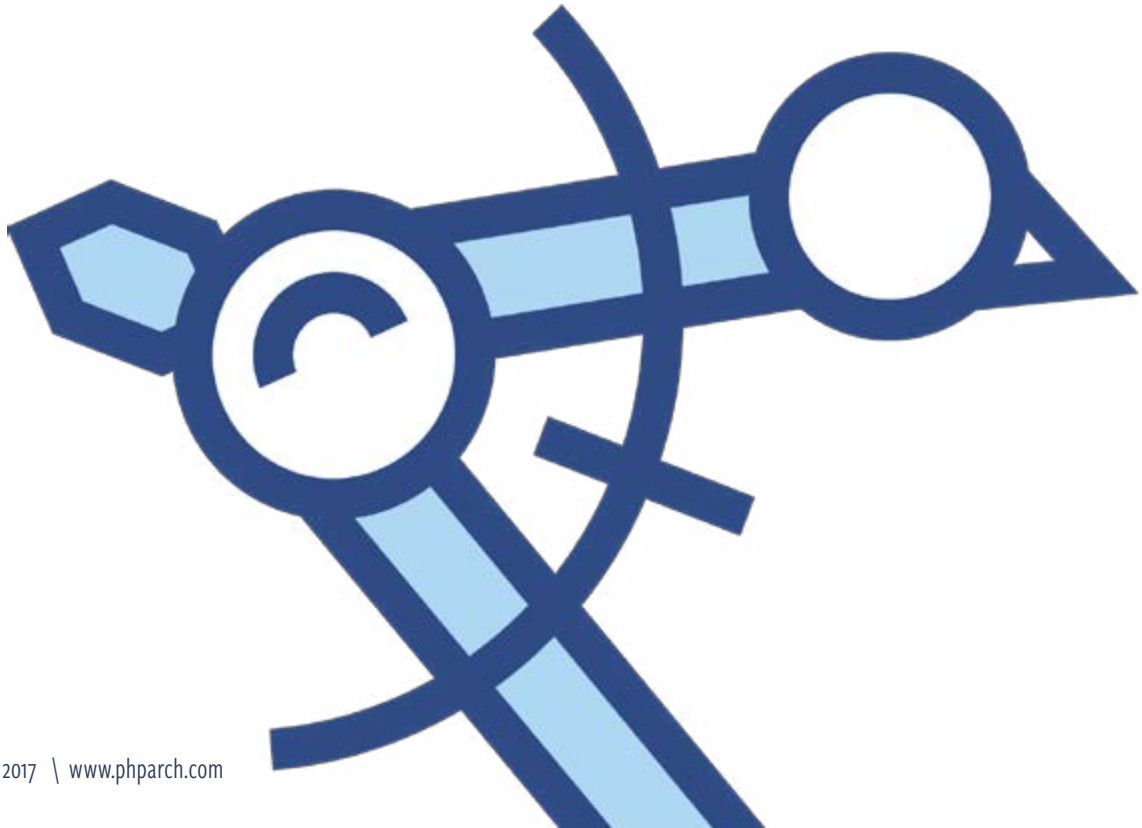
Conclusion

It's not the most glamorous job in the world, but building up a set of characterization tests for your HTTP API can be a remarkably simple job. All it takes is for someone to read the code and work out the various branches through the code so you can ensure that you've captured the entire public interface.

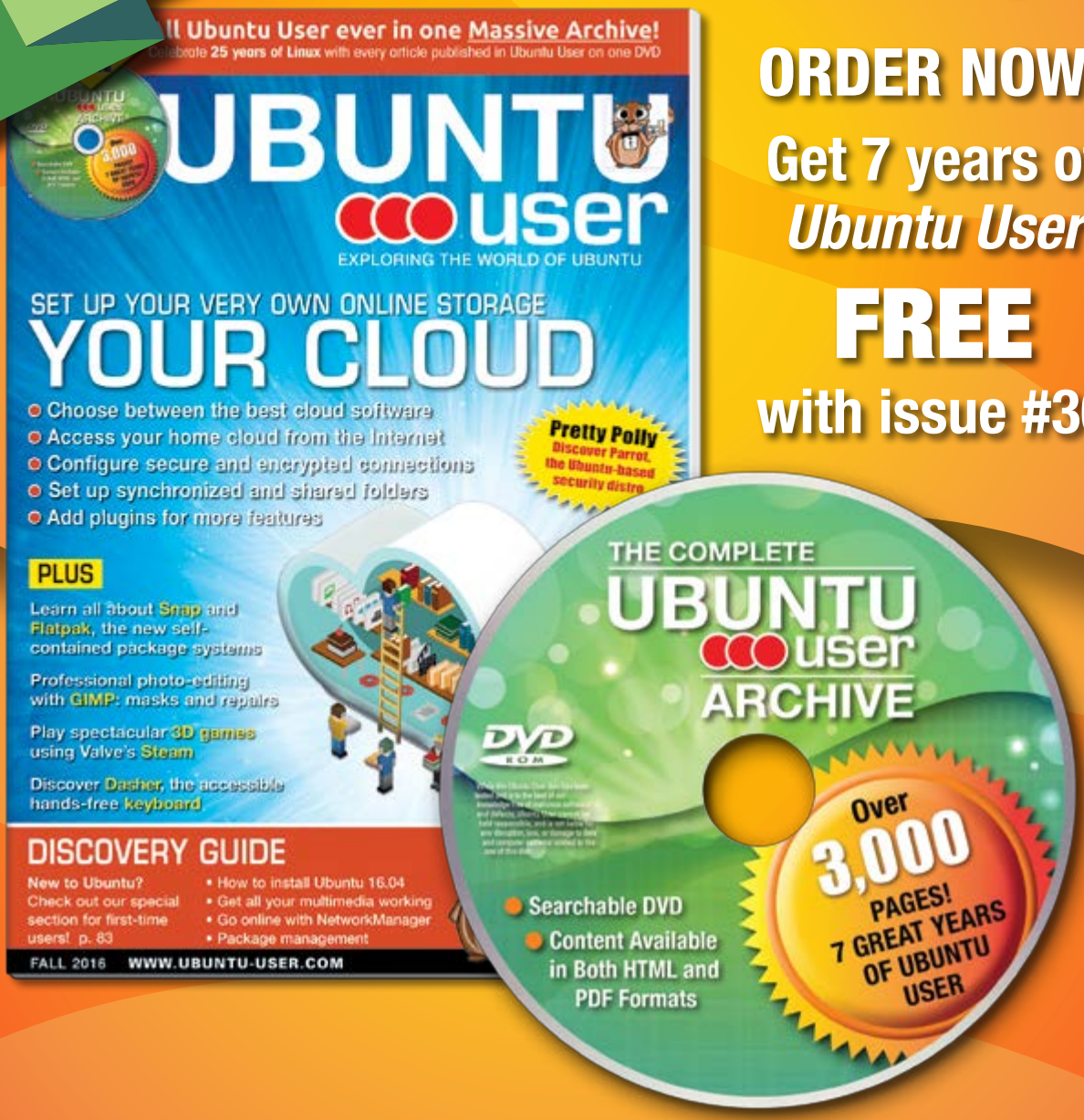
Characterization tests are a great tool which allow a developer to start adding tests to applications that currently have none. By treating the internals of an application as a black box and verifying its external behavior, it frees you up to refactor the internals for any reason you like. This could be to make the application more unit testable, to improve performance, or even to make future development easier. Characterization tests give you confidence that your changes don't have any unintended side effects.

That's all we have time for today, but keep an eye out for a follow-up article in future editions of php[architect]. We'll cover how to write some of the more complex characterization tests, including how to deal with code which makes HTTP calls to external services as part of doing its job.

Michael is a polyglot software engineer, committed to reducing complexity in systems. Working with a variety of languages and tools, he shares his technical expertise to audiences all around the world at user groups and conferences. Day to day, Michael is a fixer. He works on whatever needs an extra pair of hands both at his day job and in open source projects. @mheap



Celebrating 25 Years of Linux!



ORDER NOW!

Get 7 years of *Ubuntu User*

FREE

with issue #30

***Ubuntu User* is the only magazine for the Ubuntu Linux Community!**

BEST VALUE: Become a subscriber and save 35% off the cover price!
The archive DVD will be included with the Fall Issue, so you must act now!

Order Now! Shop.linuxnewmedia.com

Writing Better Code with Four Patterns

Joseph Maxwell

We often have a laundry list of activities we need to accomplish every day. If you think about it, many of these activities seek to solve problems we may face. Have you gone a day (or more) without food? It isn't pleasant to do so. When we feel the problematic pangs of hunger come upon us, we get some food as the solution. This is why the $750 billion restaurant business exists in America.

Because food isn't free, we could face another problem. If we don't have money, we will have a hard(er) time finding food (which is, sadly, true in so many parts of the world). Unless you're independently wealthy, you likely have a job. Our jobs pay the bills: food, utilities, and everything else. Our jobs are the solution to us being able to live the lifestyle we enjoy.

Do you see a trend here? A problem and then a solution. In many cases, we don't see the problem as a problem because we have grown so accustomed to applying the solution. For each of us, this hasn't always been the case. My daughter, who is 13 months old, can't take care of herself: she needs to be fed, needs help getting dressed, etc. These solutions are learned. And so is the content of this article.

A design pattern is a repeatable solution to common problems we face in programming. Almost every industry has something similar: soldiers have tactics and strategies; firefighters have training manuals; pilots have scads of books (and access to flight simulators); builders have blueprints and we—software engineers—have design patterns.

In this article, we will be covering four design patterns: dependency injection, factory method, strategy, and chain of responsibility.

The Benefits of Design Patterns

Perhaps the greatest benefit of design patterns is they present an already understood solution to a problem. As a pattern, they embody the idea for a solution and not a line-by-line cookbook implementation. There is no need to re-invent the wheel. There's no need to think, "How do I instantiate this class inside this other class?" I have trained myself to see this and say, "When my class needs access to another class, I use dependency injection." This means we have a common and more repeatable, and thus, less error-prone, solution.

Design patterns are also recognizable by other developers. They will better understand the code you have written. This makes it easier for you to climb the ranks at your company, as your replacements will catch on to your code more quickly.

Another benefit is that it helps you to write more legible and maintainable code. We have all probably heard of the SOLID principles for programming. Dependency injection helps to keep each class with a single responsibility. It is also an implementation for dependency inversion. The strategy design pattern provides a good solution for the Liskov substitution principle. Let's cover the specifics of several key design patterns that you, as a developer, should be familiar with.

Number One: Dependency Injection

Imagine your roof has a leak. You call ABC Whizbang Roofers who gives you a good price to fix your roof. They show up at the agreed-upon time and ask, "Where's the tools?" Contractors always bring their tools and you begin questioning what type of boss would send their workers out without any tools? You tell them to find some tools, curious if you will see them again.

What does roofing have to do with dependency injection? Dependency injection is giving an object a specification for the dependencies or objects it needs to successfully complete its job. In the previous example, dependency injection would be if the contractor provided his employees with the necessary tools to complete the job. Taking that a step further, dependency injection done right, is the contractor telling his employees to use hammers (interface), but not specifying what brand the hammer needs to be. On his shelf, he may have a Stanley, Estwing or a Stiletto, all of the hammer types. Dependencies may be a database connection, a session wrapper, a class name, or configuration value: anything.

Here is an example of a dependency in the Magento 1 framework: `Mage::registry('current_product');`. In this example, `Mage` is referencing a class that is in the global scope and is a dependency.

This could introduce an entirely new way of thinking—one that was difficult for me to grasp at first. I was used to creating an object or fetching the dependency whenever and wherever. Additionally, every application has to store global state in one way or another, and my objects fetched that state as necessary.

In the old days, we hard-coded class names into our code: `new \Transport()`. We used to have a

global configuration object which stored the values we needed: `Mage::registry('current_product');`. And in the **really** old days, we even called the `$_SESSION` superglobal or a global function like `db()`.

This leads to brittle and hard-to-test code. What if we want to change the `Transport` class to a different one? Find and replace may work, but we might miss something. What if Magento decides `primary_product` is a better label than `current_product`? What if PHP changes the name of the `$_SESSION` super global to another name? In each of these examples, we are also breaking the principle of dependency inversion.

Dependency injection is a great solution to these challenges. At its core, dependency injection follows the principle of dependency inversion. It also gives us better insight regarding how many purposes our classes have, assisting us in the quest to give every class a sole responsibility.

What's Dependency Inversion?

I have found that starting an application and following the dependency injection pattern makes testing easier. For example, to test a class that interacts with a session, I can inject a custom session wrapper—one that does not interact with PHP's session functions.

We're going to look at the more common route: constructor injection. The other way to inject dependencies is setter injection. The problem with setter injection is if you neglect to call a setter, you may have a fatal error.

Listing 1 shown an example of constructor injection:

This is the most basic example of constructor injection. We can instantiate the object by writing this code: `new CustomerManager(new CustomerSession());`. Instead of our customer manager class having to determine what type of session handler to use, we tell it what the appropriate one is. This could be a `BusinessSession` or a `GovernmentSession` (see the strategy design pattern later).

Just by looking at this code, I know what this class needs: a class implementing `CustomerSessionInterface`. Additionally, between the name of the class, `CustomerManager` and its needs, a customer session object, I can see this class would likely return an instance of a customer object, possibly based on the ID found in the customer

LISTING 1

```
01. <?php
02. class CustomerManager
03. {
04.     protected $session;
05.
06.     public function __construct(
07.         CustomerSessionInterface $session
08.     ) {
09.         $this->session = $session;
10.     }
11. }
```

session. This is without adding any docblock comments (which are still a good idea) or using return types in PHP 7.

