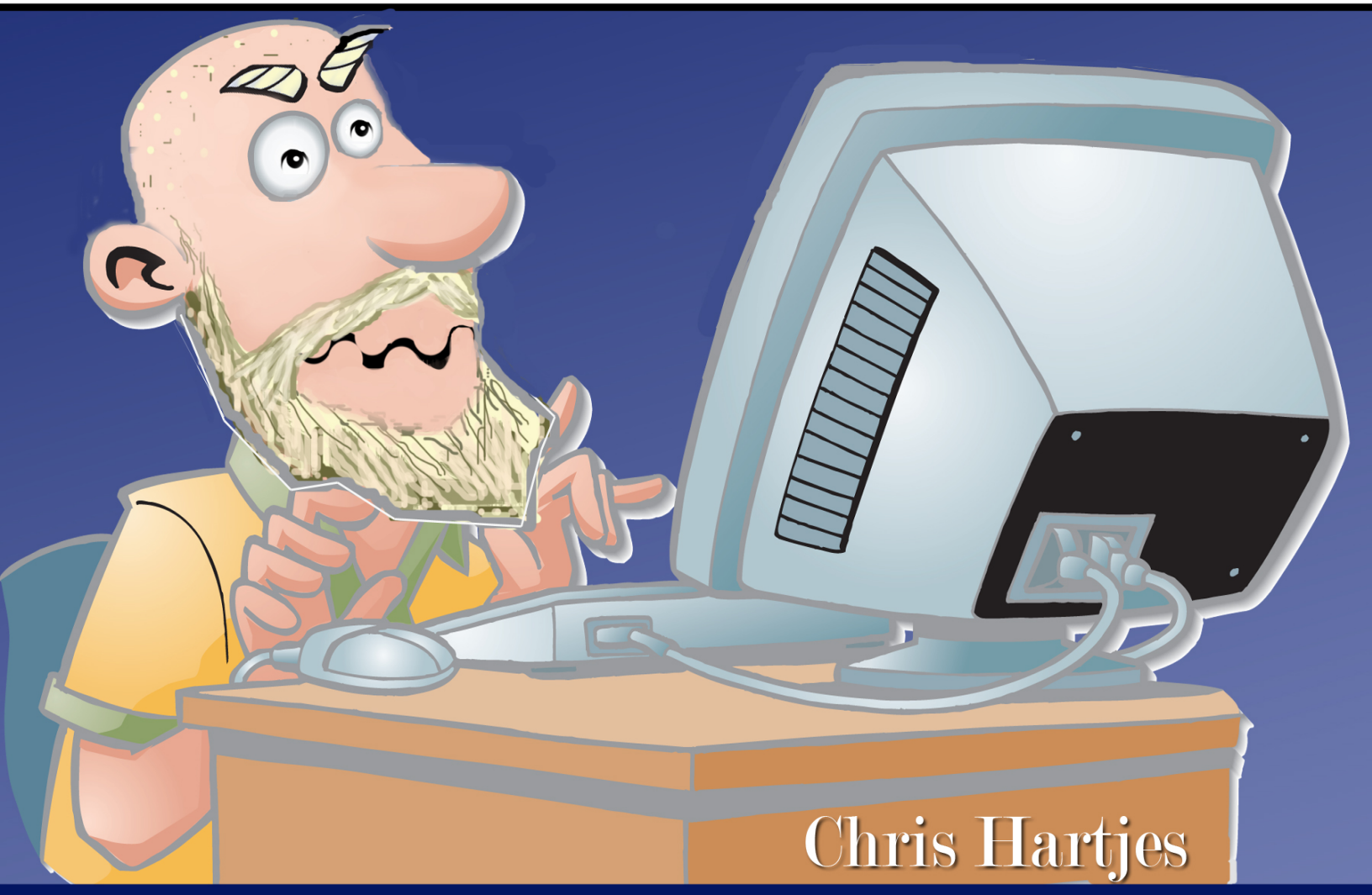
The Grumpy Programmer's Guide To Building Testable Applications In PHP



Chris Hartjes

The Grumpy Programmer's Guide To Building Testable PHP Applications

This is a Leanpub book which is for sale at http://leanpub.com.

Leanpub helps you connect with readers and sell your ebook, while you're writing it and after it's done.

Contents

1

Copyright

ISBN: 978-1-105-45488-2

2

Thanks

First, I'd like to thank my technical reviewers. They caught all the details that I missed and made this guide what it is. So thanks to (in no particular order) Peter Meth, Joel Perras, Matthew Turland, and Remi Woler for their help. Better guides come from the input of others.

Second, I'd like to thank those in the PHP community who are always pushing the envelope, always speaking loudly and clearly about the importance of best practices while trying to give back to the community at the same time. Those people inspired me to keep writing and speaking about topics that interest me and hopefully interest you. I may feel out of sync at times with the "mainstream" PHP community but if my online ranting and raving has inspired other programmers to step up their own game, then I am grateful I had an impact.

Finally, I'd like to thank my wife Claire for sticking with me on my journey from a fresh-out-of-college newbie programmer to the well-rounded, well-seasoned, extremely grumpy programmer I am today. Thanks for putting up with the long hours that a programmer spends staring at screens instead of staring into their partner's beautiful eyes. My wife is my partner in everything I do.

As always, I welcome your comments about anything you see in this guide, any questions you have, or if you just want to get my thoughts on a wide variety of topics related to programming. And baseball too.

This guide was built with the help of the awesome folks at Leanpub[1]. My initial decision to do the guide as text with some AsciiDoc[2] markup turned out to make it easier to convert things into Markdown[3] as Leanpub required. I also used Pandoc[4] to tweak some of the content in the three bonus essays I have included in this guide.

A big "thank you!" to Elizabeth Gauthier for the awesome cover. My wife commented on what an incredible likeness it was.

Unfortunately, this guide is not aimed at the beginning programmer. Writing testable applications is a hard thing to do and requires that you already have a solid background in programming. This guide is targeted at intermediate-level programmers who have had some exposure to writing tests and want to push their skills to the next level.

[1] http://leanpub.com

[2] http://www.methods.co.nz/asciidoc/

[3] https://en.wikipedia.org/wiki/Markdown

[4] http://johnmacfarlane.net/pandoc/

If you are looking for an introductory guide to writing unit tests I highly recommend the PHPUnit documentation[5]. It will cover all the basics.

I can be reached via email at chartjes@littlehart.net and on Twitter at @chartjes. I blog on a regular basis at http://www.littlehart.net/atthekeyboard. Thanks for taking the time to check out the guide and I hope you find things that help you get a better handle on your ability to test your PHP applications.

[5] http://www.phpunit.de/manual/3.6/en/phpunit-book.html

3

Testing is Good, Testable Applications are Better

There is no denying that the PHP world has finally caught up with other programming communities in terms of its attitude towards testing. As a result, you can find a lot of really good resources either online or in book form from your favourite retailer.

Most of this information focuses on the actual writing of tests and strategies for the type of tests you need to write. What I found was missing from most of these resources was how to write applications that you could actually test. Sounds like something obvious, at least to me.

There is an open secret out there: some applications can never be tested with automated testing tools. They're built in such a manner that you couldn't wrap anything beyond tests that examine the HTML output from it. That's really a shame, because these sort of things don't happen on purpose. Like technical debt in your code, it builds up over time and at some point you end up with an application that might as well be a black box: you put inputs into it and you get output back, and have no easy way to determine how to fix things if it sends you back something you don't expect.

But make no mistake: building a non-trivial testable application is something that actually takes experience. A lot of experience. You will mess up. You will need to refactor things at a later point, and will often find that refactoring to be painful when contrasted with the seemingly-simple feature request that you are trying to implement.

I am a big believer in the power of automated testing to give you an application that is maintainable, extendible and performant. I don't always get to work on applications that have made a commitment to be testable. It really is a different way of looking at the entire process of testing. It's one thing to know how to write an assertion statement, or understand how to use mock objects to isolate components of a module that you are trying to test. It's another to have to create an application that lets you test it from end to end.

I am hoping to pass on my thoughts on building applications that you can easily test, and show you some of the techniques that I use to try and make the job easier. In this guide you will learn about creating the proper environment for your application, a programming concept known as Dependency Injection, and some best practices for testing the output of your application from the command line.

I really think that an optimal architecture for a web application is to build small modules of code that you then integrate together to solve larger problems. There is no other magic bullet to build a

complicated system. I have run across an old saying (that I have been unable to find an attribution for) that says "simple systems can display complex behaviour but complex systems can only display simple behaviour".

Keep your modules, components, libraries, whatever you decide to call them small, readable, and easy to verify that it works as desired in isolation or when integrated together. That way you will have an application that is very easy to write tests for and provide you with some protection and against regression errors and other unexpected bumps in the road.

The goal is always the same: create a workflow that shortens the time it takes for code to make it from your text editor of choice all the way into production with the fewest bugs possible.

4

Building Testable Applications is Hard

If you've been a PHP developer for any length of time you have probably encountered someone who talks about the benefits of testing your code. Test-Driven Development (TDD)[1], Behaviour-Driven Development (BDD)[2], Functional Testing[3] are just some strategies you can use. Not testing your code should not be an option.

I was once there too: I did not believe in the value of writing tests for my code. I am familiar with all the excuses people give:

- "Writing tests will slow me down."
- "Why can't we just manually test the application in the browser?"
- "I'd be done with this feature already if I didn't have to write the tests."

Really, all they are is excuses. Given the tools available to PHP developers there are no reasons to not write tests for your application. Even applications that resist your efforts to refactor individual components to wrap unit tests around them, the dreaded "untestable application", can be tested in an automated setting.

The reason for investing in automated testing is obvious: any bugs you catch before your application makes it into production cost less in terms of resources (money, developer time) to fix than fixing it in production. In a study that IBM and Microsoft conducted about TDD[4] they found some interesting information. A project using TDD would take 20-40% longer to complete than one that did not use TDD. However, the resulting application contained 40-90% fewer bugs that were discovered in production. If you tweak the statistics just right, how does 20% extra time for 90% fewer bugs sound? How could someone say no to that?

PHPUnit[5], the gold standard for testing PHP code, is easily integrated into pretty much any project. Install it via PEAR[6], create a directory in your application for your tests, and start writing them. However, sometimes it's not that simple. Not every application is already set up for testing.

[1] https://en.wikipedia.org/wiki/Test-driven_development
[2] https://en.wikipedia.org/wiki/Behaviour_driven_development
[3] https://en.wikipedia.org/wiki/Functional_testing
[4] http://www.infoq.com/news/2009/03/TDD-Improves-Quality
[5] https://github.com/sebastianbergmann/phpunit/
[6] http://pear.php.net

Do you use a 3rd party system for authenticating users? You're going to have to figure out how to simulate the responses from that system. You have hard-coded values for speaking to web services? You will have to extract those and put them into a configuration file. The reality is that some applications need refactoring before you can even test them.

In this guide I'm going to show you how you can build an application that is actually testable. Of course we are starting from scratch so it is easier but I will also show alternatives for refactoring code that might not be so test-friendly.

In each chapter we will cover a different topic that will increase your ability to create applications that you can create tests for. Some of the problems you might face are quite easy to solve, while others might require some intrusive changes to make it happen.

In order to illustrate a few of what I consider best practices for creating an application that you can easily test, I will be creating a sample application that utilizes many of the things I am suggesting. The code for the application can be found via my Github account[7]

Our companion application is based on a web site for a simulation baseball league. I've been in the Internet Baseball League[8] since 1997 and created this version of the site using a now very-outdated version of CakePHP[9]. It certainly gets the job done but I've been wanting to clean up stuff going under the hood and take advantage of some of PHP 5.3's features.

Rather than duplicate the whole application I've extracted parts of the application that generates the main page for the site. It's probably the most complicated public-facing part of the site. It incorporates the use of multiple models and fairly complicated use of templates. A great candidate to highlight the value of tests to verify functionality.