PHP DI

My favorite tool for dependency injection is PHP DI[1]. It handles constructor injection eloquently and is customizable for those cases that just don't seem to fit any mold.

One thing to note is that most dependency injection utilities use an idea similar to a singleton: it's called a container. In the above example, if we are using PHP DI, I will need to instruct it to resolve that interface to a `CustomerSession` object (see following example). PHP DI instantiates a class, adds it to the container, and will inject that specific object anywhere that we need that type of object. Think of it like caching. On the surface, the object could be viewed as a singleton because it is the same instance used. This can be a surprise if you are not aware of it. Look at the factory method design pattern for more details.

Using PHP DI as our springboard, let's look at how to create a project with PHP DI. First, you will include it with Composer:

```
composer.phar require php-di/php-di
```

Next, in Listing 2, we need to setup the container:

When you call `$container->get()`, PHP DI is autowiring up that class. This involves using reflection[2] to look at the type hinting for the `CustomerManager` class to see it needs a `CustomerSessionInterface`. If we didn't add a definition for `CustomerSessionInterface`, it would try to instantiate a class of that type (which would fail). Since we have added that mapping with `addDefinitions()`, it instantiates our `CustomerSession` class.

The interesting thing is that PHP DI will traverse the entire constructor tree to create the needed objects. So, if `CustomerSession` has dependencies, PHP DI would create those, and if those dependencies have dependencies, it will create or load those too: all the way up. This is what happens the first time. For subsequent instantiations, the objects are loaded from the container.

As a side note, calling a specific object from the container is called the service locator pattern. It is considered an anti-pattern because of the lack of the

LISTING 2

```
01. $builder = new DI\ContainerBuilder();
02. $builder->addDefinitions([
03.     CustomerSessionInterface::class
04.         => DI\object(CustomerSession::class)
05. ]);
06. $container = $builder->build();
07.
08. $customerManager = $container->get(CustomerManager::class);
```

1 *PHP DI:* http://php-di.org

2 *using reflection:* http://stackoverflow.com/q/4262350

dependency inversion principle; your code is still depending on a "God" class. For example:

```
public function __construct() {
   $this->session = Container::get('CustomerSession');
}
```

Instead, we should inject the `CustomerSession` into the constructor using constructor arguments as was demonstrated in the prior code examples.

I have found dependency injection to simplify the code that I write as it helps me to maintain classes which follow the single responsibility principle. Using the PHP DI framework, you have very little to do to implement this design pattern, but understanding how this works is very important!

Number Two: Factory Method

Each of the design patterns we discuss are semi-related or assist each other. This one is no exception.

As you read this magazine, you likely have a mobile phone sitting next to you. Maybe you are sitting on a couch with a gentle light illuminating the pages. Or, maybe you received the digital edition in your inbox and couldn't start the day until you devoured a few pages. Either way, you are interacting with a number of objects you didn't create. You possibly bought your phone online. Maybe you went to the local furniture store to find that perfect couch. The odds are good you didn't build your own computer. Someone built each of those for you **at a factory**.

Think of how much cheaper (likely) smartphones would be if we received a kit of thousands of parts to assemble our own smartphone. The number of people with a smartphone would be a small fraction of what it is today because only those with a lot of time (and motivation) could own one.

On a more serious note, like many other metaphors, we can build factories into our code. As I'm going to show you, they can also provide great benefit.

Consider the previous design pattern: dependency injection. What if our class needs to create a new object (for example a new `Customer`)? The easy thing to write is `new Customer()`. This tightly couples our classes together and is an anti-pattern. If we just inject a `Customer` into the constructor, it would be an instantiated object of the `Customer` type: then we couldn't create multiple `Customers`. Instead, we can delegate that to a class designed to create other classes as seen in Listing 3.

Notice in this factory we are injecting the name of the customer class (we set that up in the dependency injection configuration). We could use this to inject a DSN for PDO or a debug flag. In this case, our factory is acting as a proxy, holding the dependencies until our child needs to be created. Additionally, we can pass unique data to the child through the `create` method's arguments.

This is just one example of a factory. Think how easy, with this example, it would be to substitute the class for something different. Utilizing YAML or XML to provide class names (for extensibility), you could use a factory to instantiate a type of class. Factories can also be used to set up complex objects (such as utilizing a new library with huge constructors).

As we go into the third design pattern, it is important to remember the factory method is for creating objects. After the factory assembled your smartphone, they shouldn't see it again (unless it spontaneously catches on fire). This is the same for software factories: after your object is created, the factory's purpose is complete.

Number Three: Strategy

A year ago, I was tired of borrowing a weed-trimmer from a relative to edge my lawn. The solution was to go to a dealer of such products and see what they had to offer. Their solution for homeowners was a multi-tool. To get a weed-trimmer, you had to buy two pieces: an engine and a weed-trimmer attachment. The benefit of this, they said, was that you could also purchase a sidewalk edger attachment or a pole chainsaw attachment. For all I know, they had a toilet plunger attachment, too.

I also have a few electric drills floating around my house. Each of these has a chuck (similar to interfaces) which

LISTING 3

```
<?php
// Dependency injection configuration (for PHP DI):

return [
   \CustomerFactory::class => DI\object()
      ->constructorParameter('customerClassName',
                             \Customer::class)
];

// Factory
class CustomerFactory
{
   private $customerClassName;

   public function __construct(
       string $customerClassName
   ) {
      if (!class_exists($customerClassName)) {
         throw new \Exception(
            "Customer class has not been set."
         );
      }
      $this->customerClassName = $customerClassName;
   }


   public function create($initialData = [])
      : CustomerInterface {
      return new $this->customerClassName(
          $initialData
      );
   }
}
```

accepts a drill bit, a screwdriver bit, and a plethora of other attachments (concrete classes).

You might see similarities in these two examples: a base unit with a mechanism to connect to an object which implements an interface. It doesn't matter what the attachment does, as long as it implements the interface. That's the strategy design pattern.

I will use the terms "parent" and "child" in the following way: "parent" describes **what** happens and the "child" describes **how** it happens. Take the example of the drill: the "parent" (drill) provides power for the "child" (drill bit) to drill a hole.

The kingpin of this design pattern is with extensive use of interfaces. This may seem like a lot of extra overhead. However, languages other than PHP, such as C++ and Objective-C require you to do this for every class, so it's not necessarily an issue—doing it for a number of files should not be a problem. This is a discipline that took a long time for me to embrace, but I have seen great results in doing so.

When your child classes implement an interface, you can write your parent class to talk to methods that are exposed by that interface. It doesn't matter what the child is or does, as long as it implements the interface.

Here is a simple interface for persisting data:

```
interface ProviderInterface {
   public function update($model);
}
```

Let's write some classes to implement this (see Listing 4):

Please note that these examples are abbreviated for clarity and won't run if you try to execute them. I am primarily trying to convey this way of thinking.

LISTING 4

```
01. /**
02.  * This is a child class:
03.  */
04. class MySqlProvider implements ProviderInterface
05. {
06.    public function update($data)
07.    {
08.       $connection = $this->getConnection();
09.       $connection->exec('UPDATE … SET');
10.    }
11. }
12.
13. /**
14.  * This is a child class:
15.  */
16. class CrmProvider implements ProviderInterface
17. {
18.    public function update($data)
19.     {
20.        $client = new GuzzleHttp\Client();
21.        $client->request('POST', '{url}');
22.     }
23. }
```

PHP 7 is helpful, in Listing 5, as it allows us to type-hint the outputs. Also, notice that we could insert any child `Provider`, and it makes no difference to the functionality of our parent `PersistenceLayer`.

The strategy pattern is useful in any case where you need to swap out algorithms or components. Take an example from ImageMagick (I have no idea what their internal code looks like, but this could certainly be a use case):

```
$imagick->resizeImage($width, $height, $filterType,
                      $blur, $bestFit);
```

`$filterType` is a constant denoting which algorithm to apply when resizing the image. If we were building this in modern PHP, we would probably have each algorithm implement a common interface. When we call the `resizeImage` function, it could take advantage of the strategy design pattern and call the child that is applicable for the filter specified.

Now that you hopefully have an understanding of the strategy pattern, let's take a look at the most complex of the four, one that is similar in concept to the strategy but quite different in implementation.

Number Four: Chain of Responsibility

Chains are very useful. They keep cargo on the back of semi-trucks. They keep your car from sliding off the road in winter. They can even cut down trees.

Typically, when I think of a "chain" in programming, I think of a vertical chain: inheritance. The chain I will be describing is very different—it is horizontal. Think of this chain as a job-runner. Instead of drilling down into data, we are passing it horizontally. Instead of

LISTING 5

```
01. <?php
02. /**
03.  * This is the parent class, providing the host for
04.  * the children:
05.  */
06. class PersistenceLayer
07. {
08.    private $providerFactory;
09.
10.    public function __construct(
11.       ProviderFactory $providerFactory) {
12.       $this->setProviderFactory($providerFactory);
13.    }
14.
15.    public function getProvider(string $endpoint)
16.       : ProviderInterface {
17.       return $this->providerFactory->get($endpoint);
18.    }
19.
20.    public function update(string $endpoint,
21.                           array $data) {
22.       $this->getProvider($endpoint)->update($data);
23.    }
24. }
```

dealing with executing each item in the chain, we call the first item, and that method call returns us the value we want (even though there may be multiple chain links that were called). Another way to think of it is similar to the Unix pipe operator: the output from the command on the left of the operator is the input for the command to the right.

Unfortunately, PHP doesn't have an exact replica for the pipe operator. I use a "transport" object to carry data from one command to the next. This is ideal because you can only return one value or object and passing variables by reference is prone to issues. This transport object can be sent to each link of the chain and have values set or unset on it, to be passed onto the next link.

The simplest way to go through a list of commands is to instantiate each link's object and use a `foreach` loop to call each item. Depending on how complex your system is, this may be fine. However, that can become unwieldy as your application gets bigger.

The Setup:

The following implementation of the chain of responsibility is very clean and easy to extend. It also takes some explaining.

The idea is that the concrete part of each link only knows about what action **it** takes. There is an abstract class each link extends, handling the chaining details. The unique thing about this way of chaining is that it is completely horizontal. You append link to link, and the abstract class handles the chaining. In Listing 6, let's take a look at how this works:

Let's look at each method:

`__construct(ChainLinkInterface $nextLink);` dependency injection! This stores the next link in the chain as a class property. `getDataFor($key);` is the function that does the work. Notice it is `protected`. We only want to have the abstract class call this function on the concrete implementation. It is hidden because we only want our concrete class to implement it, as this function likely depends on additional information to successfully complete its job. `get(\Transport $transport)`: this function gets the data from `getDataFor($key)`. If there was a value returned, the chain is complete (depending on your implementation). Otherwise, it passes it onto the next item. At the end, this function will return the `Transport` object.

You can have the chain of responsibility stop at the first winner or have it process each link no matter what. An example of it stopping on the first successful result would be looking up a key in a list of databases: the first one that finds the value returns it, and the chain stops executing. An example of the second could be running consecutive filters on an image.

Concrete Links:

Listing 7 shows an actual implementaion of the `ChainLinkInterface`.

LISTING 6

```
<?php
interface ChainLinkInterface
{
    protected function getDataFor($key) : void;
    public function append(ChainLinkInterface $nextLink);
    public function get(\Transport $transport) : \Transport;
    public function process(\Transport $transport) : bool;
}

abstract class ChainLink
{
    protected $nextLink;
    protected $result;
    protected $data;

    public function __construct(
        ChainLinkInterface $nextLink
    ) {
        $this->nextLink = $nextLink;
    }

    public function get(\Transport $transport): \Transport {
        $value = $this->getDataFor($transport->getKey());
        if (!empty($value)) {
            $this->result->setResult($value);
        }

        if (empty($value) && $this->nextLink) {
            $this->nextLink->get($transport->getKey());
        }

        return $this->result->getResult();
    }
}
```

LISTING 7

```
<?php
class FirstAction extends ChainLink
                  implements ChainLinkInterface
{
    protected function getDataFor($key) {
        return $this->doSomeActionOnOurKeyVariable($key);
    }
}

class SecondAction extends ChainLink
                   implements ChainLinkInterface
{
    protected function getDataFor($key) {
        return $this->runAComplexImageProcessingAlgo($key);
    }
}
```

Putting It All Together:

Whew, there was some code! As you look at this and study it, you will see it isn't complex after all. The `ChainLink` class is only a few lines, and every concrete chain link extends it. So, how does it all go together?

```
$masterAction = new FirstAction(
   new SecondAction()
);
$masterAction->get(new \Transport());
```

`get()` will recursively call the object that `$nextLink` points to, which will call the object that `nextLink` points to, etc. This will happen until there is no `$nextLink` variable set or you choose to halt the execution.

It is a little difficult to wrap one's mind around using an abstract class which interacts with itself. Think of it as interacting with different and unique concrete implementations of itself. The abstract class is just an auxiliary to each concrete class you write.

The possibilities are infinite. You can load actions from YAML and run them consecutively. I used this to create a bridge with a popular CRM's API: multiple actions had to be taken to get a lead (customer) pushed up into their system. The number of actions determined what type of lead was being pushed up. It needed flexibility, so the idea behind the Chain of Responsibility design pattern couldn't have been more perfect. You could use it to apply filters to an image or find a value from a chain of cache stores. There are many uses for a great pattern such as this.

In Conclusion

As I wrap up our dive into these four design patterns, I want to reiterate how each of these interacts together but also has its own, unique purpose.