Even though I have acquired a reputation over the years as being Mister Framework, we're not going to use a framework here at all. I thought it was important to show you that although most of the major players in the PHP framework world make it easy to write tests for applications built with them, you can wrap just about any application in tests.

I'm using a combination of features and design patterns. While I'm not addicted to design patterns, I'm a believer in the right solution for the right problems. Let's take a look at what we're using.

- PHP 5.3.8 installed on OS-X 10.7.2 installed using a modified Homebrew recipe[10]
- The Xdebug[11] extension for code-coverage reports
- The APC[12] extension for boosted PHP performance
- Twig[13] for templating

[7] https://github.com/chartjes/building-testable-applications
[8] http://www.ibl.org
[9] http://www.cakephp.org
[10] http://notfornoone.com/2010/07/install-php53-homebrew-snow-leopard/
[11] http://www.xdebug.org
[12] http://www.php.net/apc
[13] http://twig.sensiolabs.org

- Postgres[14] as the data store
- Pimple[15], a dependency injection container that is used as a registry of objects and values needed by other parts of the application

I have also tested this application in an Ubuntu 10.04 (Lucid) virtual machine on my MacBook. More on the role of virtual machines in testing in a later chapter.

For the application architecture I decided to go with a Page Controller structure for each section of the application. If you've used a framework you've been using an application with a Front Controller structure. This means that every request goes through a single file and then the framework itself interprets the request and determine what controller and action pair needs to be executed.

A Data Mapper pattern as outlined in Jason Sweat's awesome book on Design Patterns[16] is used to provide access to the database. If you think you might have to change your data store, whether it's moving to a clustered solution or switching to one of the newer NoSQL options out there, a data mapper will help make that change easier.

In this guide I have only given you a glimpse at the tests that accompany our companion article. Please grab a copy of the code from Github and play around with the application and tests yourself. The code can be downloaded at https://github.com/chartjes/building-testable-applications[17] and I think having the source code will make a lot of the concepts outlined in this guide easier to understand.

[14] http://www.postgresql.org

[15] http://pimple.sensiolabs.com

[16] http://www.phparch.com/books/phparchitects-guide-to-php-design-patterns/

[17] https://github.com/chartjes/building-testable-applications

5

Environmental Consistency

When it really comes down to it, building your application in a way that it can be easily tested requires discipline and attention to detail. Lots of it. I tell you this because you have to do a lot of work before you've even written a line of code.

Undisciplined developers end up experiencing a large number of problems that are really fixable. If you've ever come across the phrase "well, it worked for me on my computer / laptop / slice / EC2 instance" then you will understand what I'm getting at. I've uttered those words from time to time when 100% convinced that my environment wasn't the problem.

What can be done to help out with this problem? If you can do only one thing that is not related to code to help make your applications easier to test (automated or otherwise), make sure that the environment you develop your application in is as similar to the environment where it will be deployed as possible. The only difference between your development environment and your production environment should be the data you are manipulating.

It is increasingly rare to come across an application that is more than just one PHP script that does not rely on other tools. Even look at the simplest programming stack available to us: LAMP, or Linux + Apache + MySQL + PHP. A project I am working on also includes MongoDB[1], Redis[2], APC, and Memcached[3]. I'm probably leaving out a few other minor components, but even that list shows how complex a relatively simple application can get. Whatever components you choose to use, it is vitally important that you make sure that every environment is using the same one.

The easiest way to accomplish this is to actually separate your development environment from whatever you are doing your development work on. This usually means you have two choices: use virtual machines locally or virtual machines remotely.

My personal preference is to use virtual machines locally. Most modern computers can run a 512MB to 1GB virtual machine image without trouble. In fact, you would be surprised how responsive a 512MB virtual machine is when doing development work, even when running a few extras like MySQL and even Memcached. To do work on a local virtual machine, I feel like you really have two choices: VMWare[4] or Virtualbox[5]

[1] http://mongodb.org

[2] http://redis.io

[3] http://memcached.org

[4] http://www.vmware.com/

[5] https://www.virtualbox.org/

The only real difference between VMWare and Virtualbox is that VMWare is not free but Virtualbox is. I prefer to use open source solutions so I go with Virtualbox. Use whatever tool you feel comfortable using, but they both accomplish the same task: letting you run a virtual machine containing the operating system of your choice on your own machine. For both choices there is an abundance of information on how to set things up so you can even locally edit code that is on the virtual machine.

If you go the Virtualbox route I recommend the use of Vagrant[6], a tool that uses Virtualbox to help you build and distribute virtualized environments. Once you have things configured just right (you can use tools like Chef[7] or Puppet[8]) you can create developer and production instances that can be shared between your team in order to develop and test your application. Why is this such a good approach? You can experiment with your application and simply trash it if you do things like accidentally delete the database or screw up a configuration file for one of the associated components your application relies on.

You don't have to use Chef and/or Puppet as Vagrant can just use a shell script as part of your provisioning process. Technical reviewer and physicist-turned-machine-language-programmer Joel Perras had this advice to offer me about Vagrant:

> - Use RVM to install the latest Ruby (1.9.3 as of right now), and then install Vagrant in its own gemset. This will save you a *lot* of trouble down the road. I once had to spend 1.5 days debugging a Vagrant install that would basically crash every time it loaded up, and it turned out to be an issue with the Ruby version that the developer was using which was different than the one that I had been testing it on. The irony of that wasn't lost on us.
> - You don't need to go crazy using Chef/Puppet for provisioning. Vagrant also supports a provisioning method that lets you point it to a shell script that will just be executed, so you can fill that simple text file with a few lines of `apt-get foo`, and that'll let you get pretty far without needing the brain twist that is puppet/chef.

Now, if you're not comfortable messing around with virtual machines on your own desktop, a great fallback option is to do your development work on a remote server. VPS accounts (Virtual Private Server) are great for this, along with using "cloud computing" services like Rackspace[9] or Amazon's AWS[10] to create remote environments that you can do your work in. My current employer has our development servers running on Amazon's EC2 service.

For those who are comfortable working from a CLI (Command Line Interface) doing your development work is as easy as connecting to the remote development instance using SSH and then editing code using your choice of editor designed for the CLI (Vim or Emacs, usually). For those not so

[6] http://vagrantup.com

[7] http://www.opscode.com/chef/

[8] http://projects.puppetlabs.com/projects/puppet/

[9] http://www.rackspace.com

[10] http://aws.amazon.com/

comfortable working like that, with a little work you can edit those files remotely if your GUI editor supports connections via SFTP or SSH. I have found that the performance hit from waiting to save those files remotely gets annoying real fast when you have to make all sorts of little changes. I did not find the experience of using TextMate[11] along with ExpanDrive[12] an enjoyable one.

My crankiness about network latency and how certain plugins for editors like Vim[13] and Textmate behave over a network connection aside, you should consider the advantages that the use of virtual machines for development work can give you. Distributing virtual machines configured for the needs of your application is the ultimate in consistency.

We have also witnessed the rise of services like Orchestra[14] and Zend PHP Cloud[15] for those looking for a managed, pre-configured environment to run their PHP applications in. I like these services because they actually help you in terms of consistency. Part of the reason is because they themselves have standardized on what version of PHP they will support, what version of MySQL they will support, and other components they offer.

The difference between something like this and services like EC2 or Rackspace is that you can't really run a version of it in your local development environment. While I really like things like Orchestra and PHP Cloud, you have to be careful to not deviate far from the versions being offered when setting up your development environment.

Whatever solution you choose, keep in mind the goal: a consistent environment for all locations that your application will exist in.

[11] http://macromates.com

[12] http://expandrive.com

[13] http://www.vim.org

[14] http://orchestra.io

[15] http://www.zend.com/en/products/php-cloud/

6

Static Code Analysis

Tools for doing static code analysis have come to PHP very late in its lifecycle. Tools like these have been far more common in languages like Java or C++: languages where it's easy to create complex code...and difficult to tame if it gets out of control.

Obviously it is easier to do static analysis of code that is statically typed and compiled. Proper static code analysis of dynamic languages is difficult and has been the subject of a few research papers. So treat them as tools that compliment what you are doing, not critical parts of the testing toolchain.

In this chapter we will examine several tools that I use every day when working with fairly complex code bases.

For a project where more than one person has to work on it (or integrate it into their own project) it's important to pick a coding standard. This covers things like tabs vs. spaces, brace location, variable naming conventions, things like that. Sure, there have been plenty of nerd fights over what set of coding standards are the "best", but what is important is to not only set a standard but FOLLOW it.

Have you ever worked with a programmer who coded in such a distinct style that you could instantly tell which code was his? Don't be that person. Follow the standard and save being an individual for your choice of code editor.

PHP Code Sniffer[1] is a CLI utility that will take your code and compare it to a particular standard. I have found it to be useful tool for making sure that your code stays organized and consistent.

Let's run it on a file I created for this project, a Data Mapper for talking to our Postgres database to get information about franchises in our simulation baseball league.

```
chartjes@yggdrasil [09:39:25] [~/Sites/local.ibl/lib/ibl] [master *]
-> % phpcs --standard=Zend FranchiseMapper.php

FILE: /Users/chartjes/Sites/local.ibl/lib/ibl/FranchiseMapper.php
FOUND 4 ERROR(S) AND 5 WARNING(S) AFFECTING 9 LINE(S)
   6 | ERROR   | Expected 0 spaces before opening brace; 1 found
   7 | ERROR   | Protected member variable "conn" must contain a leading
     |         | underscore
   8 | ERROR   | Protected member variable "map" must contain a leading
```

[1] https://github.com/squizlabs/PHP_CodeSniffer

```
     |         | underscore
  15 | WARNING | Line exceeds 80 characters; contains 86 characters
  77 | WARNING | Line exceeds 80 characters; contains 84 characters
 139 | WARNING | Line exceeds 80 characters; contains 90 characters
 142 | ERROR   | Line exceeds maximum limit of 120 characters; contains 181
     |         | characters
 152 | WARNING | Line exceeds 80 characters; contains 117 characters
 154 | WARNING | Line exceeds 80 characters; contains 86 characters

Time: 1 second, Memory: 5.50Mb
```

Ah, look at all the problems it found. Tsk, tsk, Mister Hartjes. That's some laziness in there. Complaints about line lengths (makes things less readable) and not following proper naming conventions for protected variables.

I will go and fix those right now but let's see what happens if we go with the default standard it uses.

```
chartjes@yggdrasil [09:39:37] [~/Sites/local.ibl/lib/ibl] [master *]
-> % phpcs FranchiseMapper.php

FILE: /Users/chartjes/Sites/local.ibl/lib/ibl/FranchiseMapper.php
FOUND 13 ERROR(S) AND 5 WARNING(S) AFFECTING 18 LINE(S)
   2 | ERROR   | Missing file doc comment
   5 | ERROR   | Missing class doc comment
   6 | ERROR   | Expected 0 spaces before opening brace; 1 found
  10 | ERROR   | Missing function doc comment
  15 | WARNING | Line exceeds 85 characters; contains 86 characters
  20 | ERROR   | You must use "/**" style comments for a function comment
  36 | ERROR   | Missing function doc comment
  53 | ERROR   | Missing function doc comment
  74 | ERROR   | Missing function doc comment
  95 | ERROR   | Missing function doc comment
 109 | ERROR   | Missing function doc comment
 127 | ERROR   | Missing function doc comment
 136 | ERROR   | Missing function doc comment
 139 | WARNING | Line exceeds 85 characters; contains 90 characters
 142 | WARNING | Line exceeds 85 characters; contains 181 characters
 149 | ERROR   | Missing function doc comment
 152 | WARNING | Line exceeds 85 characters; contains 117 characters
 154 | WARNING | Line exceeds 85 characters; contains 86 characters

Time: 0 seconds, Memory: 6.75Mb
```

That's a completely different set of problems. In this case, it is expecting that I would be putting DocBlock[2] comments around all my methods along with some warnings about line length.