Dependency injection is **the practice of providing each class the values it needs to do its job**. The class should not be reaching to global objects. Using a constructor injection provides good visibility into what the classes' dependencies are.

The factory method is **used to instantiate objects**. I have found it helpful to pair it with dependency injection. In that case, the factory acts as a proxy. We can inject dependencies into the factory, the ones that will be needed by our instantiated object. With a `create` method, or something similar, we can pass additional and custom arguments into this instantiate object, providing two paths to provide data to the new object.

The strategy pattern **represents a plug-and-play for build actions**. A parent doesn't know who the child is as long as the child implements a predetermined interface.

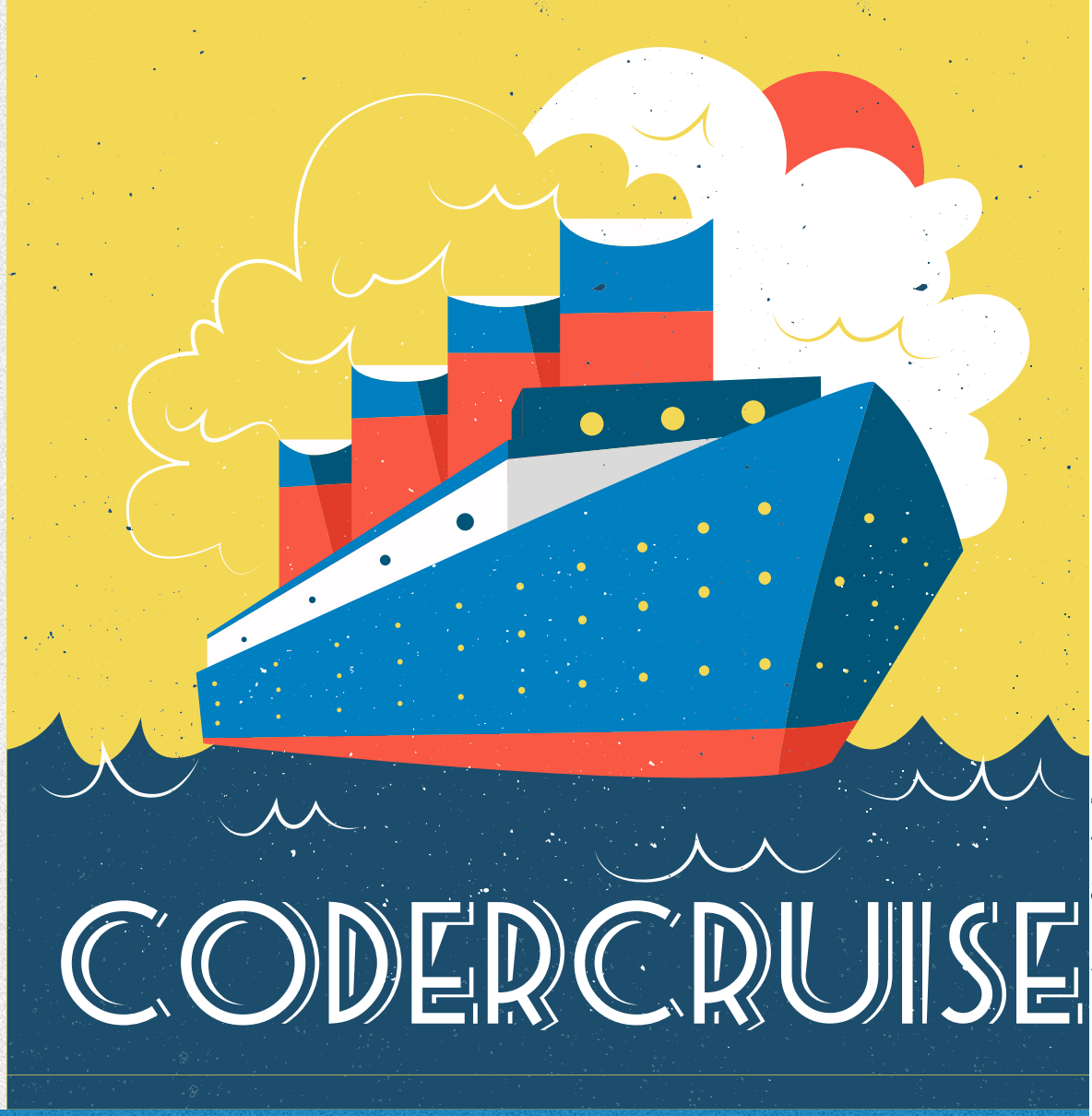
Finally, the chain of responsibility is helpful for chaining multiple actions together. After one completes, the next link takes over. Each should extend a common abstract class, which deals with the chaining functionality. The concrete class then deals only with the action to be taken.

Just like we are blessed to have food that solves our problem of hunger, we have the factory design pattern to solve the problem of instantiating complex objects. Earning money is nice, as that allows me to feed my family. In that way, these patterns can be applied just about anywhere, leading to cleaner and more maintainable code.

I challenge you to become a constant student of the school of design patterns. It's a learning center that continually makes you a better programmer.

Joseph Maxwell is the president of his Kansas-based company, SwiftOtter Studios. A Zend Certified Engineer and a multiple Magento certification hold, he's been programming for almost 15 years. Joseph is passionate about helping others write and understand code, and he's active in his local PHP user group, including speaking at meetings. @swift_otter

Sponsors



7 days at sea, 3 days of conference

Call for Speakers
Open until Jan 6th

Leaving from New Orleans
and visiting Montego Bay,
Grand Cayman, and Cozumel

July 16-23, 2017 — Tickets $250

www.codercruise.com

Presented by One for All Events

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

James Titcumb

Reflection is commonplace in many modern programming languages and allows analysis of code structure and more. What if we could do more? Better Reflection is a library that goes further than the simple reflection we're used to in PHP. I'll explore how the library works, some of the difficulties we faced, and how you can leverage this new power today.

Reflection In PHP

Reflection. Many experienced PHP developers have used it before. Most likely, you've used it in unit tests to access a private or protected property, or to call a private or protected method. Some projects, like Symfony and Doctrine, use reflection to parse annotations in comment blocks. The code in Listing 1 might look pretty familiar to many PHP developers:

LISTING 1

```
01. public function testSomething()
02. {
03.     $myObj = new Thing();
04.
05.     $propReflection = new \ReflectionProperty($myObj,
'foo');
06.     $propReflection->setAccessible(true);
07.     $propReflection->setValue($myObj, 'whatever');
08.
09.     // ... rest of test ...
10. }
```

But what is reflection? It's not a new concept by any measure. The concept of reflection is introspection about code being executed. It's a meta-concept, allowing you to examine code structure of classes, properties, functions, methods and so on. It allows you to examine values; even those which are private or protected properties, as shown above. Reflection, in general, also grants the ability to modify the structure or behavior of code at runtime in a limited way. It's a concept which exists in multiple programming languages, such as Go, Java, Python and so on; PHP exposes this functionality in the `reflection` extension[1].

The `reflection` extension is built-in to PHP, can't be disabled, and is around 7,000 lines of C code. The reflection extension works very well because it has direct access to the real values that make up the variables we see in what is known as userland PHP—the PHP world we're used to working in. Looking at one of the more simple methods in the reflection extension (Listing 2), the implementation of `ReflectionClass#hasMethod` reveals there's not much going on here.

LISTING 2

```
01. ZEND_METHOD(reflection_class, hasMethod)
02. {
03.     reflection_object *intern;
04.     zend_class_entry *ce;
05.     char *name, *lc_name;
06.     size_t name_len;
07.
08.     METHOD_NOTSTATIC(reflection_class_ptr);
09.     if (zend_parse_parameters(ZEND_NUM_ARGS(), "s", &name,
10.         &name_len) == FAILURE) {
11.         return;
12.     }
13.
14.     GET_REFLECTION_OBJECT_PTR(ce);
15.     lc_name = zend_str_tolower_dup(name, name_len);
16.     if ((ce == zend_ce_closure &&
17.         (name_len == sizeof(ZEND_INVOKE_FUNC_NAME)-1)
18.         && memcmp(lc_name, ZEND_INVOKE_FUNC_NAME,
19.                   sizeof(ZEND_INVOKE_FUNC_NAME)-1) == 0)
20.         || zend_hash_str_exists(&ce->function_table,
21.                                 lc_name, name_len)) {
22.         efree(lc_name);
23.         RETURN_TRUE;
24.     } else {
25.         efree(lc_name);
26.         RETURN_FALSE;
27.     }
28. }
```

If we skip over things like parsing parameters and a few necessary macros to perform some various housekeeping tasks we see that the primary operation of this method boils down to the call to `zend_hash_str_exists`. This function checks to see if the name provided as the parameter to `hasMethod` exists in the classes function table. To understand

1 `reflection` *extension:* https://php.net/book.reflection

why this works and is so straightforward, let's quickly look at how a class instance is represented by the Zend Engine in C:

`zend_object` - this is essentially the instance of the class and has other properties not listed here, such as handlers and garbage collection.

- `zend_class_entry* ce` - the `ce` or class entry references a `zend_class_entry` which is the blueprint of the class itself; i.e. information about the structure:
  - `HashTable function_table`
  - `HashTable properties_info`
  - `HashTable constants_table`
  - `zval* static_members_table`
  - `zend_class_entry** interfaces`
  - `zend_class_entry** traits`
  - etc.

The important part we care about here is that the `function_table`, `properties_info`, and `constants_table` are all implemented as `HashTables`. We can think of these, essentially, as arrays. For example, it's very easy to look up whether a function exists.

The point is we can directly access what is going on under the hood, and expose it in userland via the reflection API, meaning we can start messing around with things that shouldn't be messed around with.

It's Like Reflection, But Better

Having the inquisitive nature to find out how things work, I thought why stop there? I'm going to introduce to you a library called Better Reflection[2], which is a reflection library written in pure PHP, which does not require any extensions, and does not use the internal Reflection API directly. One of the original goals for the Better Reflection library was to create a compatible layer offering a drop-in replacement for the internal reflection API, meaning it can work with the existing functionality that uses it, but can also provide additional features. There are a few caveats, though, so we describe the API as "mostly compatible" with the PHP core reflection API.

The first question we need to answer is, "*why*?" The goal of Better Reflection, as the name suggests, is to be better; specifically, to provide more features and awesome superpowers which allow you to do things with code that were not possible before. A key component of Better Reflection is that you can reflect on classes that have not actually been loaded yet. As explained earlier, the PHP internal `reflection` extension works by already having access to the loaded class blueprint. By creating reflections from classes that don't exist yet, we can start to do some pretty interesting things, such as modifying the class itself, or its methods, prior to loading. This also applies to third party codebases, such as downloaded PHP packages we don't trust. This technique is called monkey patching, and we'll look at how it works in a bit. Before we look into that, we're going to explore exactly how Better Reflection works.

Creating a reflection in Better Reflection is, at least behind the scenes, a four step process. It's still pretty straightforward, however, and where you might currently do something like this with PHP's internal reflection:

```
use ReflectionClass;

$reflection = new ReflectionClass(
  \My\ExampleClass::class
);
```

In Better Reflection, it's almost as simple for most use cases:

```
use BetterReflection\Reflection\ReflectionClass;

$reflection = ReflectionClass::createFromName(
  \My\ExampleClass::class
);
```

This static constructor, createFromName looks something like this under the hood:

```
public static function createFromName($className) {
   return ClassReflector::buildDefaultReflector()
          ->reflect($className);
}
```

Underneath, it uses this layer called a `Reflector`, and because we're trying to reflect a class, it uses the `ClassReflector`; yes there is a `FunctionReflector` too for reflecting on functions (not class methods) directly. We call the `buildDefaultReflector` static constructor which creates a `ClassReflector` with some source code locators which behave in a transparent way, and calls `reflect` on that, passing the class name. The `buildDefaultReflector` call looks like this:

```
public static function buildDefaultReflector() {
   return new self(new AggregateSourceLocator([
      new PhpInternalSourceLocator(),
      new EvaledCodeSourceLocator(),
      new AutoloadSourceLocator(),
   ]));
}
```

These "source locators" are different ways of finding and preparing source code to be analyzed by Better Reflection. There are several Source Locators provided out the box, and they are (at the time of writing):

- `AggregateSourceLocator`
- `AutoloadSourceLocator`
- `ClosureSourceLocator`
- `ComposerSourceLocator`
- `DirectoriesSourceLocator`
- `EvaledCodeSourceLocator`
- `FileIteratorSourceLocator`
- `PhpInternalSourceLocator`
- `SingleFileSourceLocator`
- `StringSourceLocator`

The `AggregateSourceLocator` is simply a source

2 *Better Reflection:* https://github.com/Roave/BetterReflection

locator which accepts an array of other source locators. It allows chaining and multiple methods of locating source code, such as is used in the default reflector above: it uses an `AggregateSourceLocator` consisting of, in order, `PhpInternalSourceLocator`, `EvaledCodeSourceLocator`, and `AutoloadSourceLocator`. In Listing 3, let's look at how we can use a single source locator by itself.

In this example, we are using a `StringSourceLocator` directly. Because we would like to use a specific source locator, we can't use the convenient static constructor above. Instead, we need to use the `ClassReflector` directly, passing in the source locator as the constructor parameter, with the string of code itself as the constructor parameter to `StringSourceLocator`.

The default source locator for `ClassReflector` should cover 99% of use cases. First, the `PhpInternalSourceLocator` uses a set of built-in stubs which define internal classes. Next, `EvaledCodeSourceLocator` uses the `Zend\Code` code generation library to generate source code executed by the `eval()` function in PHP. The third and final default locator, `AutoloadSourceLocator` is where some real magic happens. In order to figure out how to load classes, most autoloaders (including Composer's autoloader) map a namespace to a PHP file on the filesystem, having a 1:1 ratio of class to file. If we make the assumption that most autoloaders work this way, we can do something interesting to determine where we expect the file containing a class will be located—even if it hasn't yet been loaded.

The technique boils down to these lines of code:

```
self::$autoloadLocatedFile = null;
$previousErrorHandler = set_error_handler(function () {});
stream_wrapper_unregister('file');
stream_wrapper_register('file', self::class);
class_exists($className);
stream_wrapper_restore('file');
set_error_handler($previousErrorHandler);
return self::$autoloadLocatedFile;
```

First, we essentially remove the error handler (set it to an empty function) to silence an inevitable, unavoidable error we're about to trigger. We then unregister the existing stream wrapper for handling files—that is how PHP opens files, and register our own (`self::class`). The `AutoloadSourceLocator` implements a couple of methods (`stream_open` and `url_stat`) used to hijack file system calls:

```
public function stream_open($path, $mode,
                            $options, &$opened_path) {
    self::$autoloadLocatedFile = $path;
    return false;
}
```

When the call to `class_exists` happens, our own `stream_open` function is called, and we store the name of the file in the attempt to load it. We then return `false`, which is what causes the inevitable error mentioned above. Finally, the stream wrapper and error handlers are restored to their former glory, the balance is restored, and we can now return the name of the file the autoloader tried to open.

LISTING 3

```
01. use BetterReflection\Reflector\ClassReflector;
02. use BetterReflection\SourceLocator\Type\StringSourceLocator;
03.
04. $source = <<<EOF
05. <?php
06. class MyClassInString {}
07. EOF;
08.
09. $reflector = new ClassReflector(
10.     new StringSourceLocator($source)
11. );
12.
13. $classInfo = $reflector->reflect(MyClassInString::class);
```

Of course, the assumption we make is that your chosen autoloader will definitely load a file. If it's doing something weird and not loading a file, then this isn't going to work, and you're going to have to use one of the other source locators to find your class. Finding out what the other `SourceLocator` classes do can be left as an exercise for you to find out, but generally, they're quite self-explanatory, and most have at least basic usage documentation.

Climbing the Abstract Syntax Tree

Okay—so we have some source code located—how can we turn that into a reflection? The next step in the process is parsing this source code into something called an abstract syntax tree, or AST for short. This part of the process is pretty much entirely taken care of by a library called PHP Parser[3], written by internals developer and CS/physics student Nikita Popov. The purpose of this library is to take PHP source code as input, lex it, parse it, and return an AST representation of it:

```
use PhpParser\ParserFactory;
$parser = (new ParserFactory)->create(
    ParserFactory::PREFER_PHP7
);

print_r($parser->parse('<?php echo "Hello " . "world";'));
```

If we looked at the output of this script, it would be very verbose. To simplify the visualization, we would see this data structure represented:

- Echo
  - Concat
    - Left
      - String, value "Hello"
    - Right
      - String, value "world"

Each of these items is called an AST node and is a data structure representing a piece of code. Unlike tokenization (a method that could also be used), we don't have to deal with

3 *PHP Parser:* https://github.com/nikic/PHP-Parser

the actual syntax of the file, such as what the concatenation operator looks like, or whether or not there is a semi-colon at the end of the line. Instead, we get a pure representation of the execution steps of this code, such that we could navigate this graph of AST nodes, and execute them.

Let's look at a simple class structure to see how useful this information can be, and how we can use this to create the reflections themselves:

```
class Foo {
    private $bar;

    public function thing() {
    }
}
```

The AST of this class, if parsed and simplified a lot, might look something like this:

- Class, name "Foo"
  - Statements
    - Property, name "bar"
      - Flags: private
    - Method, name "thing"
      - Flags: public
      - Parameters […]
      - Statements […]

The reflections in Better Reflection are given the root AST node for the symbol they represent. So, a `ReflectionClass` instance in Better Reflection would eventually be instantiated using the `createFromNode` static constructor, into which the above AST structure would be passed. Subsequently, calling `ReflectionClass#getMethod('thing')` would instantiate a `ReflectionMethod` instance using its own `createFromNode` method, and the AST passed to this would be the `Method` node:

- Method, name "thing"
  - Flags: public
  - Parameters […]
  - Statements […]

Monkeying Around

At this point, we can start to see where monkey patching, which I mentioned we'd look at, comes into play. The `Method` node here has an array of statements also represented by AST nodes, and because the AST nodes are nothing more than a simple data structure, these nodes can be modified, added to, removed or even completely replaced. Although this is possible using the PHP Parser library by itself, Better Reflection provides nice helper methods and a nice, simple to use API to do this. Let's say you have a class like so:

```
class MyClass {
   public function foo() {
      return 'foo';
   }
}
```

Given you have a reflection representing this piece of code, you can use a helper method such as `setBodyFromClosure` to completely change the behavior of the foo method.

```
$methodInfo = $classInfo->getMethod('foo');

$methodInfo->setBodyFromClosure(function () {
   return 'bar';
});
```

You would then use Better Reflection's provided `ClassLoader` which allows caching of modified reflections by registering the reflection with the loader as shown in Listing 4.

Finally, we can load the class:

```
$c = new MyClass();
// should echo 'bar' instead of 'foo'
echo $c->foo() . PHP_EOL;
```

When executing the `foo` method now, we should see the result `bar` printed instead of the original foo the class defined. This part of the process is still a little work in progress, so the API may change.

There are some interesting applications for monkey patching that come to mind; for example, removing `final` modifiers on classes allowing the class to be extended for mocking, or perhaps to automatically generate proxy classes. It is also possible to patch third party library code to modify functionality. Although, in practice, my recommendation would be to discuss and propose an upstream patch to the package maintainers. Despite this functionality being available, you should be mindful about how it is used; it can quickly become difficult to maintain or even create insecure code when patching lots of code in this way. It might be wiser to think about other ways to solve the problem first!

LISTING 4

```
01. use BetterReflection\Util\Autoload\ClassLoader;
02. use BetterReflection\Util\Autoload\ClassLoaderMethod\EvalLoader;
03. use BetterReflection\Util\Autoload\ClassPrinter\PhpParserPrinter;
04.
05. // Instantiating this auto-registers it as an autoloader,
06. // so this should be done AFTER all other autoloaders
07. $loader = new ClassLoader(new FileCacheLoader(
08.     __DIR__ . '/../path/to/cache/directory',
09.     new PhpParserPrinter()
10. ));
11.
12. // Must register the class, otherwise the ClassLoader will
13. // ignore
14. $loader->addClass($classInfo);
```

Building the Library

Creating this library wasn't a particularly smooth process in retrospect. We came across many head-scratchers during the process which challenged us. The first issue was actually building the library itself, and making it work the same way. It turns out, the `ReflectionClass` object alone has 49 public methods, and even at this stage, not all of these have been implemented. Not only is aiming for 100% compatibility like-for-like a pretty lofty goal, but the API itself can be a little quirky and difficult to understand. The PHP manual, as usual, goes a long way to helping, but occasionally I found some that were very unclear. For one of them, I even contributed to the PHP documentation project to improve the description—specifically for `ReflectionProperty#isDefault`, which turns out to be a Boolean flag on whether a property was declared at compile time (i.e. hard-coded into the source) or a dynamic declaration at runtime.

Type determination can be a pretty complex topic, especially since the introduction of namespaces in PHP 5.3. Given the piece of code in Listing 5, could we easily figure out what the fully qualified class name of the `InvalidArgumentException` was?

The answer is, of course, no. Unless we look up at the `use` and `namespace` statements in the file (usually at the top, but not always!), there's no way we can tell. We need to fill those statements out, to provide the context for the code (see Listing 6).

Now we can see the `InvalidArgumentException` being thrown is actually a `Some\Package\InvalidArgumentException`. Thankfully, a library already exists to make this process much easier. In Better Reflection, we use the `phpDocumentor/TypeResolver` library[4] from the fantastic Mike van Riel, which made it very simple to figure out fully qualified class names and types given context available. We wrote a few wrappers internally called `TypeFinders`:

- `FindPropertyType`: given a `ReflectionProperty` instance, find the type(s) of that property using the `@var` DocBlock
- `FindReturnType`: given a `ReflectionFunction` or `ReflectionMethod` instance, find the return type(s) using the `@return` DocBlock
- `FindParameterType`: given a `ReflectionFunction` or `ReflectionMethod` and the `Param` AST node, find the type(s) of that parameter using the `@param` DocBlock
- `FindTypeFromAst`: given a string or AST node, and a `LocatedSource` object (which provides the above mentioned context), figure out the type.

The last finder, `FindTypeFromAst`, is probably the most useful and flexible in general use cases, as it can be used outside the context of reflection, but it's worth bearing in mind it doesn't do much more than create a consistent interface for use within the Better Reflection library.

Creating reflections from closures was another interesting one, and again we found another library that does this (also using Popov's PHP Parser in turn) called `SuperClosure`[5] from Jeremy Lindblom. The only downside to this is including a whole library (whose intent is distinctly different), just to achieve one small piece of functionality. The fact SuperClosure uses PHP Parser under the hood means we have the potential to factor out the additional dependency and bring the functionality in-house, so to speak.

We also uncovered an interesting issue in PHPUnit which resulted in a blocker for us. The issue comes from the way the `PhptTestCase` works in PHPUnit when running `.phpt` tests. The problem is the `PhptTestCase` splits the `.phpt` test in memory, and pipes the PHP source (the `--FILE--` part of the `.phpt` test) directly into PHP. Examining the CLI SAPI for PHP, we see:

```
/* We could handle PHP_MODE_PROCESS_STDIN in a different  */
/* manner here but this would make things only more       */
/* complicated. And it is consitent with the way -R works */
/* where the stdin file handle is also accessible.        */
file_handle.filename = "-";
file_handle.handle.fp = stdin;
```

The filename of the script is hard coded to hyphen because there is no way to determine what the filename is (the script came from `STDIN`). When PHP's original test suite runs, it splits out the `--FILE--` portion of the test and writes a temporary `.php` file to disk, and then runs that, so the issue is unique to PHPUnit. Unfortunately, to change this functionality in PHPUnit would mean almost a complete re-write

LISTING 5

```
01. <?php
02.
03. namespace ??????????;
04.
05. use ????????????????????????????????????;
06.
07. class Foo {
08.     public function something() {
09.         throw new InvalidArgumentException('Oh noes!');
10.     }
11. }
```

LISTING 6

```
01. <?php
02. namespace My\Package;
03.
04. use Some\Package\InvalidArgumentException;
05.
06. class Foo {
07.     public function something() {
08.         throw new InvalidArgumentException('Oh noes!');
09.     }
10. }
```

4 `phpDocumentor/TypeResolver` *library:* https://github.com/phpDocumentor/TypeResolver

5 `SuperClosure`: https://github.com/jeremeamia/super_closure

of the `PhptTestCase` class which could be time-consuming and challenging, while ensuring we don't introduce backwards compatibility breaks. For an interesting discussion on this, and to see the ultimate closure of the issue in the PHPUnit bug bankruptcy, head to PHPUnit Issue 1783[6].

Finally, we've recognized the library is much slower than PHP internal reflection. The reason for this is the use of Nikita Popov's PHP Parser library, which is doing the job of the PHP lexer and parser, but in PHP itself, isn't particularly performant. However, after discussion, we concluded it's really not a priority issue. The reason for this is all the sensible use cases for Better Reflection warrant ahead of time usage, rather than runtime reflection. For example, monkey patching should be done ahead of time, and the monkey patched code should be cached on disk and loaded just like regular PHP code. As we'll see in some of the other use cases for Better Reflection, tools can be written for static analysis, which is not, and should not, necessarily be done at runtime. Therefore, the performance issue is a non-issue because Better Reflection shouldn't even be used when a user loads a web page, for example.

Some Potential Use Cases

What can Better Reflection be used for, exactly? Lots of things! Your imagination is the limit. Static analysis comes to mind due to the way Better Reflection can analyze types in both PHP 7+ code and older PHP 5 code, thanks to the compatible nature of Popov's PHP Parser library. Better Reflection could be used in a similar way to HHVM's "Hackificator" tool—to suggest and even add type declarations to PHP 5 code to convert it to PHP 7 code.

Another use case could be to write a tool which does API analysis and alerting when a public API has changed; for example, by changing types or number of parameters and so on. Using Better Reflection, you can easily fetch a list of every class in a directory and iterate the reflections producing a specification for the public API. If you can do this against two different versions of a package, and note the differences between the two public APIs, you can easily detect basic backwards compatibility breaks.

What can Better Reflection be used for, exactly? Lots of things! Your imagination is the limit. Static analysis comes to mind due to the way Better Reflection can analyze types in both PHP 7+ code and older PHP 5 code

You could also use Better Reflection to analyze `use` statements to determine explicit dependencies used in a project and perform a check against `composer.json` if the dependencies are listed correctly. Or perhaps as a security analysis tool which checks for unescaped output, by following the path a variable takes through the application; if the variable doesn't pass through an elected escape method, then emit a warning.

Another use case could be to be used in libraries like Doctrine. Currently, an entity in a project using Doctrine ORM cannot be declared as `final`, on account of the lazy-loading proxies Doctrine uses (which extend the entity). This could be overcome by using the monkey patching functionality of Better Reflection. When the proxy needs to be made, remove the final declaration from the class before loading it, allowing extension of the entity, or even embedding the additional functionality directly into the entity itself (thus avoiding the need to extend the entity at all).