It is an interesting exercise to analyze your code using all of the standards that it includes by default, just to see some other ideas on how code should look. The PHP Code Sniffer website also shows you how to create your own coding standards should you want to undertake such a task.

However, ensuring 100% compliance with what PHP Code Sniffer reports isn't always a practical goal to aim for. For example, it complains about my use of non-camel-cased, protected variable names in our model classes. They are protected to prevent direct access by developers and are non-camel-cased to match the values that are in the database tables themselves.

Altering the code to allow the model variables to be camelCased and prefixed with an underscore (as is the Zend convention) might also not be practical. The code that dynamically determines what model variable needs to be returned would also need to be altered.

Should I do the necessary work to make it happen? That depends. Do I have tests in place that I can run while I tweak things under the hood to make it happen? Of course I do! Showing you boring patches for the changes I made to the code will not suffice. Instead, let me show you the code itself and we'll discuss what had to be done.

I really only had to change code in one place: the BaseModel I am extending all my other models off of:

```
<?php

// Base data model that all our existing models will extend off of
namespace IBL;

class BaseModel
{
    protected $_id;

    public function setId($id)
    {
        // Never allow the ID for a model to be overwritten
        if (!$this->_id) {
            $this->_id = $id;
        }
    }

    public function __call($name, $args)
    {
        if (preg_match('/^(get|set)(\w+)/', $name, $match)) {
```

[2] https://en.wikipedia.org/wiki/PHPDoc

```
            $attribute = $this->validateAttribute($match[2]);

            if ($match[1] == 'get') {
                return $this->{$attribute};
            } else {
                $this->{$attribute} = $args[0];
            }
        } else {
            throw new \Exception(
                'Call to undefined ' .
                __CLASS__ .
                '::' .
                $name .
                '()'
            );
        }
    }

    protected function validateAttribute($name)
    {
        // Convert first character of the string to lowercase
        $field = '_' . $name;
        $field[1] = strtolower($field[1]);

        // We don't use __CLASS__ here because there are scope problems
        $currentClass = get_class($this);

        if (in_array($field, array_keys(get_class_vars($currentClass)))) {
            return $field;
        }

        return false;
    }
}
```

Before all this, I used to have an inflect() module that was using PHP 5.3 closures to take something like HomeTeamId and turn it into home_team_id, and then matching it against the available class variables for the model.

In order to make all the class variables for the class models conform to the standard, I no longer needed the inflector. Sure, it was a neat bit of code that used a closure as a callback, but the cleverness wasn't needed.

Also note how I commented on the section of code where I am using a shortcut to set the first

alphanumeric character of my attribute lower case. I needed this because the code takes something like setHomeTeamId and chops off the set part. In order to match it, I need to change it into _homeTeamId.

So now all my code meets the Zend coding standard. Onto other things like seeing how messy this pretty-looking code is.

Having updated my code until I got rid of all warnings having to do with the FranchiseMapper code being up to the Zend standard, the next tool to use is the PHP Mess Detector[3].

The job of this tool is to examine the complexity of your code and make suggestions on how to fix it. People create complex methods inside their code for the most innocent of reasons, but (beating a dead horse here) complexity is the enemy of maintainability and extendability. So, let's take a look at what it says about FranchiseMapper.

```
chartjes@yggdrasil [10:30:20] [~/Sites/local.ibl/lib/ibl] [master *]
-> % phpmd FranchiseMapper.php text codesize,design,naming,unusedcode

FranchiseMapper.php:5 This class has too many methods, consider
    refactoring it.
FranchiseMapper.php:137 Avoid variables with short names like $id
```

Aw, that's no fun at all. No real news to report there. But what if we use it on something a lot messier? Here's the same command run on a complex library for a work-related project I did.

```
> % phpmd lib/vendor/Moontoast/RpcApi.php text
    codesize,design,naming,unusedcode
RpcApi.php:12    The class RpcApi has an overall complexity of 77 which
    is very high. The configured complexity threshold is 50
high. The configured complexity threshold is 50.
RpcApi.php:12    The class RpcApi has a coupling between objects value
    of 14. Consider to reduce to number of dependencies under 13
RpcApi.php:27    Avoid unused parameters such as '$params'.
RpcApi.php:114   The method getProduct() has a Cyclomatic Complexity of 14.
RpcApi.php:114   The method getProduct() has an NPath complexity of 655.
RpcApi.php:114   Avoid really long methods.
RpcApi.php:114   Avoid variables with short names like $id
RpcApi.php:234   Avoid unused parameters such as '$params'.
RpcApi.php:282   The method getStore() has a Cyclomatic Complexity of 12.
RpcApi.php:282   Avoid really long methods.
RpcApi.php:302   Avoid unused local variables such as '$price'.
RpcApi.php:303   Avoid unused local variables such as '$previousPrice'.
```

[3] http://phpmd.org

```
RpcApi.php:398   Avoid unused parameters such as '$csc'.
RpcApi.php:477   The method saveCart() has a Cyclomatic Complexity of 26.
RpcApi.php:477   The method saveCart() has an NPath complexity of 5644802.
RpcApi.php:477   Avoid really long methods.
RpcApi.php:477   Avoid variables with short names like $id
RpcApi.php:588   Avoid excessively long variable names like
    $useBillingForShipping
RpcApi.php:608   Avoid excessively long variable names like
    $shippingAddressObject
RpcApi.php:671   Avoid unused local variables such as '$gateway'.
RpcApi.php:702   Avoid variables with short names like $q
RpcApi.php:707   Avoid variables with short names like $mm
```

You can see all kinds of places where you could make some improvements to your code. The PHP Mess Detector web site has good documentation on what all these messages mean, although some of them should be pretty self-explanatory.

My personal experience with PHP Mess Detector is that it does an awesome job of identifying places in my code where I'm trying to do too much in one method or have come up with an algorithm that might be too complicated for the task I'm asking it to do. It surprised me to see those complaints about unused parameters and local variables, as I am usually very diligent about not doing stuff like that.

Reducing the "mess" in your code will also go a long way to figuring out the best way to actually test it.

One of the biggest temptations when trying to "just get this done" is to slack off on your efforts to reuse code or extract functionality into its own methods (or even own classes) and just cut-and-paste code everywhere. PHP Copy Paster Detector[4] will statically examine your code and identify lines in your code where you appear to be repeating yourself, making it a candidate for refactoring that code into a method of it's own. Let's see what it says about our trusty FranchiseMapper

```
-> % phpcpd ./FranchiseMapper.php
phpcpd 1.3.2 by Sebastian Bergmann.

0.00% duplicated lines out of 197 total lines of code.
```

FLAWLESS VICTORY. Again, that's pretty boring. Let's take a look at another bit of code from a work project, code that deals with handling AMF calls from a Flash file used to generate store fronts for an ecommerce site:

```
-> % phpcpd lib/services/AMF_Checkout.php
phpcpd 1.3.3 by Sebastian Bergmann.
```

[4] https://github.com/sebastianbergmann/phpdcd

```
Found 2 exact clones with 52 duplicated lines in 1 files:

  - AMF_Checkout.php:60-68
    AMF_Checkout.php:576-584

  - AMF_Checkout.php:88-132
    AMF_Checkout.php:602-646

7.21% duplicated lines out of 721 total lines of code.

Time: 0 seconds, Memory: 6.50Mb
```

That's a little better. The first bit of analysis worried me, but 9 lines of duplicate code isn't that bad. Closer inspection revealed that it was the same set of 'if' statements being executed. I'll think about whether or not to extract that into a method of its own.

That second problem is worth examining. Almost 50 lines of code that's been duplicated. Definitely worth extracting into it's own method, to cut that down to just 50 lines of code. Less code means less bugs...usually.

I have found that the best way to use these tools is to compliment your efforts to have well-organized code. If you find you are having problems in generating code that is easily understood by others, these tools can be used to identify places where friction is happening in the code base.

7

Decoupling Your Objects Using Dependency Injection

Whenever I encounter a code base for the first time, my initial thought is about how I can test individual components as I modify parts of the application to meet new requirements or to fix bugs. In a perfect world, you will inherit an application with lots of relevant tests and great code coverage.

Most of the time you have an application that is working…and no way to test components in isolation. That doesn't mean the application sucks, it just means it will be difficult to make changes without the nagging feeling that you might have broken something else.

Most PHP developers are using object-oriented programming in their applications. You create objects to accomplish certain tasks, and if you went with a Model-View-Controller design pattern you are hopefully isolating your business logic in models, handling your routing requests in your controllers, and displaying output in nicely marked-up templates.

The problem with testing some of these things is the tendency for most developers to create tightly-coupled objects that perform tasks that often stray from their original purpose. So if you have a module of code that you need to test, the first thing to look at is decoupling things.

To guide you in your attempts to decouple your code and move it towards a scenario where it consists of well-crafted modules that talk to each other via rainbows and unicorn sightings, the first tool I offer you is the concept known as the Law of Demeter[1].

One of my all-time favourite programming books "The Pragmatic Programmer"[2] describes the Law of Demeter like this:

> The Law of Demeter for functions states that any method of an object should call only methods belonging to itself, any parameters that were passed in to the method, any objects it created, and any directly held component objects.

So what is really the point? The point is that by sticking to these rules as much as you can, you will reduce the friction when making changes to your code and also make it easier to test, since you will be able to use mock objects when really trying to test components in isolation.

[1] http://en.wikipedia.org/wiki/Law_of_Demeter

[2] http://pragprog.com/book/tpp/the-pragmatic-programmer

However, one of the downsides of really embracing the Law of Demeter is that you will find yourself writing a lot of "wrapper code", code that takes what was passed to it and then handing it off to some other function or object for processing.

How can we stay on the positive side of LoD? It's quite simple: we use Dependency Injection (hereafter DI) to pass in the various objects and values that our classes need to get the job done and don't force them to assume anything about the dependencies we given them except public interfaces to them.

The method I prefer to use is called Constructor Injection. Typically I will create a constructor for my object that accepts all the dependencies it will require. This is easily accomplished by creating a __construct() method for your class. I think some code examples are in order.

In our sample application, we using two types of objects to interact with our data source. A Model object that provides a representation of a single record in our database and a Data Mapper object that speaks to our data source and in turn creates Model objects.

Now, if I wasn't concerned about testability I would create one big object that performed all that functionality for me. I'd create a connection to my database in the constructor and then write methods that talk to the database and send me back the results I want. Makes sense, right? Not from a testing perspective.

If we apply the Law of Demeter here, by the rules our Model doesn't need to know anything about a Mapper except how to use it. That way we are free to do things like provide mock Mappers if we really want to get down and dirty about testing things.

So what if I want to test things out? If I've hard-coded the details and creation of my DB connection in the constructor for my Mapper, then I cannot run any tests on a database dedicated just to testing. Here's a test showing why just a sprinkling of DI lets you easily test your code.

```
<?php

include 'test_bootstrap.php';

class FranchiseModelTest extends \PHPUnit_Framework_TestCase
{
    protected $_conn;

    public function setUp()
    {
        $this->_conn = new PDO(
            'pgsql:host=localhost;dbname=ibl_stats',
            'stats',
            'st@ts=Fun'
        );
    }
```

```
    ...

    public function testSaveUpdatesDatabase()
    {
        $mapper = new IBL\FranchiseMapper($this->_conn);
        $franchise = new IBL\Franchise();
        $franchise->setId(25);
        $franchise->setNickname('TST');
        $franchise->setName('Test Team');
        $franchise->setConference('Conference');
        $franchise->setDivision('Division');
        $franchise->setIp(0);
        $mapper->save($franchise);

        // Update existing model
        $franchise->setIp(35);
        $mapper->save($franchise);

        // Reload Franchise record and compare them
        $franchise2 = $mapper->findById($franchise->getId());
        $this->assertEquals(35, $franchise2->getIp());

        // Clean up the franchise
        $mapper->delete($franchise);
    }
}
```

This makes sense, right? Our Mapper doesn't need to know the details of creating a connection to the data source. It just needs to know that it has been given a connection to use.

To further see this at work, you can even look at the save() method for FranchiseMapper.

```
<?php

namespace IBL;

class FranchiseMapper
{
    protected $_conn;
    protected $_map = array();

    ...
```

```
    public function save(\IBL\Franchise $franchise)
    {
        if ($this->findById($franchise->getId())) {
            $this->_update($franchise);
        } else {
            $this->_insert($franchise);
        }
    }

    protected function _insert(\IBL\Franchise $franchise)
    {
        try {
            $sql = "
                INSERT INTO franchises
                (nickname, name, conference, division, ip, id)
                VALUES(?, ?, ?, ?, ?, ?)";
            $sth = $this->_conn->prepare($sql);
            $binds = array(
                $franchise->getNickname(),
                $franchise->getName(),
                $franchise->getConference(),
                $franchise->getDivision(),
                $franchise->getIp(),
                $franchise->getId()
            );
            $sth->execute($binds);
        } catch(\PDOException $e) {
            echo "A database problem occurred: " . $e->getMessage();
        }
    }

    protected function _update(\IBL\Franchise $franchise)
    {
        try {
            $sql = "
                UPDATE franchises
                SET nickname = ?,
                name = ?,
                conference = ?,
                division = ?,
                ip = ?
```

```
                WHERE id = ?";
            $sth = $this->_conn->prepare($sql);
            $fields = array(
                'nickname',
                'name',
                'conference',
                'division',
                'ip',
                'id'
            );
            $binds = array();

            foreach ($fields as $fieldName) {
                $field = $this->_map[$fieldName];
                $getProp = (string)$field->accessor;
                $binds[] = $franchise->{$getProp}();
            }

            $sth->execute($binds);
        } catch(\PDOException $e) {
            echo "A database problem occurred: " . $e->getMessage();
        }
    }
}
```

Again, we're telling our methods that "hey, here's an item that you need to use" and it knows only how to use it, not what the internals look like.

Here's a scenario that might make more sense of how to refactor code with some help from LoD and DI.

Imagine you're creating your own framework and you are writing an Access Control Layer (ACL for short) component. For those not familiar with the idea, ACL's take a look at the resource you are trying to access and compares it to the user level you currently have, then tells you whether or not that user can access that resource. It might look something like this:

```
<?php

namespace Grumpy;

class Acl {
    protected $_acls;

    ...
```

```
    public function accessAllowed()
    {
        $request = \Grumpy\Context::getRequest();

        return ($acls[$request->getUri()] >= $_SESSION['user_level']);
    }

}

// Meanwhile inside your controller

$acl = new \Grumpy\Acl();

if (!$acl->accessAllowed()) {
    \Grumpy\View::render('access_denied.tmpl');
} else {
    \Grumpy\View::render('show_stolen_cc.tmpl');
}
```

We're pulling in our request object via a static method call, and we're getting user info right out of the session. Very common coding practices but not testable at all. Let's clean this up a bit.

```
<?php

namespace Grumpy;

class Acl {
    protected $_acls;
    protected $_request;
    protected $_userLevel;

    ...

    public function __construct($request, $userLevel)
    {
        ...

        $this->_request = $request;
        $this->_userLevel = $userLevel;
    }
```

```
    ...

    public function accessAllowed()
    {
        return ($this->_acls[$this->_request->getUri()]
            >= $_SESSION['user_level']);
    }

}

// Meanwhile inside your controller

$acl = new \Grumpy\Acl($this->request, $_SESSION['user_level']);

if (!$acl->accessAllowed()) {
    \Grumpy\View::render('access_denied.tmpl');
} else {
    \Grumpy\View::render('show_stolen_cc.tmpl');
}
```

Now that you've learned about how to use Constructor Injection to pass items into your objects, and you've also learned how proper use of interfaces in your code can let you mock objects for testing purposes, we can now explore cranking up the power level through the use of Dependency Injection Containers[3], hereafter referred to as a DIC.

At some point your application becomes large enough that you will find yourself having to manage not only a lot of objects, but complicated objects. You will also be asking yourself how to move these objects around for the purposes of dependency injection. The best way to handle this is to put these objects that you know you will need to move around in a container and then simply retrieve them when you need them.

So what sorts of applications need things like this? The prime candidate for getting the most use of a DIC is a web application framework. Why? Most frameworks are heavily object oriented and use a large number of objects to accomplish tasks. You can really make it easier for users of the framework if you place objects that they are likely to need to use frequently inside the container. They can be retrieved for later use, already configured and ready to go.

Consider our application with its Model and Mapper structure. We could use a container to store our database connection object instead of always creating the instance ourselves. Here, we will use Pimple as our DIC.

```
<?php
```

[3] http://fabien.potencier.org/article/12/do-you-need-a-dependency-injection-container

```
// Inside our bootstrap

// Create our container
$container = new \Pimple();

// Inject our DB connection object for later reuse
$container['db_connection'] = function ($c) {
    return new PDO(
        'pgsql:host=localhost;dbname=ibl_stats',
        'stats',
        'st@ts=Fun'
    );
};


// Then in a controller somewhere...
$mapper = new IBL\FranchiseMapper($container['db_connection']);
```

The upside to this sort of thing is now your objects REALLY don't need to know anything about the other objects or parameters they are using. If you really wanted to go completely nuts about this sort of stuff you could create instances of all your Mappers ahead of time in the bootstrap and then just pull them out of the container when you need them.

```
<?php

// In the bootstrap, after having already added
// in the database connection
$container['franchise_mapper'] = function ($c) {
    return new \IBL\FranchiseMapper($c['db_container']);
};
$container['game_mapper'] = function ($c) {
    return new \IBL\GameMapper($c['db_container']);
};
```

I'm not saying that this is a best practice but it illustrates that proper use of a DIC can go a long way towards making your application in line with the principles espoused in the Law of Demeter.

So, what else can we do with our dependency injection container now that we have it? Well, use it! In many ways you can also look at this as a "registry on steroids". I have typically used components for frameworks that act as registries of information. I store various values that I require persisted throughout the application and then retrieve them as required through calls to this registry or configuration object or whatever you want to call it.

Another reason to use a container for the dependencies is that it creates one place where you can find them. Without it, your code is littered with requests to instantiate those dependencies prior to injection. Again, imagine the scenario where you are switching out a data mapper from one that uses Postgres as the backend to one that uses MongoDB: how much pain would it be to go through all your code and replace all the instances where you create an old mapper and replace it with the new one? If you were using a container, you only change it in your bootstrap.

Being able to extract your dependencies when you need to inject them is helpful when trying to build your testable application. It saves you the trouble of instantiating them just before you use them, and if you use a container you can return an object that is already configured the way you need it.

It is also very easy to abuse the user of a DIC. Let's say that you have an object called Omega, that has three dependencies that you need to pass in:

```
<?php

class Omega
{
    protected $foo;
    protected $bar;
    protected $bizz;

    public function __construct($foo, $bar, $bizz)
    {
        $this->foo = $foo;
        $this->bar = $bar;
        $this->bizz = $bizz;
    }
}

$dep1 = new Foo();
$dep2 = new Bar();
$dep3 = new Bizz();
$newOmega = new Omega($dep1, $dep2, $dep3);
```

You decide to be clever and use a dependency injection container to store stuff in. Then you think "why don't I just pass in the container since the stuff is in there already?"

```
<?php

class Omega
{
```

```
    protected $foo;
    protected $bar;
    protected $bizz;

    public function __construct($container)
    {
        $this->foo = $container['foo'];
        $this->bar = $container['bar'];
        $this->bizz = $container['biz'];
    }
}

$container['foo'] = new Foo();
$container['bar'] = new Bar();
$container['bizz'] = new Bizz();
$newOmega = new Omega($container);
```

Now, let's suppose further that I am the one who is assigned to review this code as part of the normal development flow. I would take a look at this and immediately point out that you have now coupled your dependency injection container with Omega.

"What happens if you ever change the container? Then you have to go through all your existing code that uses the container and rework it!"

If you are going to use a dependency injection container (and I highly recommend doing so for an application of medium to large complexity) then treat it as a container only, and retrieve your dependencies from it before injecting them into your objects.

Dependency injection is a concept that I think every developer needs to understand even if they prefer not to use it. I also think that dependency injection is a critical part of creating objects that can be integrated with other objects with the lowest amount of friction.

You can increase decoupling in your code with some help from mock objects in your tests. The concept of mock objects is a very simple one: you create a representation of an object that is a dependency for your test and then create just the functionality for that object that your test requires in order to succeed. Here is an example that ties in the use of a mock object and ensuring that your code can run both on the web and the CLI.

Part of the reason of PHP's success in tackling and solving the problems of Web 1.0 was that it was built FOR the web. Contrast this with common "competitors" like Python and Ruby, who started out on the server and over time have built up libraries and frameworks that allow them to work just fine in web applications.

Given that PHP is so tightly coupled with creating web pages, it requires effort to avoid talking directly to all those awesome features built into the language that are web-specific. Cookies, sessions and $_SERVER immediately spring to mind. For PHP developers interacting with these things are second nature.

But this becomes a problem if you want to build an application that can be tested via CLI (Command Line Interface) tools like PHPUnit. You have to make sure that you are abstracting away any access to web-specific information or functions so that your application will actually work when modules of it are access from the CLI. Cookies come to mind and I'm sure that with more research you will find more examples of things that only live on the web.

Let's consider the example of your own ACL implementation that I showed you before. If you recall, it requires that you access the request object used in your application…which is tied to the web. Then it requires a value that you stored in the session…which is usually associated with the web but can be used in a CLI environment.

All hope is not lost. In our first quick little refactoring we used dependency injection to move grabbing the request object and the value from the session that we needed outside of the ACL object itself. How could you deal with this if your script is not running in the context of a web request?

```
<?php

// Now here is our test case

include 'test_bootstrap.php';

class AclTest extends \PHPUnit_Framework_TestCase
{
    ...

    public function testDashboardAcl()
    {
        // Create a mock object
        $testRequest = $this->getMock(
            '\Grumpy\Request',
            array('getUri')
        );

        // Create our test response
        $testRespect
            ->expects()
            ->method('getUri')
            ->with('/dashboard');

        $testUserLevel = 3;
        $testAcl = new \Grumpy\Acl($testRequest, $testUserLevel);
        $response = $testAcl->accessAllowed();
        $this->assertTrue($response);
    }
```

```
}
```

Again, it might seem like extra work just to make a test pass but it illustrates how straight-forward it really can be to ensure that your code is decoupled to the point where it doesn't even care what environment it is running in.

8

Like An Onion, Your Application Has Layers

I think that it is obvious that if you are going to be able to test your application with any sort of automated tool, you are going to have to split up your business logic from your display logic.

We've all built the application that is one big ball of mud. You know, two or three PHP scripts that mash together code that checks the value of $_GET, then branches off to do a MySQL query, then further loops through the result set outputting things. All in one place, but tremendously difficult when you need to update something. Which you, of course, do live on the server. I've been that guy. Don't be that guy any more. Life is too short and too many tools exist for you to use that excuse any more.

Most web application frameworks accomplish this task by encouraging the use of the Model-View-Controller software architectural pattern[1]. Why is this a good idea? Because it forces the decoupling of the different layers of your application. Unless, of course, the framework itself is not built properly. Hey, it happens.

So take your cue from these web application frameworks. Even if you are not going to use one of them because you are using your home-spun Precious Snowflake Framework (PSF) at least be smart and build it in such a way that you have your business logic in one place, your request-response logic in another, and your display logic somewhere else that you can work on in isolation.

Think of these things as layers for your application. To use a food analogy, your models, code that represents the data your application will use, is the bottom layer of your cake. Your controllers, which are called as the result of a specific request and then determines what the correct response should be, are the second layer of your cake, maybe with a little filling. Finally, your presentation layer is the frosting and delicious toppings for your cake. You want to be able to make changes to any of these layers at any time.