As you can see now, the Better Reflection library aims to provide the basic needs for reflection but allows a huge amount of flexibility and control over your code, which isn't currently possible with PHP's internal reflection. There's plenty of scope for new features too, and as it starts being rolled out more and being used in various ways, we can see lots of potential for this library. Better Reflection is an open source project, which means of course feedback, issues, and pull requests are very welcome!

James is the founder of the UK based PHP Hampshire user group and the PHP South Coast Conference. He's also a Zend Certified Engineer and consultant at Roave. During his downtime, he continues to run the PHP Hampshire user group and the conference, and keeps up with active contributions to various open source projects.
@asgrim

6 *PHPUnit Issue 1783:* http://phpa.me/phpunit-issue-1783

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

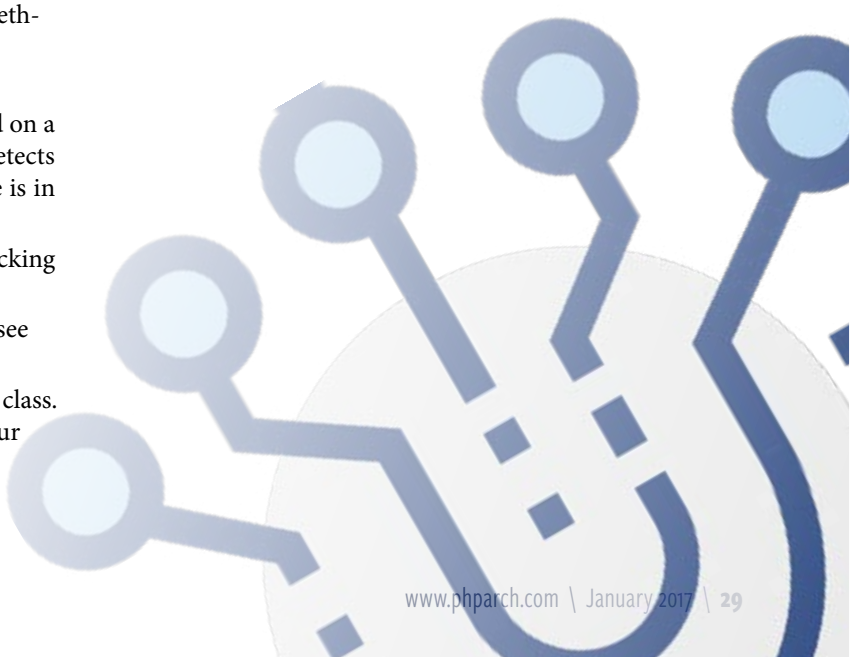
```
01. protected function verbose($message, $lines = 1) {
02.    if ($this->caller) {
03.       $this->caller->verbose($message, $lines);
04.    }
05. }
06.
07. public function doSomething($memberId = 0) {
08.    $memberId = (int)$memberId;
09.    if ($memberId <= 0) {
10.       $this->verbose('Invalid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

Monkey Patching Your Way to Testing Nirvana

Matthew Setter

Ever found PHP's built-in functions lacking? Ever concluded they didn't quite do what you wanted them to do? Ever wanted them to operate just a little differently?

Ever wondered what your application might do if you could change what these functions returned? Then welcome to Monkey Patching. In this month's edition of Education Station, we're going to see how to apply monkey patching in your testing efforts.

What Is Monkey Patching?

If you've not heard of Monkey Patching[1] before, don't be surprised. It's not something all that common to PHP. But, it is quite common to other dynamic languages, especially Ruby, Python, and Emacs Lisp. The following quote from culttt.com[2] sums it up most succinctly:

> *[Monkey Patching is] the ability to re-open any class [or function] and change its methods.*

Said another way, existing classes and functions can be replaced or modified, allowing them to operate differently than normal.

Why Should I Use It?

Monkey patching is often used in a combination of the following four ways:

1. To replace methods attributes and functions at runtime.
2. To modify or extend behavior without maintaining a private copy of the source code.
3. To apply a patch at runtime to objects in memory, instead of the source code on disk.
4. To distribute security or behavioral fixes that live alongside the original source code.

Sounds quite powerful, and it is. To help put it in context, let's consider a few scenarios in which using monkey patching could come in handy.

Imagine we're back in 1999; the year 2000 is right around the corner, and you've noticed a gaping hole in PHP's `time()` method. However, you don't have the technical expertise to submit a patch in C to PHP internals; it looks like no one else is going to either.

Here's another one, graciously borrowed from James Titcumb, *Better Reflection—The Lowdown*[3]: What if a patch you need won't get accepted by an upstream project yet, for some reason, you can't or don't want to maintain a fork?

In both of these scenarios, monkey patching would come in very handy, and allow you to do what you need to do. I'd argue it would be a good time to make use of the power it offers.

Why Should I Not Use It?

Like anything which is quite powerful, there's also a downside. Others, far more experienced than I am, can articulate the reasons more eloquently; such as expert developers like Avdi Grimm, who had this to say about them in *Monkeypatching is Destroying Ruby*[4]:

> *[Monkey] patches interact in unpredictable, combinatoric ways. And by their nature, bugs caused by monkey patches are more difficult to track down than those introduced by more traditional classes and methods.*

So while monkey patching can be incredibly powerful, it appears to be a tool best used with measured restraint and discipline, in the *right* circumstances.

Monkey Patching Can Rock Your Testing Noodle

Now this sounds like an awfully bad way to start off a column about using them. Granted. But, having given both sides of the coin, I want to share and explore with you a way which is quite valid—in testing.

As one of my core pursuits with software development is software craftsmanship, I care deeply about the quality of the code I create. As a result, I've invested the hours learning about a range of testing libraries and frameworks, including PHPUnit, Mockery, Codeception, PHPSpec, Behat, and so on.

1 *Monkey Patching:* https://en.wikipedia.org/wiki/Monkey_patch

2 *quote from culttt.com:* http://phpa.me/monkey-patching-ruby

3 *Better Reflection—The Lowdown:* http://www.jamestitcumb.com/posts/better-reflection-the-lowdown

4 *Monkeypatching is Destroying Ruby:* http://phpa.me/patching-destroy-ruby

www.phparch.com \ January 2017 \ 31

- Through these libraries, I've learned to create tests with mocks, spies, and fakes to help me ensure the code which I'm creating does what I expect it should. After I felt comfortable with my level of expertise there, I moved on to learning about mutation testing, which I wrote about back in the March 2016 issue[5].

If you missed that month's column, mutation testing creates pseudorandom variations in your code and checks if your tests complain. If not, then they're perhaps not as reliable as you might think.

As a result of learning all these libraries, I feel a lot more rounded as a developer; I feel a lot more comfortable with the code I ship. While I love these libraries, there are times that I have felt the need to go further.

After all, if I can mock the third-party libraries which I use, why not mock the granddaddy of them all—PHP? Why should that be any different? Arguably, it's not. It should be fair to treat PHP as just another third-party library which my application makes use of.

Now, I can't say with complete confidence, but I believe up until version 5.3.0, it wasn't possible to mock PHP's built-in functionality; which includes such functions as `time()`, `rand()`, `array_key_exists()`, `push()`, and so on.

However, with the advent of namespaces, and the namespaces fallback[6], it now is. If you have a second, have a read through that section of the PHP manual. If not, I want to draw your attention to one key sentence:

> *For functions and constants, PHP will fall back to global functions or constants if a namespaced function or constant does not exist.*

What this means is we're now in a position to play around with PHP's internals, while not actually changing them. Think about the possibilities this opens up. Let's not just think about them, let's get in and find out with some code.

PHP Mock

Specifically, we're going to experiment with a library which facilitates this very functionality; it's called PHP-Mock[7]. Written by Markus Malkusch, it allows for the mocking of PHP's built-in functions, including `time()`, `exec()`, and `rand()`. In the words of the project itself, PHP-Mock:

> *Is a testing library which mocks non deterministic built-in PHP functions like `time()` or `rand()`. PHP-Mock uses that feature by providing the namespaced function.*

As you can infer from that description, it's not as powerful at mocking as Ruby's implementation. But, for our purposes, it's perfectly adequate. So, let's install it. As always, let's use Composer to take care of the heavy lifting, by running the following Composer command in the root of your project:

```
composer require --dev php-mock/php-mock
```

After a minute or so, if that, you're ready to go. In your favorite editor or IDE, create a new PHP file with the contents in Listing 1 graciously borrowed from the official documentation for Mocking built-in functions[8]. Now, let's step through it together.

Up until now, what we've done is used a `MockBuilder` object to indicate we're going to override the built-in `time()` function, using the body of the closure passed to the `setFunction()` method. Instead of returning the current system timestamp, it will instead return the fixed timestamp value 1417011228.

With the mock built, we can now run some assertions upon it, which further demonstrate the functionality of the library. At the point that we run the first assertion, the monkey patch for the `time()` function is not in effect, as `$mock`'s `enable()` method's not been called.

LISTING 1

```
01. <?php
02.
03. namespace foo;
04.
05. require_once('vendor/autoload.php');
06.
07. use phpmock\MockBuilder;
08.
09. $builder = new MockBuilder();
10. $builder->setNamespace(__NAMESPACE__)
11.     ->setName("time")
12.     ->setFunction(
13.         function () {
14.             return 1417011228;
15.         }
16.     );
17.
18. $mock = $builder->build();
19.
20. // The mock is not enabled yet.
21. assert(time() != 1417011228);
22.
23. $mock->enable();
24. assert(time() == 1417011228);
25.
26. // The mock is disabled and PHP's built-in time() is called.
27. $mock->disable();
28. assert(time() != 1417011228);
```

5 *the March 2016 issue:* https://www.phparch.com/magazine/2016-2/march/

6 *the namespaces fallback:* https://www.php.net/language.namespaces.fallback

7 *PHP-Mock:* https://github.com/php-mock/php-mock

8 *Mocking built-in functions:* https://github.com/php-mock/php-mock-prophecy

```
assert(time() != 1417011228);

$mock->enable();
assert(time() == 1417011228);

$mock->disable();
assert(time() != 1417011228);
```

We can test for that by comparing the value returned from calling `time()` against the string which our patched version will return. When doing so, we'll find they don't match.

After that, we enable our patch, by calling `enable()`, and verify it's working by asserting the value returned from calling `time()` does match what it should with our patch in place. Then, we see how we can later disable the mock, and return to the inbuilt functionality by calling `$mock`'s `disable()` function.

This, to me, shows the library's had thought put into it. You must *consciously* choose to enable the patch. You can use it intermittently, on an as-needed basis. It's not a case of once on, always on. I'd like to think it will be that much harder, as a result, to create inadvertent bugs in your tests.

If you'd prefer not to use a closure, to override the functionality of a given function, you can instead use a `FixedValueFunction` object. This lets you directly specify a value to return, whether that's a string, integer, and so on. To use it, we'd replace the call to `setFunction()` from before, with a call to `setFunctionProvider()` as follows:

```
->setFunctionProvider(
   new FixedValueFunction(1417011228)
);
```

Here, our patched copy of `time()` will still return the same value, just supplied slightly differently.

PHP Mockery Mock

Now, this is all well and good. But, perhaps you're concerned about having to learn yet another library, as you're already familiar with others, such as Mockery[9] and Prophecy[10].

If that's the case, don't sweat it. PHP-Mock works with both via extensions. Let's see how to use it with Mockery, my currently preferred mocking library.

As always, we'll need to install the Mockery extension[11], again using Composer. To do so, run the following command, in the root of your project directory:

```
composer require --dev php-mock/php-mock-mockery
```

When the library's installed, in a new file, perhaps called `php-mockery-mock.php`, add the code in Listing 2.

There's not a lot to this, which is handy. Here, we make a call to the static `mock()` method, passing in two parameters;

9 *Mockery:* http://docs.mockery.io/en/latest/

10 *Prophecy:* https://github.com/phpspec/prophecy

11 *the Mockery extension:* https://github.com/php-mock/php-mock-mockery

LISTING 2

```
01. <?php
02. namespace foo;
03.
04. require_once('vendor/autoload.php');
05.
06. use phpmock\mockery\PHPMockery;
07.
08. // Other Mockery related code...
09.
10. $mock = PHPMockery::mock(__NAMESPACE__, "time")
11.       ->andReturn(3);
12. assert (3 == time());
13.
14. // When all tests are finished, remember to clean up
15. \Mockery::close();
```

LISTING 3

```
01. <?php
02. namespace foo;
03.
04. require_once('vendor/autoload.php');
05.
06. use phpmock\prophecy\PHPProphet;
07.
08. $prophet = new PHPProphet();
09.
10. $prophecy = $prophet->prophesize(__NAMESPACE__);
11. $prophecy->time()->willReturn(123);
12. $prophecy->reveal();
13.
14. assert(123 == time());
15. $prophet->checkPredictions();
```

these are the currently defined namespace, and the name of the method that we're going to monkey patch.

After that, we make a call to `andReturn()` supplying the value we want to return, when that method is called. As in the first example, we can test it using `assert()`.

One thing to note about the Mockery extension is that there is no call to enable the patch. So once a method is patched, then it's done for all tests. After all of your tests are done you have to call Mockery's `close()` method. This collectively disables **all** patched methods.

PHP Prophecy Mock

Let's have a look at how to use PHP-Mock with Prophecy. As with Mockery, we first need to install the extension for Prophecy[12]. Using Composer, we do this by running the following command, from the root of our project:

```
composer require --dev php-mock/php-mock-prophecy
```

With that done, let's create one final PHP file, called `php-prophecy-mock.php`. In it add the code shown in Listing 3.

12 *the extension for Prophecy:* https://github.com/php-mock/php-mock-prophecy

Similar to how the library works with Mockery, we don't have to do much to patch a method. First, create a new `PHPProphet` object, which will store our patched functionality.

On it, call the `prophesize()` method, passing in the name of the namespace within which we will create patched functionality. The example above uses PHP's `__NAMESPACE__` directive to specify the current namespace.

Doing so will return an object implementing `ProphecyInterface`, similar to how Prophecy typically works. On that object, call the method you want to patch, as though it were a method on the object, and then call the `willReturn()` method to specify the value the patched method should return.

Next, call reveal, to "reveal" the prophecy. If you're familiar with using Prophecy, all of this should be pretty standard. As with the Mockery extension, the Prophecy extension doesn't have an `enable()` method.

So, once a patch is created, it's in effect for that namespace. However, also like Mockery, they can be globally disabled. This is handled by the `checkPredictions()` method.

Something to Keep In Mind

There's just one thing to bear in mind when using these libraries. The function's namespace **cannot** be qualified. Any functionality which you want to patch has to be within the current namespace. So you can't patch `\DateTime`. But, you could patch `DateTime`. Don't let that trip you up. If you need to

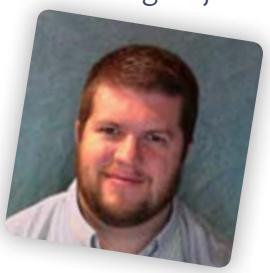
In Conclusion (tl;dr)

That's been a whirlwind introduction to monkey patching in PHP, specifically within the confines of testing. While I'm not against using the functionality, it's not something I would wholeheartedly encourage, for the reservations stated above.

However, in the right circumstances, when applied with due thought and discipline, I don't see any harm in it. What's more, like interactive rebasing in Git, I see it as a good tool to make use of. Just don't get carried away with it.

Matthew Setter is an independent software developer who specializes in creating test-driven applications, and a technical writer http://www.matthewsetter.com/services/. He's also editor of Master Zend Framework, which is dedicated to helping you become a Zend Framework master? Find out more http://www.masterzendframework.com/welcome-from-phparch.

Understanding Objects



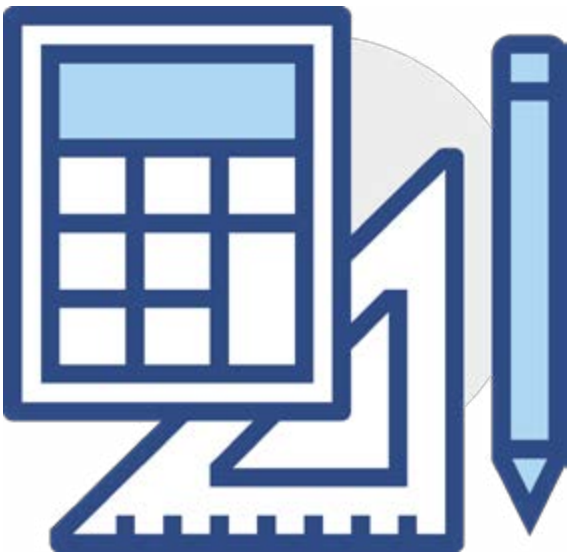
David Stockton

As developers, we understand how our programs work. Perhaps that's an assumption. Some of us understand how certain aspects of our code works. Others, but fewer, understand all aspects of how the code we write works. As developers who are improving and making the transition from junior to mid-level and to senior and beyond, our understanding increases as we learn the ins and outs of the language, the application and the techniques that work to build successful applications. Over the next few months, we'll be building up a solid foundation of Object-oriented concepts and patterns, how they work, and how Object-oriented code can be used to build more maintainable software.

Some of you are likely quite familiar with Object-oriented programming (OOP). This article may not be for you, but I hope there are some aspects I'll cover that will be new or open up new ways of thinking. This month, though, I want to address developers who want to learn more about OOP.

Some Background

I realized a few days ago that sometime this year I'll have been writing code for 30 years. I've been doing it professionally (read: getting paid for it) for nearly 20. I started with BASIC on an Apple II clone. In college, I learned enough Fortran to test out of the class and then went to C, C++, and Java. We covered the concept of "class" in C++, sort of, but despite being an object-oriented language, there was really no discussion or learning about objects in the Java class. We all just learned that our classes filled with functions all worked if the function definitions started with `public static` and they stopped working if we removed static or changed public to something else, even though the code could compile.



In 1997, I was tasked with building a website which allowed users to find information about articles concerning women in tech and computing. I'd built some basic sites with HTML and some garbage JavaScript that would animate elements around the page, thanks to Adobe GoLive, but I hadn't yet built anything that would dynamically create different pages based on user input. I did some research, considered Bash for the job, quickly found it was going to be a nightmare and went back trying to find answers for something that would work. During my quest to find a solution, I stumbled on ASP, but it required an expensive development tool and web server.

I even considered writing it in C and using CGI, but fortunately, I stumbled on PHP and some tutorials on webmonkey.com. They were undoubtedly awful and full of security issues, but I had no idea at the time. Every site and tutorial I found (thanks, Alta Vista, Yahoo! and WebCrawler, and a little bit later Google) relied on register globals; database examples included queries built with string concatenation, usually directly from PHP's superglobals. PHP 3 didn't support objects. The scripts were easy to follow, mostly. You started at the top, read down, and everything was mixed together. HTML, code, SQL queries. In some cases, there was a full file of just functions which would be `require_onced` into another file, but that was typically the extent of the organization and structure.

Shortly after I started learning PHP on the Homesite web server, PHP 4 was released. PHP 4 had some primitive object support, but nothing like what we've had since around PHP 5.2. I remember trying to learn OOP with my procedural background was a struggle. The early examples I found were ill-conceived, and there wasn't a lot of explanation. I came to the conclusion OOP was not something that helped with anything.

With PHP 5, the language improved and gained better support for OOP over the years. While PHP still supports coding procedurally, much of the current code you'll see written in the language is Object-oriented, or taking advantage of the OOP features PHP brings. And, here we are now. I've been developing my PHP code using OOP concepts for more than a decade

now. I want to share some of what I've learned in hopes that I can help you come to a solid understanding much more quickly than I did, by avoiding some pitfalls which took me a long time to get around.

There are plenty of excuses I've heard to avoid OOP. To truly take advantage, it requires thinking in a different way than procedural code. Some people argue their application isn't complicated enough to warrant using OOP. That was one of my excuses before I saw a good example. Properly modeling objects with data and behavior takes practice. On the other hand, properly designed classes and objects lead to code that is easier to test, as well as code you can reuse in ways you may not have conceived of originally. I've found that the simpler the classes, the easier it is to use in different ways. Some people feel OOP takes longer to write. In some cases, this may be true since it does take time to design objects. However, as far as actually writing the code, it's faster in many cases. If the objects have one job, then being able to focus on only that small part means the code can be simple and straight-forward.

Understanding Objects

When you're starting out with OOP, many of the examples start with ideas and concepts and objects which have real-world counterparts. Whether it's trying to explain the concept of inheritance using animals or vehicles, or building shapes which can all calculate their area or perimeter using the same method names, thinking of the objects of code having the same sorts of constraints as real-world objects can often lead to an incorrect mental model of what they are and what they can do. As such, if I use one of these examples to help illustrate a concept, it helps to separate and remove the limitation of objects needing to behave like, or even have a real-world equivalent at all.

Let's take a quick look at what I mean. Suppose there's a `Car` class that does all sorts of Car things. Cars need engines, so we give the car an engine.

```
$engine = new Engine();
$myCar = new Car($engine);
$yourCar = new Car($engine);
```

In the example above, we have two separate and distinct objects that are cars, and they have an engine. They can do all the things cars with engines can do. However, both cars have the exact same engine. In the real-world, this is ridiculous. Every car in the real-world needs its very own engine. But in OOP, it's perfectly legitimate and often desirable to provide the same object to other objects at the same time during the same execution of a program. However, there are design decisions which need to be considered when doing this sort of thing.

Consider the two cars with one engine example. Suppose the car provides us some methods like `turnOn` and `pressGas`. We build our class so that if the car is on, pressing the gas will move it forward, and if the car is off, pressing the gas has no effect. So taking our two cars we run the following code:

```
$myCar()->turnOn();
$yourCar()->turnOn();
$yourCar()->pressGas();
$yourCar()->turnOff();
$myCar()->pressGas();
$myCar()->turnOff();
```

If the intent of the code was that both cars were going to drive forward, the reality is the way we've built the code would probably result in something else happening. Since the engine is shared between the cars, the engine will be turned on by both cars, at which point the engine is running. Then, the `$yourCar` gas pedal is pressed, and the car lurches forward. Then, it shuts off the engine.

The code continues with `$myCar` and presses the gas. However, since the engine is shared, it's now off

and `$myCar` goes nowhere. There are numerous ways to design the system differently, so this is not an issue, but my point is that we still have to build and design systems in such a way that they do what we want and don't do what we don't want.

You may be wondering why there's only one engine for both cars. In the Zend Engine, and in other languages, objects are assigned by reference so `$engine` always refers to the same instance of an object, even if we pass it into more than one object that needs an engine. This wasn't the case in PHP 4.

Classes and Objects

Object-oriented programming starts with classes. You may have heard the terms used interchangeably, but they are different. Classes are a description of the shape of the data, behaviors, and dependencies of an object. An object is an instance or instantiation of a class. To put it another way, and using another analogy, a class is related to an object in much the same way a blueprint relates to a house. With that out of the way, let's continue.

Understanding Scope

Scope in terms of programming refers to what parts of a program can see or use it. PHP supports a number of different scopes for variables. First

up, we have superglobals[1]. These are variables that are typically defined and assigned by PHP, in most cases before your script starts. They include variables like `$_SERVER`, `$_REQUEST`, `$_GET`, `$_POST`, and others. `$_SESSION` is initialized when you start a session. They are accessible by everything in your program without the `global` keyword. It's usually in your best interest to avoid using these in your code directly. It's better to abstract usage of these into as few places in code as you can. I prefer to let whatever framework I'm using deal with those variables usually.

Next, we have globals[2]. These are also variables that are available everywhere in your program. Unlike superglobals, unless the code we're talking about is not defined in a function or class, you have to tell PHP that you want to access the global. Again, this should be avoided at all costs. Globals can be dangerous in that they lead to unpredictable execution and bugs which are difficult to track down. Because they can be changed by anything within your code, changing the order that certain bits of code run can cause it to have a completely different result.

Once we introduce classes, there can be variables at the class level. These are class static variables. They belong to the class, not an object or instance.

```
class Samoflange
{
   public static $doohickey;
}
```

If a static class variable is defined as public, it is effectively a global variable, but rather than accessing it via `$doohickey` it is accessed via `Samoflange::$doohickey`. Whether these variables are defined as public, protected, or private, they are all effectively global. The protected and private static class variables are just globals that cannot be read from everywhere.

Next, we have instance variables or variables that belong to a single instance of a class. These are the fields or attributes that can be used to hold on to state and dependencies of an object.

Finally, we have the local variable scope. These variables are defined, used, and eventually destroyed when the method or function completes. By the way, if you've heard the term "method" before and not understood what that is, it's just a function defined within a class. The value of local variables can live beyond a function or method if they are returned.

1 *superglobals:* https://www.php.net/language.variables.superglobals

2 *globals:* https://www.php.net/language.variables.scope

Visibility Modifiers

When starting out, it seems that knowing as much as possible about how every aspect of the code works is important. For a programmer, knowing it all is good. However, from the perspective of the code, it's best to limit what any given piece of code knows about as much as possible. To use another real world analogy for this point, imagine how difficult it would be to drive a car if we had to know every aspect of how a car works. If we were required to understand that pressing on the gas moves a lever (the pedal), which pulls a cable, which is attached to another lever which opens a… and the…, so more gas is pumped through the… and then, you know… um, stuff happens and the car goes faster. If we were required to know or keep in mind all the workings of the car's systems in order to drive, we'd probably have a lot fewer drivers and very few people would know how to drive more than a single vehicle. Instead, all of the complex workings of the car are hidden behind a few common interfaces that are shared between different vehicles. We have gas pedals, brake pedals, and a steering wheel. Whether your car has a rack and pinion, power steering, articulated steering, or something else, the steering wheel abstracts that all away so all we need to know is when we turn the wheel clockwise, the vehicle goes to the right, and counter-clockwise goes to the left. It means in general if you know how to drive one car, you can drive almost any other car.

> *This hiding of information and functionality is called encapsulation in OOP terms. It means we should only expose the minimal amount of information we can for the system to work.*

This hiding of information and functionality is called encapsulation in OOP terms. It means we should only expose the minimal amount of information we can for the system to work. PHP provides visibility modifiers we can use to allow the language to enforce what parts of the program can see variables or access a method. Some other languages (like Python) don't have this and rely on convention, and variable naming to infer that certain variables or functions should not be used by developers. You may see old PHP code where variables and functions are prefixed with an underscore (_), which was how PHP 4 code hinted at visibility and trusted convention to enforce it.

PHP provides three levels of visibility in objects: `public`, `protected`, and `private`.

Public variables and methods are available to anything in our code with access to the instance of that object, as well as any code in the object itself.

Protected hides variables and methods to the outside world, and makes sure the code is only available within the class and its inheritance hierarchy. Don't worry if you don't know what I mean by inheritance; we'll get to it later.

> *I want to talk about inheritance, mostly to say in most cases, inheritance is not the right way to solve a problem.*

Finally, private variables and methods are available only to instances of that class. This is not a typo. In addition to private variables and methods being usable within a single instance, if another class of the same type has access to a different instance, it can access private variables and call private methods *on that other instance*.

Inheritance of Classes

I want to talk about inheritance, mostly to say in most cases, inheritance is not the right way to solve a problem. However, since it seems every introduction to OOP includes a discussion on inheritance, I shouldn't pretend it doesn't exist. Inheritance allows a class to be built based on another class. If you can describe the relationship between two classes with "is a" then inheritance *might* be a good solution. However, if the relationship is not "is a," then it is definitely the wrong solution.

In PHP we use inheritance with the `extends` keyword. For example:

```
class Foo extends Bar {
}
```

In this case, we're saying a `Foo` "is a" `Bar`. Foo should be providing some different but related functionality, potentially overriding or augmenting existing functionality. Over the next few articles, I hope to show and convince you (if you don't already believe it) that we should prefer composition over inheritance. This means that we can build powerful solutions by combining objects and delegating work from one object to another, rather than making a bunch of objects that inherit and change functionality.

If inheritance is not your thing, or you feel that a class should never be extended, you can declare it `final`. This ensures at the language level that nothing else can extend it. Marco Pivetta wrote a good post arguing that classes which implement an interface, and no other public methods should always be marked final. You can read it at http://phpa.me/ocramius-final-class

Do One Thing and Do It Well

I've said for years that if you have to use the word "and" to describe what the purpose of your class is, then it's doing too much. The same can be said at the method level and arguably at the namespace or package level as well. In OOP, we get better, more powerful code by building simple objects and combining them. If you're building a class and you find the functionality you're creating could be delegated or moved out, chances are it's a good idea to do that. It may end up that your original design was fine with the functionality happening in a single object, but later as requirements change and the software evolves, more classes and objects may make themselves apparent.

Creating classes that are limited in what they do as well as what is exposed via the public visibility modifier means maintenance of the code will be easier, testing it will be simpler, and refactoring, updating, adding and changing functionality will be more straightforward and cheaper.

Conclusion

Over the coming months, we'll focus more on different patterns I see and use in code, talk about why those patterns exist, what they are good for, and how they should be used. This month, I talked some aspects of my philosophy towards Object-oriented programming and design which should help in understanding why I build some things in the way I do. I hope you'll join me next month. If you've got a particular pattern or question about Object-oriented design or programming that you'd like to read about, please be sure to reach out and let me know.

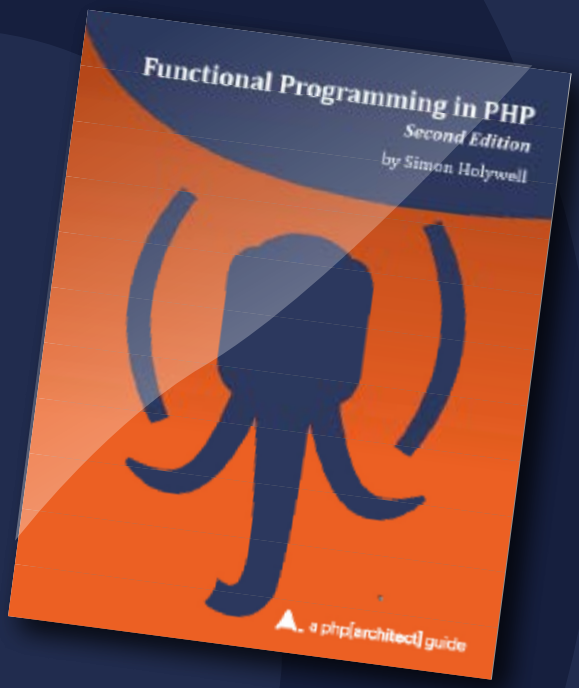
David Stockton is a husband, father and Software Engineer and builds software in Colorado, leading a few teams of software developers. He's a conference speaker and an active proponent of TDD, APIs and elegant PHP. He's on twitter as @dstockto, YouTube at http://youtube.com/dstockto, and can be reached by email at levelingup@davidstockton.com.

Functional Programming in PHP

Second Edition

by

Simon Holywell



Many languages have embraced Functional Programming paradigms to augment the tools available for programmers to solve problems. It facilitates writing code that is easier to understand, easier to test, and able to take advantage of parallelization making it a good fit for building modern, scalable solutions.

PHP introduced anonymous function and closures in 5.3, providing a more succinct way to tackle common problems. More recent releases have added generators and variadics which can help write more concise, functional code. However, making the mental leap from programming in the more common imperative style requires understanding how and when to best use lambdas, closures, recursion, and more. It also requires learning to think of data in terms of collections that can be mapped, reduced, flattened, and filtered.

Functional Programming in PHP will show you how to leverage these new language features by understanding functional programming principles. With over twice as much content as its predecessor, this second edition expands upon its predecessor with updated code examples and coverage of advances in PHP 7 and Hack. Plenty of examples are provided in each chapter to illustrate each concept as it's introduced and to show how to implement it with PHP. You'll learn how to use map/reduce, currying, composition, and more. You'll see what external libraries are available and new language features are proposed to extend PHP's functional programming capabilities.

Buy Your Copy Today!

http://phpa.me/functional-programming-in-php-2

Uncle Cal's Thank You Letter

Cal Evans

As we start out the new year, I did what I do every year at this time, reflect back. One of the things that came to mind this year—I'm not sure why—is a letter I wrote for roughly 40 people back in 2010. I meant for a letter of this sort to be an annual tradition, but sadly, life intervened.

Last year, we saw continued adoption of PHP 7 with goodies like static type hints. The core team capped off the year with the release of PHP 7.1. Six years on, we have more events, user groups, and contributors in our corner of the web. This year, instead of selecting only certain people for it to go out to, I want to send it out to everyone in the PHP community: core developers, extension writers, documentation maintainers, user group leaders, conference organizers, and PHP programmers around the world. This is Uncle Cal saying "Thank You," to you.

`Hi!

I wanted to take a moment as we begin the new year, to say thank you. Each and every one of you reading this has contributed something to the PHP community over the past year. For that, I am in your debt.

Some of you are core developers and are the heroes of PHP. You make it possible for me to work with this great language and community. I could send you a letter like this every day for the rest of my life and wouldn't be able to adequately express the positive impact your work has had on my life. Thank you.

Some of you lead user groups around the world. You deserve to be put up on a pedestal and worshiped for the sacrifices you make personally to start and keep your group going just to help others learn. Helping developers is a passion of mine;you will always have a warm spot in my heart for your contributions. Thank you for your efforts and keep up the good work.

Some of you work on projects built around PHP. You volunteer your time and talent to build something that others will use. Your work touches lives around the world and makes it possible for people to build systems that to us may seem trivial or silly, but to them, it's an expression of their passion. Thank you for your time, your talents, and your imagination for what could be.

Some of you run businesses built around PHP. Thank you! To thrive, PHP has to have a vibrant ecosystem. I know it's not easy and there are a lot of naysayers out there but seriously, thank you for helping ensure that PHP thrives and succeeds.

Finally, some of you are just grunts like me, doing what you can to give back. I know the issues you face, I know how difficult it can be sometimes. However, I also know the rewards you get, when someone tells you after a talk, "That made things click for me!" So thank you for all you do and let me encourage you to keep going even when you don't get thanks. Sometimes people don't speak up; that doesn't mean you didn't help them.

That's all I've got. Here's wishing that you and yours have a safe and wonderful new year!

Sincerely,

=C=

Cal Evans

Nerd Herder to the Worldwide Herd

Past Events

December 2016

SymfonyCon Berlin 2016
December 1–3, Berlin, Germany
http://berlincon2016.symfony.com

ConFoo Vancouver 2016
December 5–7, Vancouver, Canada
https://confoo.ca/en/yvr2016

PHP Conference Brazil 2016
December 7–11, Osasco, Brazil
http://www.phpconference.com.br

MageCONF'16
December 10, Kiev, Ukraine
http://mageconf.com

Upcoming Events

January

PHPBenelux Conference 2017
January 27–28, Antwerp, Belgium
https://conference.phpbenelux.eu/2017/

February

SunshinePHP 2017
February 2–4, Miami, Florida
http://sunshinephp.com

PHP UK Conference 2017
February 16–17, London, U.K.
http://phpconference.co.uk

March

ConFoo Montreal 2017
March 8–10, Montreal, Canada
https://confoo.ca/en/yul2017/

Midwest PHP 2017
Bloomington, Minnesota,
https://2017.midwestphp.org

April

PHP Yorkshire
York, U.K.
https://www.phpyorkshire.co.uk

DrupalCon 2016
April 24–28, Baltimore, MD
https://events.drupal.org/baltimore2017

May

phpDay 2017
May 12–13, Verona, Italy
http://2017.phpday.it

php[tek] 2017
May 24–26, Atlanta, Georgia
https://tek.phparch.com

PHP Tour 2017 Nantes
May 18–19, Nantes, France
http://event.afup.org

PHPSerbia Conference 2017
May 27–28, Belgrade, Serbia
http://conf2017.phpsrbija.rs

International PHP Conference 2017
May 29–June 2, Berlin, Germany
https://phpconference.com

June

PHP South Coast 2017
June 9–10, Portsmouth, UK
https://2017.phpsouthcoast.co.uk

These days, when not working with PHP, Cal can be found working on a variety of projects like Nomad PHP. He speaks at conferences around the world on topics ranging from technical talks to motivational talks for developers @calevans.

New Year's Security Resolutions

Chris Cornutt

Another year is in the bag, and the state of security in PHP is at an all time high. Sure, there's still plenty of poorly written and unsecured code out there. However, there were several new advancements, in both the language and its ecosystem making it easier than ever to secure your applications.

The State of PHP Security

In December of 2015, we were presented with the most major update to the PHP language in years—PHP 7. This version came with a whole host of new features including several that relate more to the security side of things (like the "filtering" allowed with unserialize). Since its release and the release of follow-up versions, over 80 CVE (Common Vulnerabilities and Exposures) security issues have been identified and resolved, usually within a few weeks of them being discovered and reported. The PHP development group and all of its members have done an excellent job doing the hard work to make PHP as secure as it can be.

The beginning of December 2016 saw the release of PHP 7.1, the first "minor" release in the post-PHP 7 world. This release added plenty of new features in its own right including the start of a long road for a lot of developers—the deprecation of mcrypt[1] support in PHP core. The mcrypt library itself is way out of maintenance, having been abandoned in 2007. It's about time PHP deprecated it in favor of other functionality like OpenSSL. There are a *lot* of libraries out there based on mcrypt, so this could be a major shift for many tools. Fortunately, OpenSSL support comes standard in just about every PHP installation these days so making the transition should be relatively easy with only a few "gotchas" to look out for.

Sharing the Knowledge

Along with all of these core changes, there have been a lot of tools and articles released to help you along the path to more secure code. Right here, in php[architect], there have been some great security-related articles published over the past year including:

- *Drupal Security: How Open Source Strengths Manage Software Vulnerabilities* by Cathy Theys
- *Learn from the Enemy: Securing Your Web Service* Parts 1 & 2 by Edward Barnard
- *Implementing Cryptography* by Edward Barnard

In addition to these articles they've also released a "Web Security" anthology by combining these and several others into a single book, Web Security 2016[2].

Another group promoting security in PHP is The Paragon Initiative group[3]. They've used both their blog and development work to teach PHP developers security from the start and teach best practices for security current applications. Their blog covers topics like:

- public key encryption in PHP
- secure software updates
- securing account recovery
- cryptographically strong random number generation
- secure serialization of data

There's plenty more out there, so be sure to check out the Paragon Initiative blog[4] for lots of great articles.

Some Security New Year's Resolutions

It's a new year, and there's still plenty of work to be done when it comes to securing PHP applications. While there were a lot of advancements in the previous years, there are still many things that have to be done manually because of the nature of the language. It's a habit this time of year to have some New Year's resolutions; things you want to improve over the course of the next year. I'd like to propose a few development security related items you could add to the list to help you be more proactive when it comes to security.

Don't Just Think "Happy," Think "Bad" Too

It's all too easy in our day-to-day development to think along the "happy path." We write our code and madly click until things work as expected and (hopefully) bug-free. We pat ourselves on the back and move along to the next task, sometimes building on the previous feature and others moving on to a completely different topic. We're all guilty of it, and the "happy path" is what meets the deadlines and makes product managers happy.

However, what happens when something breaks? What happens when a bug report is filed that an endpoint a developer forgot to secure is spewing private customer information out for all the world to see? It's possible that, if the developer had thought like an attacker about the work they were doing they would have realized relying on "security through obscurity" is never an effective method. Had they

1 *deprecation of mcrypt:* https://wiki.php.net/rfc/mcrypt-viking-funeral

2 *Web Security 2016:* http://phpa.me/web-security-2016

3 *The Paragon Initiative group:* https://paragonie.com

4 *Paragon Initiative blog:* https://paragonie.com/blog

protected this chunk of code the "pants on fire" feeling they had when the bug report came in could have been avoided.

When you're working through your code, sit back for a second and think about how it could be abused. Consider the most common kinds of attacks out there today (XSS, CSRF, SQL injection, etc.) and see if the code you just wrote could be vulnerable. The larger the application, the more tricky this becomes but there are many tools out there to test for these things and libraries to help prevent them.

Validate All the Things

The major issues in web applications boil down to one thing—improper validation practices. Much like the "happy path" development practices, it's all too easy to fall into the trap of thinking the input into your application will only come from the forms you provided or some other "trusted" source. You set up your inputs and pull in the values for use on submit, but how can you be sure what you're using isn't tainted?

Validation of all data, most importantly user input, should be a requirement on your resolution list for this year. Most frameworks make it simple to validate the incoming data on your requests with handy libraries or, if there isn't one, it can be included easily with Composer—like Respect/Validation[5]. Trust me; there's nothing like the satisfaction the data you're working with is exactly what you're expecting and that there are no sneaky attackers worming their way in.

Implement a Development Security Workflow