I can already hear you saying "But Chris, how often do you really change things like that?" More often than I would care to admit. Nobody makes the correct choice every single time when they build an application. Now, you might end up getting trapped within a particular framework because it requires a lot of discipline to only use the parts of a framework that you really need.

What you really want is an application that consists of well-crafted, well-tested reusable components that use a framework as simply the glue to get them to speak to each other. I've been involved in an

[1] http://en.wikipedia.org/wiki/Model-view-controller

open source project known as Phix[2] that is attempting to help build a set of tools to encourage this exact type of architecture for your application. But this is not a guide about Phix. We can use some of the philosophy behind Phix to help you.

If you're serious about providing useful tests for your application, you have to follow the mantra:

> If you can't automate a test for it, then it's broken.

Why do I say that? If you don't have a good set of tests then you really don't know if the application is going to behave in a way you expect. Of course there are many barriers to making this happen. Almost all of them are because the developer has chosen an incorrect solution for the problem. That might be a harsh thing to say, but it is usually the truth. I am not an exception to this rule either.

I think that testing data models is one area where many beginner to intermediate testers make mistakes. They fail to understand that the goal is to ensure that you are representing your data in a way that you expect and that you are also able to separate the generation of the data from the code that represents it.

I can be honest with you because you've read my guide this far: testing does take a lot of discipline and you often find yourself writing as much support and scaffolding code than code that actually works. Personally, I think that most of this work is very useful because it is a down payment on the future. All this extra work is an attempt to protect your code against unintended consequences. In other words, you want your tests to detect if you've broken something without knowing it.

Here's an example of what I call scaffolding code that you need in order to support the types of tests that provide value. In our test application, we have to generate standings. To do this we select a series of Games objects across our desired date range, then do a bunch of work to figure out how many wins and losses, and then return that list sorted by winning percentage.

Now, if I wanted to, how would I test this thing? The correct way to test it would probably involving creating a data fixture file that contains serialized objects that represent all the games we wish to generate standings from.

How do we create that fixture?

```
<?php
$conn = new PDO(
    'pgsql:host=localhost;dbname=ibl_stats',
    'stats',
    'st@ts=Fun'
);

echo "Collecting all games in our season...\n";
$gameMapper = new \IBL\GameMapper($conn);
```

[2] http://www.phix-projoect.org

```
$allGames = $gameMapper->findAll();
echo "Writing games objects into fixture file...\n";
file_put_contents('./fixtures/games.txt', serialize($allGames));
echo "Done\n";
```

Since we have the fixture, what's our test look like to make sure that our standings model is working the way we want it to work?

```
<?php

$testGames = unserialize(file_get_contents('./fixtures/games.txt'));
$conn = new \PDO(
    'pgsql:host=localhost;dbname=ibl_stats',
    'stats',
    'st@ts=Fun'
);
$testFranchiseMapper = new \IBL\FranchiseMapper($conn);
$testStandings = new \IBL\Standings($testGames, $testFranchiseMapper);
$testResults = $testStandings->generateBasic();
$this->assertTrue(count($testResults) > 0);
$testResult = $testResults['AC']['East'][0];
$this->assertEquals(1, $testResult['teamId'], 'Got expected team ID');
$this->assertEquals(97, $testResult['wins'], 'Got expected win total');
$this->assertEquals(65, $testResult['losses'], 'Got expected loss total');
$this->assertEquals('--', $testResult['gb'], 'Got expected GB total');
```

Okay, not bad for a first pass, but do you see the problem here? We are using a real Franchise mapper, instead of a mock one. That matters because we shouldn't have to actually talk to a database to get our information about what Franchises exist. That would be making the test dependant on our database, which we don't want.

What we should do is create a fixture that passes in all the Franchise objects that we will need, and then alter the Standings model itself internally to rely on those for any info that it wants.

Create a tool that generates the fixture data based on the data you have in your database.

```
<?php
include 'test_bootstrap.php';

$conn = new PDO(
    'pgsql:host=localhost;dbname=ibl_stats',
    'stats',
    'st@ts=Fun'
```

```
);
echo "Collecting all fixtures for our league...\n";
$mapper = new \IBL\FranchiseMapper($conn);
$allFranchises = $mapper->findAll();
echo "Writing franchise objects into fixture file...\n";
file_put_contents('./fixtures/franchises.txt', serialize($allFranchises));
echo "Done\n";
```

Now, we rewrite our test to use the new fixture and to test that our Standings model accepts an array of Games and array of Franchises as constructor arguments.

```
<?php
include './test_bootstrap.php';

class StandingsTest extends \PHPUnit_Framework_TestCase
{
    public function testGenerateRegular()
    {
        $data = file_get_contents('./fixtures/games.txt');
        $testGames = unserialize($data);
        $data = file_get_contents('./fixtures/franchises.txt');
        $testFranchises = unserialize($data);
        $testStandings = new \IBL\Standings($testGames, $testFranchises);

        // Rest of the test is the same
    }
}
```

I told you that I wasn't kidding that it requires a lot of extra code to support a test suite that does anything other than a very superficial set of tests. At the same time, the gyrations often required to provide this extra support gives you a very (and often brutal) look at the design of your code.

If you've noticed, I'm doing this using what people would normally consider fixtures when doing unit testing. The most common method is to use database fixtures and then pull in some sort of library that handles turning those fixtures into data. You then are able to use some sort of mock object that acts like a database connection as far as your tests are concerned.

There are times when you do need database fixtures. Maybe you cannot decouple things in the way I have been able to. If you want to go the route of using database fixtures then I recommend you take a look at Phactory[3]. It provides a simpler, code-driven way to create database fixtures and is an ideal fit if your application is using PDO for it's database needs

[3] http://phactory.org

Either way, you have to be aware of the drawbacks of using fixtures. When you change a test you will find yourself having to change the fixtures as well. It also slows down your tests when it has to load fixtures either the traditional way or my way of creating data objects. The slower your tests, the more likely developers will try and skip running the full test suite.

For this sample application, creating new data fixtures if my tests change is simple because I wrote small scripts to actually create them. Modify the fixture creation script, run it again, and I have a newly updated data fixture. It's certainly a lot easier than manually editing a large number of XML files if your tests require different data.

In other words: be smart about how much data you are importing for testing purposes. Chances are that you do not need a complete data set, just a small slice of data in order to validate functionality. If you can refactor your code to use objects instead of mocking out database connections, that's even better.

Here's another mantra you should take to heart:

> If it's not yours, wrap it up

If you are using an 3rd-party API to do something like user authentication you will quickly discover that it becomes very difficult to test it. A typical scenario might be where this 3rd-party API requires a redirect to their site where the user fills out some information and then they are returned to your application with some sort of token that says you are who you claim to be. Now this is good for you because it removes the burden of managing those credentials yourself, but bad because how do you test such a thing?

You might be lucky and the application you are using has a way to capture the request and response cycle. The sample application for this guide does not, so let's think of how you could use something like Facebook Connect.

The answer is, of course, that you place your use of a 3rd party authentication service inside a wrapper. Stay with me as I explain the logic.

Instead of thinking "how do I create a test where I talk to Facebook" you turn it on it's head and say "how do I create a test for an authentication service that uses Facebook Connect as it's data source?". I hope you can see the difference.

The normal way of using Facebook Connect is that you use their own SDK (or if you are a masochist you write your own OAuth[4] implementation) which basically redirects the user to Facebook, where they enter their login credentials for the site. If their credentials are good, they are redirected back to your application along with a token that you then can use to get information about the user.

In cases where I myself have used Facebook Connect for authentication purposes I tend to store information about the user in the session and then refer to it later. Here's a sample of what I mean:

[4] http://en.wikipedia.org/wiki/Oauth

```php
<?php
// Assume $facebookUser is the object representing info about our user

$_SESSION['email'] = $facebookUser->getEmail();
$_SESSION['name'] =
     $facebookUser->getFirstName() .
     ' ' .
     $facebookUser->getLastName();

// In other parts of the app when I need to check if the user is
// authenticated properly

if (isset($_SESSION['email'])) {
    // Execute the desired functionality
} else {
    // Redirect user to log in with Facebook Connect
}
```

As an aside, I tended to put the code that checks if a user is properly authenticated either in whatever structure bested suited the framework I was using for the project. Some frameworks make it easier than others to execute code in a controller before it runs your action-specific code.

However, if you look at my little sample above you can see why that would be so difficult to test: we're relying on having information stored in the session. How could we attempt to fix this?

The first thing we need to do is stop doing that direct check of the contents of the session and create an object that handles the task of telling us if we are authenticated or not.

```php
<?php
// New object that holds info about your Authenticated user

class AuthUser
{

    ...

    public function check()
    {
        return (isset($_SESSION['email']));
    }
}

// Then in the controller where you check authorization...
```

```
$authUser = new AuthUser();

if ($authUser->check() === true) {
    // Execute the desired functionality
} else {
    // Redirect user to log in with Facebook Connect
}
```

That's better, but we're still relying on a session being available. How can we reduce that particular dependency?

```
<?php

class AuthUser
{
    protected $_sessionObject;

    public function __construct($sessionContents)
    {
        $this->_sessionContents = $sessionContents;
    }

    public function check()
    {
        return (isset($this->_sessionContents['email']));
    }
}
```

Now we're making even more progress. Now let's imagine we are wanting to test some controller logic. In our sample application we are aiming to be able to load controller code independently of the environment we are in. This means that we have to make sure that do not depend on the existence of things like $_SESSION in this code.

However there are also some additional concerns. To run our test we also do mock object framework(https://github.com/padraic/mockery) not want to have to worry about actually connecting to Facebook in order to get info about the user. This is where the concept of Mock objects can be added to our toolkit.

Mock objects are created specifically for the purpose of making testing easy. At their most basic level you use them to replace an object that might need to communicate with the "outside world" such as a database or a 3rd party API. There are many different tools that can create mock objects for you (PHPUnit has built-in support for it[5]) but my personal preference is to use Pádraic Brady's excellent Mockery[6] mock object framework.

[5] http://www.phpunit.de/manual/3.6/en/phpunit-book.html#test-doubles

[6] https://github.com/padraic/mockery

In our hypothetical case we would want to create a mock object for our Facebook object. Here's how you might do it:

```
<?php
// Inside your unit test
$facebookMock = \Mockery::mock(
    '\FacebookAPI\Facebook',
    array(
        'getEmail' => 'test@littlehart.net',
        'getName' => 'Testy McTesterson'
    )
);

$sessionContents['email'] = $facebookMock->getEmail();
$auth = new AuthUser($sessionContents);
$this->assertTrue($auth->check());
```

Notice how effective the mock object is here. You get all the benefits of having an object that matches one that your code is dependant on without having to actually speak to Facebook in order to make a test. I believe that is a win-win situation when it comes to testing.

Testing your display logic via automated tests is something that often generates blank stares when I talk about it. "Why don't you just test that stuff in a browser." My usual answer is along the lines of "because I'm a human and humans make mistakes and skip things when they are testing a web page for the 100th time." Luckily it doesn't take much work to capture the output of your PHP application for testing purposes.

Here's a snippet from a controller in our sample application

```
<?php
include 'bootstrap.php';                                                    \


...

echo $twig->render(
    'index.html',
    array(
        'currentWeek' => $currentWeek,
        'currentResults' => $currentResults,
        'currentRotations' => $currentRotations,
        'currentSchedules' => $currentSchedules,
        'franchises' => $franchises,
```

```
        'rotationWeek' => $rotationWeek,
        'scheduleWeek' => $scheduleWeek,
        'standings' => $regularStandings,
    )
);
```

Using that same structure we could very easily write a test that looks something like this

```
<?php
include './test_bootstrap.php';

class MainPageViewTest extends \PHPUnit_Framework_TestCase
{
    protected $_twig;

    public function setUp()
    {
        // also include our libraries installed using Composer
        include APP_ROOT . 'vendor/.composer/autoload.php';

        // We are using Twig for templating
        $loader = new \Twig_Loader_Filesystem(APP_ROOT . 'templates');
        $this->_twig = new \Twig_Environment($loader);
        $this->_twig = new \Twig_Environment($loader, array('debug' => true\
));
        $this->_twig->addExtension(new \Twig_Extensions_Extension_Debug());
    }

    public function tearDown()
    {
        unset($this->_twig);
    }

    public function testMainPage()
    {
        $dbConn = new PDO(
            'pgsql:host=localhost;dbname=ibl_stats',
            'stats',
            'st@ts=Fun'
        );
        // Load data that we will need for the front page
        $gameMapper = new \IBL\GameMapper($dbConn);
```

```
$franchiseMapper = new \IBL\FranchiseMapper($dbConn);
$rotationMapper = new \IBL\RotationMapper($dbConn);
$scheduleMapper = new \IBL\ScheduleMapper($dbConn);

$games = unserialize(file_get_contents('./fixtures/games.txt'));
$franchises = unserialize(
    file_get_contents('./fixtures/franchises.txt')
);
$standings = new \IBL\Standings($games, $franchises);
$regularStandings = $standings->generateRegular();
$currentWeek = 27;
$currentGames = unserialize(
    file_get_contents('./fixtures/games-27.txt')
);
$currentResults = $gameMapper->generateResults(
    $currentGames,
    $franchises
);
$rotations = unserialize(
    file_get_contents('./fixtures/rotations-27.txt')
);
$currentRotations = $rotationMapper->generateRotations(
    $rotations,
    $franchises
);
$rawSchedules = unserialize(
    file_get_contents('./fixtures/raw-schedules-27.txt')
);
$franchiseMap = unserialize(
    file_get_contents('./fixtures/franchise-mappings.txt')
);
$currentSchedules = $scheduleMapper->generate(
    $rawSchedules,
    $franchiseMap
);

// Display the data
$response = $this->_twig->render(
    'index.html',
    array(
        'currentWeek' => $currentWeek,
        'currentResults' => $currentResults,
```

```
                'currentRotations' => $currentRotations,
                'currentSchedules' => $currentSchedules,
                'franchises' => $franchises,
                'rotationWeek' => $currentWeek,
                'scheduleWeek' => $currentWeek,
                'standings' => $regularStandings,
            )
        );
        $standingsHeader = "Standings through week 27";
        $resultsHeader = "Results for week 27";
        $rotationsHeader = "Rotations for Week 27";
        $scheduleHeader = "Schedule for Week 27";
        $rotation = "KC Greinke, CHN Lilly -2, CHN Wells -2";
        $this->assertTrue(stripos($response, $standingsHeader) !== false);
        $this->assertTrue(stripos($response, $resultsHeader) !== false);
        $this->assertTrue(stripos($response, $rotationsHeader) !== false);
        $this->assertTrue(stripos($response, $scheduleHeader) !== false);

        // Look for a known team abbreviation
        $this->assertTrue(stripos($response, "MAD") !== false);

        // Look for a specific rotation to appear
        $this->assertTrue(stripos($response, $rotation) !== false);
    }
}
```

So what is the best way to look for the data inside our HTML output? You have two options in my opinion.

If you're looking for a specific string to appear in the HTML output, like say a message that you successfully created a new widget, then you are probably better off using PHP's built-in string searching functions.

```
<?php

$standingsHeader = "Standings through week 27";
$resultsHeader = "Results for week 27";
$rotationsHeader = "Rotations for Week 27";
$scheduleHeader = "Schedule for Week 27";
$rotation = "KC Greinke, CHN Lilly -2, CHN Wells -2";
$this->assertTrue(stripos($response, $standingsHeader) !== false);
$this->assertTrue(stripos($response, $resultsHeader) !== false);
$this->assertTrue(stripos($response, $rotationsHeader) !== false);
```

```
$this->assertTrue(stripos($response, $scheduleHeader) !== false);

// Look for a known team abbreviation
$this->assertTrue(stripos($response, "MAD") !== false);

// Look for a specific rotation to appear
$this->assertTrue(stripos($response, $rotation) !== false);
```

If you are trying to make sure that your output is formatted in a specific way, you should probably be looking to use a tool like PHP's bulit-in SimpleXML functions to load your output, create XML out of it (after all, HTML is really a subset of XML) and then use the xpath() method. Learning XPath[7] is not the easiest thing to do. I did a lot of work with it when I used to work for a sports data integration company that provided XML data feeds. But if you want to write a test to make sure that your error messages are being correctly wrapped in the proper CSS class, well, there really isn't any other way to make it happen.

So if you wanted to look for, say, a specific tag in the output being generated, I think the test would look a little like this with a little help from a third-party library called Zend_Dom_Query[8].

```
<?php

// $response contains the output of our test page...

$domQuery = new \Zend_Dom_Query($response);

// Find the div for our standings display
$node = $domQuery->queryXpath('//div[@id="standings"]');
$this->assertTrue($node != "");
```

I think the important thing here to remember is that our goal for creating building a testable view is that we need to be able to capture the HTML output before it is sent to the browser. That way we can search that output for strings (or DOM elements if you prefer that method) to match our expectations.

[7] http://en.wikipedia.org/wiki/Xpath

[8] http://framework.zend.com/manual/en/zend.dom.query.html

9

Shells and Sandboxes

This content orginally appeared on my blog in 2009. It covers a tool that is used constantly in languages like Python and Ruby but is pretty much ignored in PHP. Hope you enjoy it.

Now that I have become a born-again tester, I've started looking at the testing I'm doing at work from more of a high-level point of view. What are tools like SimpleTest[1] and PHPUnit[2] *really* doing? They are providing a mechanism for the automating of running of tests. Such tools also exist for frameworks like Django[3] and Ruby on Rails[4]. I cannot speak for Django, but the Rails community has made a very large commitment to testing tools and testing best practices. But there is also another very powerful testing tool that Ruby and Python make available that PHP does not.

Both Ruby and Python offer a Read-eval-print loop[5] that can be run from the command line to allow people to do the opposite of what conventional testing tools do: allow you to run interactive tests of your code. I used the REPL available in Python (you know, just typing 'python' from my command line) to test out how to use various geocoding libraries for a side project. My favourite Ruby programmer[6] uses irb[7] all the time to work on stuff.

So in a way, a REPL is the anti-automated test. I think that if there was a good REPL available for PHP, testing best-practices might take better hold. Perhaps there is hope for a good PHP REPL in Alan Pinstein's iphp[8] project. I remember trying to continue work on the testing console I had created a number of years ago for CakePHP, and quickly realized that what I needed was a REPL.

Since I did not have the desire to write my own limted PHP parser by messing around with eval[9] (although, to be honest I did try playing with it and realized I did not know what the hell I was doing), the testing console was dead. Besides, how many PHP devs really use PHP on the command line? 10%? 1%? Hard to know, as the group I follow is a self-selecting one and I'm sure most of them have used PHP on the command line. Hard to become an advanced PHP developer without trying to do stuff on the command line with it.

[1] http://www.simpletest.org/
[2] http://www.phpunit.de/
[3] http://www.djangoproject.com
[4] http://www.rubyonrails.org
[5] http://en.wikipedia.org/wiki/REPL
[6] http://gilesbowkett.blogspot.com/
[7] http://en.wikipedia.org/wiki/Interactive_Ruby_Shell
[8] http://github.com/apinstein/iphp
[9] http://www.php.net/eval

I was also pleased to find out (after a year of using it, no less) that eXist[10] had it's own REPL for XQuery - they call it the "sandbox". It's awesome because I can take my "wrote this 6 months ago and can't remember if it still works" XQuery scripts and test them out. Sort of like a command line client to your RDBMS. It even tells me when I've got syntax or formatting errors in them BEFORE I try and execute them. Talk about saving a developer time.

Do things like this exist for RDBMs like MySQL and Postgres? Imagine an interactive client that is checking your syntax out *while you type it.* If such a thing exists, please tell me!

Once again, I see I have started to diverge. Back on track again.

So, from a testing perspective it's easy to see how shells and sandboxes can become essential tools. For example, all the good advice I have ever seen about unit testing clearly advocates a sandbox approach. You create fixtures containing data. You create mock objects to simulate functionality that might potentially be destructive. You might even create a separate database for doing test operations on. Clearly, the goal here is to create an environment that is the same as the one your code will eventually be running in, but separate from the "real" one.

A REPL, on the other hand, is a very powerful and very dangerous. If you're not careful, you could very easily insert data into your current "working" environment instead of a "test" one. As with so many other things, power comes with a price. Sure, I can run some code to figure out how to use, say, SQLAlchemy[11] but I better make sure that I am not writing data to my production database. That would be silly.

I'm guessing the main reason PHP does not have a easy-to-use REPL is because it grew up on the web first and the command line second. Ruby and Python started on the command line first, and out to the web second. It's really that simple I think. Will PHP ever get a widely-accepted REPL?

[10] http://exist-db.org/

[11] http://www.sqlalchemy.org/

10

Rebuilding Your Programming and Deployment Environments

Some of this content originally appeared on my blog in 2009 and it goes into some more detail on the concepts behind being able to quickly and easily rebuild your environments. Hope you find it useful.

I've been paying close attention to what sort of things people have been doing in terms of best practices and actual application deployment. Looking at some of these things, I realize how lax I've gotten lately. Is it the crushing ennui of ongoing, incremental programming by a lone developer for his employer? I'm being sarcastic (obviously) but the more I think about HOW I've been doing the development work the more I realize that I need to refocus.

So, my programming environment appears somewhat sane to me. I write my code using the One True Editor(tm)[1] on my laptop, where I've installed PHP, Python and Perl. Since I work with both MySQL and Postgres, they are both on my laptop as well. Is it any wonder I've crammed my MacBook with 4 GB of RAM? I typically have a few Terminal.app instances running and a couple of MacVim[2] instances. I find that I use Sequel Pro[3] and pgAdmin III[4] a lot, as all that work with sports data requires large queries that I have to tweak little by little.

I also have found a powerful tool that I always ignored, but has been a godsend for development work. What do you do when you have a large data set that your application needs to work against and it is totally impractical to copy it locally? SSH tunneling to the rescue!

```
ssh -fNg -L3308:localhost:3306 notchris@server.domain.com
```

That creates a tunnel to the MySQL server on one of our production machines (we name them after NHL teams), which I then can access by making a MySQL connection on port 3308. Very simple and VERY effective. I can't believe I had forgotten about this. This technique works well when using something like Sequel Pro, and we had an SQL consultant in who connected the same way.

See, I'm having problems deciding WHAT to test. Let's take a look at one project for example. We have self-hosted web service[5] that we released under the GPL that we not only use ourselves for

[1] http://www.vim.org

[2] http://code.google.com/p/mavcim

[3] http://www.sequelpro.com/

[4] http://www.pgadmin.org/

[5] http://www.sportsdb.org/sd/sportscaster

some clients but also use it to troubleshoot things as well. In terms of writing tests, I believe the best way to approach this is to write tests that speak directly to the Sportscaster "engine" using a known set of parameters and ensure that the output we get back matches what we expect. Is unit testing overkill for this? Very hard to say because we have an HTML front end that talks to a "gateway" that talks to an "engine". I'm not the one who architected this, but I do have to make sure that it works properly. In this case I believe that "your tests should verify your application is outputting what you expect it to" is the best approach. Comments about this are totally welcome.

For another project, I tried something that integrated Selenium with PHPUnit[6], but it was so slow as to be totally useless. Sportso[7] is a very visual application, one that is written in PHP an talks to an XML database[8], but presentation is really the key here. Sure, we need to make sure that we're placing the right data into our XML documents but that part of the application is solid. In this case, I wish to have tests that simulate a user clicking around the application and using it. Then I can compare output again and make sure that things are showing up when and where I expect them to. I'm thinking the PHPUnit-Selenium bridge is a bad idea, and that I should instead just focus on Selenium. Again, comments on how to best approach this are also welcome. I cannot believe that Selenium is really this slow, so there must be something I am doing wrong.