Most larger development groups (and some smaller ones) have embraced the idea of continuous integration and, possibly, continuous deployment. They've set up their automated testing and workflow tools to notify them of breakage in their test or other resources. They've defined coding standards and have sniffs to detect issues with syntax. Maybe they've even set up a mandatory review process before any code can make it to production. All of these are great, but if you don't have any security-related step in your workflow you're missing a major component.

One of the biggest complaints I hear from developers is that worrying about security "takes too much time" or it's "just one more thing to worry about" when they're writing their code. Developers who talk like that have something in common—their efforts to integrate secure development into their workflow have security as an "add on" closer to the end of the development process. They're given the command "…and now make sure it's secure," but only after they've completed the "happy path" through their code. This results in developers feeling like they have to go back and wade through all the work they've just done to look for potential security issues.

The real key to integrating security and secure development into the workflow is to "shift left" and move the security evaluation back before the development work is even done; back to the planning stages. New features and changes should be evaluated prior to code being written or changed to see what possible threats it could be vulnerable to. These findings should then be passed over to the developers writing the actual code to keep in mind as they're hacking their way to success. By introducing it at this level, you're setting the development up for a more secure end result and making your development group much happier in the long run.

Keep It Simple

As a final resolution to add to your list, I want you to remember to "think simple" when it comes to securing your applications. It's all too easy to come up with grandiose plans that rely on multiple moving pieces to protect your code. Maybe it's coming up with some kind of special "encryption" scheme for your data (*hint: don't do this*) or maybe it's implementing a more complex than required authentication mechanism when just a username and passphrase will do.

Keeping it simple applies to data access in your application too. I've seen plenty of applications which require a series of complex steps to filter out or modify data based on who is accessing it. Unfortunately, these steps are implemented different ways in different places, resulting in potential exposure of sensitive information. Rules and functionality like this should be kept in a standard place and should be something which can easily be applied to any data set across the application with very little effort.

Remember, complexity is one of the main enemies of security. The more complex a system is the harder it is to secure. Keeping things simple and approaching them from both a "fail fast" and "principle of least privilege" mindset helps you create more secure and robust code that's easier to work with in the long run.

A Secure and Safe 2017

So, here's to wishing you a secure and safe start to 2017! I hope you'll take some of these resolutions to heart and apply them during your day-to-day development. It's amazing how just a few simple steps can help protect your code (and your company) from embarrassing or damaging breaches. I hope over the course of 2016 I've provided you with some helpful hints, good resources, and methods you can use in your code. It's a new year, and there's plenty of work to be done—let's get out there and drive security forward!

For the last 10+ years, Chris has been involved in the PHP community. These days he's the Senior Editor of PHPDeveloper.org and lead author for Websec.io, a site dedicated to teaching developers about security and the Securing PHP ebook series. He's also an organizer of the DallasPHP User Group and the Lone Star PHP Conference and works as an Application Security Engineer for Salesforce. @enygma

5 *Respect/Validation:* http://phpa.me/respect-validation

December Happenings

PHP Releases

PHP 7.1.0:
http://php.net/archive/2016.php#id2016-12-01-3

PHP 7.0.14:
http://php.net/archive/2016.php#id2016-12-08-1

PHP 5.6.29:
http://php.net/archive/2016.php#id2016-12-08-2

News

Freek Van der Herten: Symfony and Laravel will require PHP 7 soon

As Freek Van der Herten mentions in this recent post to his site, it was announced by both Fabien Potencier (Symfony) and Taylor Otwell (Laravel) that the upcoming versions of the frameworks—Symfony 4 and Laravel 5.5—will require PHP 7 by default. Freek talks some about the improvements that come with PHP 7 and which he thinks will show up in the different frameworks' codebase. http://phpdeveloper.org/news/24711

Codeception Blog: Writing Better Tests: Expectation vs Implementation

The Codeception blog has recent post they've written up talking about writing better tests for your application and the difference between expectation and implementation as it relates back to meaningful tests. They give an example of a test that's "bound to implementation details" from the Magento codebase that relies on a specific function implementation (the `method`). This function is a part of the Symfony functionality, not Magento, and what might happen if things change in your application. They note that the main difference is testing for the result versus testing for the behavior of the functionality. http://phpdeveloper.org/news/24742

Paul Jones: PECL Request Extension: Beta 1 Released!

As Paul Jones has announced in this post to his site the PECL "Request" extension has reached the beta stage with the release of beta v1. The post also lists out some of the new functionality introduced in this beta mostly focused around the fetching of the "forwarded for" information. http://phpdeveloper.org/news/24722

Fabian Schmengler: Collection Pipelines in PHP

In a new post to his site Fabian Schmengler has written up an introduction to collection pipelines and how it could be applied to a Magento-based environment. He starts by illustrating the idea in Bash and Ruby, showing the three main types of collection operations: map, filter and reduce. He talks about the advantages these methods have over traditional looping and what kind of value they can provide in both Laravel and plain old PHP. He illustrates the PHP-only versions using the array_filter, array_map and array_reduce functions and some thoughts on when it's good to use them over normal looping (and when it's not). http://phpdeveloper.org/news/24748

SitePoint PHP Blog: Social Logins with Oauth.io—Log in with Anything, Anywhere

The SitePoint PHP blog has a tutorial posted from Meni Allaman showing you how to use the OAuth.io SDK for social logins, integrating multiple social network logins in one centralized place. The tutorial then breaks down the steps to follow for getting the service set up and getting the required package installed. Following this the author shows how to connect your account to the various services and provides the code you'll need to connect to the OAuth.io service http://phpdeveloper.org/news/24730

Amine Matmati: Symfony: the Myth of the Bloated Framework

Amine Matmati has written up a post with a few quick points refuting the "bloated framework" myth as it relates to the Symfony framework. He then goes on to talk about how, despite many Symfony components being used individually by other projects, the overall framework still has the reputation for bloat. He goes through some of the main points usually mentioned by the opponents. http://phpdeveloper.org/news/24719

Leonid Mamchenkov: Feature Flags in PHP

In a new post to his site Leonid Mamchenkov talks about feature flags, a handy tool you can use in your application to enable/disable features and or risky changes in your code allowing you more production-level control. He talks about some of the challenges that he had in his own feature flag implementation including naming of the flags and where the flags should be placed. He then links to the PHP Feature Flags site and various PHP libraries that implement feature flags slightly differently and cover cookie-based, IP-based and URL-based features. http://phpdeveloper.org/news/24716

Symfony Blog: How to solve PHPUnit issues in Symfony 3.2 applications

On the Symfony blog there's a quick post sharing helpful advice about fixing PHPUnit tests in Symfony 3.2 applications, mostly around an issue involving the use of the "phar" distribution and a class constant error. They provide the commands to get this bridge installed (via Composer) and how to execute the PHPUnit tests post-install (using the `simple-phpunit` command instead). http://phpdeveloper.org/news/24698

Medium.com: The Three Pillars of Static Analysis in PHP

In this post over on Medium.com Ondřej Mirtes looks at what he calls the "Three Pillars of Static Analysis in PHP"—three kinds of testing you can do to catch errors "at rest" in your codebase. He covers some of the things the last option verifies and links to another introductory article about the tool to help you get started. http://phpdeveloper.org/news/24689

Matthew Turland: On Remaining Employable

Matthew Turland has an interesting new post to his site sharing some of his own thoughts on how you can stay employable as a developer with some great suggestions both on the technical and personal side. There's a lot of good content in the post so be sure to give it a read, especially if you're a developer that's been in the same role for a while… http://phpdeveloper.org/news/24680

DaedTech Blog: Avoid these Things When Logging from Your Application

On the DaedTech blog Erik Dietrich has written up a list of a few things he suggests avoiding when using logging functionality in your application. The suggestions range from the actual contents of the message out to some logging best practices. He ends the post with a suggestion of "sensible logging"—capturing as much meaningful information as possible while not overdoing it. http://phpdeveloper.org/news/24666

TutsPlus.com: Building Your Startup: Issue Tracking and Feature Planning

TutsPlus.com has continued their "Building Your Startup with PHP" series today with this latest tutorial covering the creation of an issue tracking and feature planning process with the help of the Asana service. He walks you through getting started with the service (they have a free tier) and how to set up your first project. He includes examples of task creation, tagging tasks and tracking bugs right inside the service. http://phpdeveloper.org/news/24655

Welcome to php[architect]'s new MarketPlace! MarketPlace ads are an affordable way to reach PHP programmers and influencers. Spread the word about a project you're working on, a position that's opening up at your company, or a product which helps developers get stuff done—let us help you get the word out! Get your ad in front of dedicated developers for as low as $33 USD a month.

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

Listen to developers discuss topics about coding and all that comes with it.
www.developershangout.io

The PHP user group for the DC Metropolitan area
meetup.com/DC-PHP

Kara Ferguson
Editorial

Refactoring your words, while you refactor your code.

@KaraFerguson
karaferguson.net

technology : running : programming
rungeekradio.com

The Frederick Web Technology Group
meetup.com/FredWebTech

On Being a Polyglot

Eli White

Why would I, in a magazine dedicated to PHP, be discussing the virtues of being a polyglot? In computer science terms of course.

The reasons are very straightforward. There is a great benefit to be found in exposing yourself to different programming languages and the concepts that they hold. They make you a better programmer overall as you expand your knowledge.

> **pol·y·glot** / *pälē glät/*
> *noun*
>
> *1. person who knows and is able to use several languages.*

We have a huge benefit in the PHP world that is rarely looked at. By our nature, we are already one of the most technology diverse communities. Recently the PHP community posted 24 stories for the first 24 days[1] in December, and in one of those posts[2], was this gem of knowledge:

> *PHP has always been a glue language. Everything we glue together has nothing to do with the language we've chosen. Things like using Varnish for caching, using Cassandra for sessions, building good relational database schemas, effectively using NoSQL data stores, and building work queues with ZeroMQ.*
>
> *—Davey Shafik*

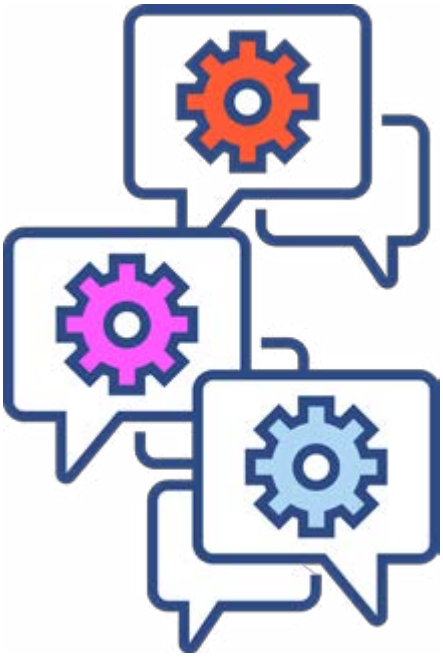
I cannot think of a more true statement about the technology that surrounds PHP. We have always been a community, a technology, based around finding the best other technologies on the web and stitching them all together into a cohesive bundle of tech that somehow works. From the very beginning PHP was built as an add-on to a web server; an extra set of keywords that you put into the HTML itself to make magic happen. For that matter, what PHP programmer out there doesn't also have to dabble in JavaScript. It's truly rare to find that 100% back-end PHP developer who doesn't end up writing JavaScript to go along with it.

PHP itself has grown organically, adding features as it ~~steals~~ borrows ideas that originate in other programming languages. It began as a mix of C and Perl syntax and has added concepts from Java, JavaScript and others. PHP itself could be described as polyglot, being a programming language of multiple ones fused together.

Now we can see that the PHP community is already polyglot. We've embraced Javascript (and other technologies as well) as strongly as our own dear PHP. It's why at PHP conferences, you don't find talk after talk about PHP itself, but a diverse mix of topics ranging from databases to task queues. In fact, recently my current company announced it was hosting CoderCruise[3], a polyglot web programming conference, and I had someone come up to me and thank me because they felt that the PHP community could surely pull off a polyglot event.

I have programmed heavily in Applesoft Basic, Ada, C, Perl, Java, JavaScript, and PHP over the years, and dabbled in dozens more. Each programming language has taught me something new, which I've brought back to PHP itself. So as a New Year Resolution I encourage all of you to expand yourself, to dabble in another language, see what it has to offer you, and bring what you learn back to our PHP community so that we can continue to evolve and grow.

> *The mind of the polyglot is a very particular thing, and scientists are only beginning to look closely at how acquiring a second language influences learning, behavior and the very structure of the brain itself.*
>
> *— Jeffrey Kluger*



1 *24 stories for the first 24 days:* https://24daysindecember.net

2 *one of those posts:* http://phpa.me/24days-php-dead

3 *CoderCruise:* https://www.codercruise.com

Eli White is a Conference Chair for php[architect] and Vice President of One for All Events, LLC. He may be a polyglot in computer science terms, but has never managed to master another spoken language (having had enough issues stumbling through English.) @EliW

www.phparch.com \ January 2017 \ 48

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