Also, I've thought about continuous integration. This incredibly inspiring post about continuous deployment[9] has me really thinking hard about how my changes actually get pushed up out of dev to production. We don't have the resources to use a cluster of machines that IMVU appears to have, but when you think of the infrastructure and (more importantly) the practices in place, you just have to be in awe of it all. Just the *scale* of it is inspiring. If they can do this, I can certainly implement a very effective subset of this. There is really no reason other than laziness on my part to not do it. Harsh reality, to be sure.

But the real problem is the political game that you inevitably end up fighting over bring in what is usually a very radical concept: a commitment to testing as part of the development process. Especially when the developers are severely outnumbered. A lot of people seem to have a pathological fear of automated testing, saying things like "we can just quickly test it by hand". Then when you see the errors being reported by users of your products, you can't help but wonder "could we have caught this earlier?" Unless it's a very obscure error involving a race condition or something, of course the answer is YES.

I read a post from one of the main developers at justin.tv talking about why he felt unit testing is useless[10]. While he is entitled to his opinion I can't help but think that he really is doing TDD, but he just refuses to acknowledge it. Yes, ultimately, your users are the only ones who can tell you if your application is spitting out errors. But if you aren't doing any rigorous testing of stuff BEFORE your users see it, I think you are taking chances with your business. My company is small (just 7 of us) so I'm so close to most aspects of the business I can really see the impact of certain decisions on the bottom line of the company. It's not about finding bugs, which is what is being claimed by the

[6] http://sebastian-bergmann.de/archives/631-Integrating-PHPUnit-with-Selenium.html

[7] http://www.sportso.com

[8] http://exist.sourceforge.net

[9] http://timothyfitz.wordpress.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/

[10] http://abstractstuff.livejournal.com/60388.html

justin.tv folks never happens with unit testing. It's about making sure those bugs never happen in the first place.

I have to say that I've never really delved into the world of post-deployment monitoring, so maybe I should start asking a few more questions of people who deal with that sort of thing. Again, the key is to make it automated and have it report when it finds things that are wrong. We have a very good nagios[11] infrastructure at work that alerts us to problems with not only servers being down, but stale and/or missing data for our customers. I guess the next logical step is figuring out ways to look at those mountain of server logs and making sense of the errors? Definitely something to think about the next time I get stuck on some programming problem at work and need a change of pace.

I know I've been rambling a bit here, so let me pull some of this stuff back together to make it more cohesive:

- SSH tunneling to access resources that you cannot easily put a snapshot of into your work environment. I recommend read-only access if you can do it, because if you need to do updates or deletes you should always work off a local copy
- TDD is a powerful yet still controversial tool that too many developers are still ignoring (including me, but it's not too late to save myself!)
- Automation is the key to really taking advantage of what you gain by going with TDD
- You need tools in place after deployment to see if your changes have had the desired effect

[11] http://en.wikipedia.org/wiki/Nagios

11

Continuous Integration Landscape for PHP Developers

This content originally appeared on my blog in 2010 and has been updated due to some changes in the tools themselves. I hope you find the information here helpful.

For those not familiar with the concept of Continuous Integration[1], I can sum it up with one very glib phrase: it lets you break your code before you buy it. By this I mean, when combined with other programming practices like automated deployment and Test Driven Development, you get a chance to see if the code you've written *today* breaks anything you wrote *yesterday*.

If I were to design a Continuous Integration (hereafter referred to as CI) system, what would it look like? Setting aside the one I use already (details on that later), I believe there are some key features it requires:

- Ability to integrate with your version control system
- Web interface for cross-platform access to results of build
- Ability to run your unit test suite and view the results
- Ability to deploy code wherever you want
- Ability to email results of test builds
- Must be able to automate all of this for you

This is what I would call a non-trivial set of requirements.

It also illustrates what became the biggest realization for me: you cannot have CI without writing tests (whether they are unit tests or integration tests), and you cannot have CI without the process of creating a "build" being triggered without any further action by the developer. Without a commitment to writing meaningful tests it definitely won't work. It is my firm feeling that there *is* such a thing as pointless tests, and to me getting to 100% code coverage is often more about programmer ego.

[1] http://en.wikipedia.org/wiki/Continuous_integration

So, there are a few choices for PHP developers. You could use phpUnderControl[2], which is a plugin for CruiseControl[3], a Java-based tool. I took a look at it, saw all those XML configuration files and said "no thanks." If you like screwing around with XML configuration files, then maybe this is the right tool for you. Me, I like my pointy-clicky interfaces. Or plain-text configuration files. Also targeted mostly at Java developers. My project was using PHP *and* Python, and needed to be able to support Java as well for a non-web project another developer was working on.

If you prefer the cool ideology of a PHP-powered solution, you could look at Xinc[4]. Installs via PEAR (which is always a good idea), but drags a few other PHP components in with it. I dunno, doing it PHP-only held no special appeal to me.

Some organizations are also using Buildbot[5] but search results are full of people discussing how difficult it is to use for more inexperienced programmers. It is written in Python so if you have the right skills you can modify it to bend it to your will.

In the end, I settled on Jenkins[6]. Much like CruiseControl, it's Java based, but that was no big deal as we already have Java in our infrastructure at work with more little bits being moved to it. Also, Jenkins has the type of architecture I like: a small, tight core with lots of plugins to add the functionality I wanted. Using Subversion? Add the plugin. Need to run py.test[7] scripts? There's a plugin for that. Want to be able to see the results of your PHPUnit tests? There's a plugin for that.

So, not only that, but I could also write a shell script that is run when a build is triggered. In this case, I was using a post-commit trigger to start builds, but if I wanted to be dogmatic about it I would use a pre-commit trigger, and fail the commit attempt if the tests broke. I remember how the younger, angrier me flew into a rage at a previous employer when a hook was put into place in our CVS repository (yes, I've been around long enough to have actually used CVS) that ran a syntax checker on any code that got checked in. Predictably, something was broken because the version of PHP running on the CVS server was different from what we were using both in development and production, and a difference in extensions caused a problem. That is a story for another time.

I also find that using a post-commit hook allows me to look at the code in the staging environment to find bugs that are caused by subtle differences between the development environment and staging. Today, I fixed one bug by using the new ternary shortcut available in PHP 5.3. Learn something new every day, is my motto.

So, how does a build actually work? Here's how I've set things up:

1. A developer commits code to the trunk of the SVN repository. We don't currently use branches, so I'd have to tweak things if we did work in branches and then merged things into trunk.

2. Jenkins wakes up when it sees a commit, and checks the code out into a workspace.

[2] http://phpundercontrol.org/

[3] http://cruisecontrol.sourceforge.net/

[4] http://code.google.com/p/xinc/

[5] http://en.wikipedia.org/wiki/BuildBot

[6] http://jenkins-ci.org

[7] http://codespeak.net/py/dist/test/

3. Jenkins then runs a list of shell script commands to do a bunch of things: run my Doctrine migration scripts, run my PHPUnit tests, run my py.test tests, then copy the code into place on the staging server.
4. Create a bunch of test reports that can be viewed in Jenkins itself
5. Send out an email to our tech mailing list telling people that a build has been triggered along with the results

I have to say there is nothing that puts a smug feeling of satisfaction in my heart like seeing that a build was successful…and nothing that makes me curse quite as much as seeing that a build failed. But without those tests in place, then I would not find bugs that would've gone into production. Any bug making it into production is BAD, and although I often use gallows humour at work to try and deflect my internal anger at such a thing, it IS possible to production pushes of new code that contain no errors.

As a follow-up to this idea, I now have a policy that whenever a bug does get found in staging or production, I write a test to recreate the bug, and then put code into place to fix it. I have found that this has the interesting effect of giving you a bunch of unit and integration tests (thank you for the ability to test the output of controllers in Zend_Framework PHPUnit tests) that cover edge cases found by real users. Besides, you can get really lost going deep into the "we need to create fixtures so I can test stuff that manipulates dynamically-generated content" rabbit hole. I saw a test today that could benefit from the use of fixtures, instead of simply assuming that data will always be there. Your mileage will vary, but you have to understand that slavish devotion to testing does you no good if you're not accomplishing the goal of producing a working application that meets the goals of the people who are actually, you know, using it.

I hope this post helps you if you're trying to decide whether or not to take the CI plunge. I have no regrets at the length of time it took me to get things to a happy place in terms of Jenkins's configuration. The benefits of automated builds with the goal of protecting you from yourself more than outweighs a few afternoons of scratching your head and trying to figure out what is going on with Jenkins itself.

Infrastructure Debt

This content was originally a blog post from November 2011 that got some traction and ended up on Hacker News. It is directly related to the concepts that go into buliding testable applications. Hope you enjoy it.

You might have come across the term "technical debt", used to describe the small mistakes that are made in your code base as the application grows and requirements change. You end up with a big tangled mess if you are not very careful about how you make those changes. If you've ever worked on an application where you were afraid to make changes for fear of breaking something, then you have run into technical debt.

It's very easy to build up technical debt: you put a quick hack in because you feel like you are under pressure to get a task accomplished. It takes courage to push back and say "no, damnit, I need time

to fix this particular problem correctly!". I understand, and it's okay. We can hug it out if it will make you feel better.

Technical debt can be dealt with via ruthless refactoring and wrapping your application in tests that poke and prod at the edge cases that your application deals with. But there is another type of debt you end up dealing with. One that is even more difficult to deal with and change. I call it infrastructure debt.

Infrastructure debt is debt that you build up because you have not been paying attention to the process of creating the environments your application will exist in and have not been paying attention to the process of how your code gets from development into production. In my opinion, infrastructure debt is much more difficult to "pay off" than technical debt. Why? It is often very difficult for people to understand that it even exists.

Before I go any further, if you are not using a version control system to keep track of changes to your code, then please stop reading and go and install one. Having versions of your files with extensions like .1 and .old is the worst kind of infrastructure debt that you could possibly have. It's the 2nd decade of the 21st century. Version control is an almost 40-years-old, well-understood concept. Nothing I suggest will work if you insist on acting like version control is of no use to you. Passing around diff patches is not version control, my friends. It's denial. Don't tell me just because people used to do it in the past that it's okay now given all the available options. People used to think that the earth was flat once too.

Let's start with an easy infrastructure debt to pay off: inconsistency in development environments. These days it is pretty easy for someone to create a development environment for their language of choice along with associated components like databases. The problem? That it's pretty easy for someone to create the development environment of their choice.

You end up with your developers all having environments that are slightly different due to choice of source for packages or preferences on versions of components. I have PHP 5.3.8 via a custom Homebrew recipe while one of my co-workers has the stock version of PHP that comes with OS-X Snow Leopard. He has a version of MySQL that is two versions ahead of mine. See how easy it is to get out of sync?

You have two options. The first one is the draconian one, where you force everyone to use the same operating system and the same tools to minimize the chances of conflicts between development environments. Given how that leads to developer unhappiness, I prefer a more sane approach: use tools like VMWare[8] or VirtualBox[9] to let your developers do their work on identical virtual machines while giving them the freedom to use the development and debugging tools of their own choosing.

Also, by using a virtual machine you can encourage developers to push the envelope and test destructive things like library upgrades or database changes. Messed it up? Just fire up a new virtual machine and try again. When you combine it with good version control practices you should be able to recover from your mistakes even faster.

[8] http://www.vmware.com

[9] http://www.virtualbox.org

Just make sure to take the time to create virtual machines that are as close to an exact match as your production systems as you can. Honestly, the only difference between your development environment and your production environment should be the data your application is manipulating. It's not really that hard to even create a cluster of virtual machines on your own computer to simulate an architecture that has application servers, a database server and a memcached server.

It also helps you to get new developers up to speed quickly. "Oh, working on Project Alpha? I'll email you the link on the wiki where you can download the virtual machine for that project."

Check out Vagrant[10] for a great option for developers wishing to do their work in a virtual machine.

So, having injected some structure into your development environment there is really one last frontier where infrastructure debt builds up like the credit card bills of a shopaholic: deployment processes.

I really think there is one rule: if you cannot do your deployments with one command then you are DOING IT WRONG. If you can type the commands you are doing into a shell then you can script them so you don't have to type them in again. If you can script deployment, then you can automate deployment. If you can automate deployment then you now have a consistent and repeatable process that will behave the same way every single time you deploy.

At Moontoast[11] we have automated deployment in place. When we merge code into our staging and production branches, a version control hook triggers our deployment tool. It checks out our code into it's proper location on the production server. It runs database migrations that need to be done. It starts and restarts various services that are used by the application. In short, it does everything that I see far too many of my colleagues doing manually. Stop that. Clean up your infrastructure debt by making the process of deploying your code a non-event.

If you want to automate your deployment process, examine tools like Capistrano[12] or Phing[13] or you could even use a tool like Jenkins[14] and use it's concept of Builds to automate the process of taking your latest code changes into production.

Note that I said the PROCESS, because technical debt might mean that your application is fragile enough (doesn't have enough tests and/or test coverage) that the changes you made might cause it to behave improperly or even crash the system altogether. Not a good feeling, so by investing the time into automating your deployment process you can eliminate at least one headache.

I think the main point I am trying to get across with respect to infrastructure debt is that you really need to examine your processes beyond just coding. If you find yourself constantly fighting the "works on my computer" battle, you likely have serious infrastructure debt. If you find yourself constantly referring to a checklist whenever you do a deployment, you likely have serious infrastructure debt.

[10] http://vagrantup.com

[11] http://www.moontoast.com

[12] https://github.com/capistrano/capistrano/wiki

[13] http://www.phing.info/trac/

[14] http://jenkins-ci.org

Whenever I bring this topic up, I hear many of the same excuses that I hear when I advocate a commitment to writing tests and automating the execution of those tests. What is it that people have against this sort of thing?

I have found that people terribly underestimate the amount of manual work they do to accomplish a task. For many, it is a fear of losing control that leads them to conclude that automated deployment cannot be trusted, or that developers shouldn't have the freedom to pick the tools that they use to build their applications.

Technical debt is real, make no mistake. But so is infrastructure debt. Just like the cost of finding bugs in production is more than finding them in development, the cost of fixing infrastructure debt gets more and more expensive. Just think of the large number of problems solved by eliminating inconsistencies between development environments. Add to that the ability to eliminate user error during manual deployments of your application. Why would you not want to have this?

The decision to start paying down your infrastructure debt is not an easy one, because it will often require many changes to be made by many people. Most programmers are creatures of habit and dislike change, even when it is obviously better. Don't be scared of change, be scared of the debt growing in your code base and in your infrastructure. It won't go away and there is no government bailout on the way to fix it.

Fear is the mind-killer

This content orginally appeared on my blog in March 2011. It discusses how fear can prevent you from doing the necessary work to make your applications easy to test. Hope you enjoy it

Dune is probably my favourite science-fiction novel of all time. It's in a cool settings, with some really well-designed characters, and is a sprawling epic in the way that sprawling epics should be. In that book, the main character Paul Atreides learns the ways of the mysterious Bene Gesserit through his mother. One of the things he learns is called the *litany against fear*, something they say to themselves whenever they are in danger:

> *I must not fear.* Fear is the mind-killer. Fear is the little-death that brings total obliteration. I will face my fear. I will permit it to pass over me and through me. And when it has gone past I will turn the inner eye to see its path. Where the fear has gone there will be nothing. Only I will remain.

Living on the bleeding edge of technology means you spend a lot of time trying to convince the people around you that this awesome bit of technology that you've noticed is worth investigating. I remember being mocked and dismissed over my thoughts on memcached[15] in 2003. And being mocked and dismissed on my thoughts on how valuable writing automated tests could be for an out-of-control code base in 2004. It's now 2011 and both of those things are considered integral parts of best practices for web applications.

Some of the problems I faced were due to an abrasive personality very early in my programming career (although some would argue that not much has changed) that resulted in ideas being rejected

[15] http://www.memcached.org/

because of who the messenger is. Let's be honest, we've all done that: dismissed information because of who was delivering it. I've gotten used to facing the uphill battle to get my particular set of programming practices accepted but it is a battle I'm winning.

But there is often another reaction to new technologies. That reaction is fear. Fear of working with a set of technologies they are not comfortable with. Fear of automated systems deploying code that might not be adequately tested. Fear of putting effort into things that provide very little benefit in the short-term but lots of benefits in the long-term. Fear is the mind-killer, the little voice in your head telling you that you are not good enough, that your ideas are crap, that you have no idea what you're talking about.

Seth Godin talks about how when a technology has been labeled dead that the real cool stuff can happen[16] with it. PHP has been labeled dead by the "drive-by technorati" that Mr. Godin talks about, despite the fact that it still well-positioned as a tool to be used when building web applications. At the same time, I can feel that tickle of fear in my brain. "Don't get left behind" is whispers to me. "Drop PHP for Technology X" it pleads to me. When the voice gets too loud, I start reciting the litany against fear. There is still lots of innovation to be done within PHP itself. Each successive release of PHP is getting faster and (quite correctly) implementing features that have made other scripting languages interesting. Exhibit A? Anonymous functions[17], allowing for some very interesting code to be written. It's an integral part of the newest breed[18] of[19] frameworks[20].

Don't let fear dictate the technologies you use. Today, I am most comfortable building web applications using PHP with some Javascript thrown in to improve the user experience. Tomorrow, it will be a different story. Not literally tomorrow. You know what I mean. I've been adding Python, Django[21], Flask, and Google App Engine[22] to the skill set. To be blunt, being a mono-ecosystem (ie PHP only) specialist is a very risky for someone with the "Jack of all trades" career path such as mine. Far better to be standing on the shoulders of giants[23] than to just stay only where you are comfortable.

I guess the point I'm trying to get across is that you can use the *litany against fear* as a tool to motivate to truly analyze your reactions to situations. For example, I'm having a very hard time figuring out how to get Zend Framework + PHPUnit to adequately test the ability of an application at work to use Facebook Connect. I just can't get it to do things with a regular request, for a bunch of reasons I'm unable to understand. I can't peer inside the request to the extent I would like. I foresee having to drop down a level in the application and admit that I cannot test the request itself, but the code that is executed after control is returned to the application.

Fear would say "give up because it's too hard to properly test things". The reality is that while I cannot test the *output* of the request, I can certainly test what is supposed to be going on behind

[16] http://sethgodin.typepad.com/seths_blog/2011/03/bring-me-stuff-thats-dead-please.html
[17] http://ca.php.net/manual/en/functions.anonymous.php
[18] http://lithify.me/
[19] https://github.com/symfony/symfony
[20] https://github.com/zendframework/zf2
[21] https://github.com/chartjes/ibl
[22] https://github.com/chartjes/liesitoldmykids
[23] http://en.wikipedia.org/wiki/Standing_on_the_shoulders_of_giants

the scenes. Tests like that are just as valuable as being able to capture the response from a request and verify that elements exist in the DOM as expected. Perhaps one day there will be better tools in PHP to accomplish this task. I know it can be done using other tools[24] and I'll be honest here: it makes me bark out a few choice words about the state of PHP testing tools.

[24] http://opensoul.org/blog/archives/2009/03/05/testing-facebook-with-cucumber/

12

Where Are The Tests?

This content originally appeared on my blog in October 2008 and is still very applicable today. Hope you enjoy it.

As a project for work[1] gets ready for an alpha release, I've managed to eliminate all the serious bugs and now have some time for what should've been part of the project from the beginning: writing tests. As a lapsed tester, I always feel guilty about not writing tests when things start to get complicated with an application. Whenever you make a change, you have to point-and-click your way through the application to make sure that nothing has broken. I got tired of pointing-and-clicking, so I decided it was time to shut up and write some tests. Since I'm using Code Igniter instead of CakePHP for this project (did I mention that I inherited the project and couldn't switch?) I started looking into the culture of testing surrounding Code Igniter. It's weaker than a newborn baby.

The built-in testing system is some weird mishmash of unit testing and "you have to create your own test results template". WTF? You can't be serious. So I quickly ignored that. Besides, the first set of tests were going to be using PHPUnit[2] combined with Selenium[3] so I could make the "acceptance testing" automated. As an aside to this, I remember speaking with Sebastian Bergmann (the driving force behind PHPUnit) at a conference maybe 3 years ago and him telling me that he knew someone was working on getting Selenium and PHPUnit to play nicely together. Anyway, it's working just fine.

So, next step was to see if I could use something else than the incredibly lame testing tools that Code Igniter came with. I found something called fooStack[4], that contained CIUnit. It lets you use PHPUnit with Code Igniter. Now we're talking! Lickety split I had some unit tests for my models written. Well, after spending quite a lot of time screwing around with the tests and finding out the difference in relative paths between the application running via Apache and trying to test the models via the command line. Thanks to all those on Twitter who yelled "use php_sapi_name to figure out what environment you are in, you idiot!" at me.

But all this got me to thinking: how do some other PHP frameworks stack up in terms of having tests?

[1] http://sportso.com

[2] http://phpunit.de

[3] http://selenium.openqa.org/

[4] /www.foostack.com/foostack/

- Code Igniter has ZERO tests for the framework itself. Perhaps I will mention this to my boss next time I'm asked to create a project from scracth. Hrm, guess I'll have to trust that their code actually works the way it's supposed to.
- Zend Framework (full disclosure: I was one of the original authors of the Zend_Service_-Audioscrobbler component) has huge amounts of tests for the framework itself. In fact, any new contributions MUST have working tests submitted with it.
- CakePHP has lots of tests for the core, and I believe that I saw something in the changelog for CakePHP 1.2RC3 that indicated they were at 80% code coverage in terms of testing for the CakePHP core. Bug submissions for CakePHP are supposed to come with tests to prove that there is a repeatable problem.
- Symfony recently rewrote it's core tests and created their own testing framework for it.

So now when you start comparing frameworks to each other, I think it's important you also consider how much effort has gone into creating tests for the core functionality of that framework. A well tested framework should mean far less surprises when using it. Given how many tools are out there to assist you in testing, I don't think there are any excuses other than being totally lazy to not create tests for the core functionality of any application. Pay now, or pay later, but you will PAY to fix bugs in your application. Paying now is simply cheaper.

Would you stake your project on complex code written by someone else that is simply not testable by you? Perhaps I am not looking at from the proper distance because I am inside the framework-creator culture, but I can't see why you wouldn't want to provide tests to prove your stuff actually works the way you claim it does. Tests prove that the code is behaving the way you expect, and code that is testable is also usually easier to modify as time goes on. There is some sort of sick joy in making a change to code, running the unit tests, and seeing that you didn't break anything.

13

Advice From a Grumpy Programmer

I know there is a lot of stuff in this guide for you to consider. Writing tests is easy but creating a suite of tests that really helps protect you from regressions and other problems is hard.

The scope of knowledge required to do this is vast. This guide is really just scratching the surface of the topic of testing, automation, continuous integration and deployment. There are numerous ways for you to get there so don't get discouraged that you have so many options.

Start slow and build out the pieces you really need first. The fancy stuff that companies like IMVU[1] and Etsy[2] usem didn't happen overnight. It took years to get there, and they made lots of mistakes along the way to find the right set of tools and practices that worked for them.

Take the time and find the rest set of tools for what you need. To reach the nerdvanna of continuous deployment, all you need is this:

- a testing suite with enough coverage to catch regressions
- ability to automatically run tests
- ability to do automated deployments to a specified target

That's really it. Your journey towards this starts with the decision to write one test using one set of tools, and it never really ends. You will find that when you make the commitment to a "test-first" mentality that your code ends up being more modular and simpler than before. That's all you can really ask for.

[1] http://www.startuplessonslearned.com/2009/06/why-continuous-deployment.html

[2] http://codeascraft.etsy.com/