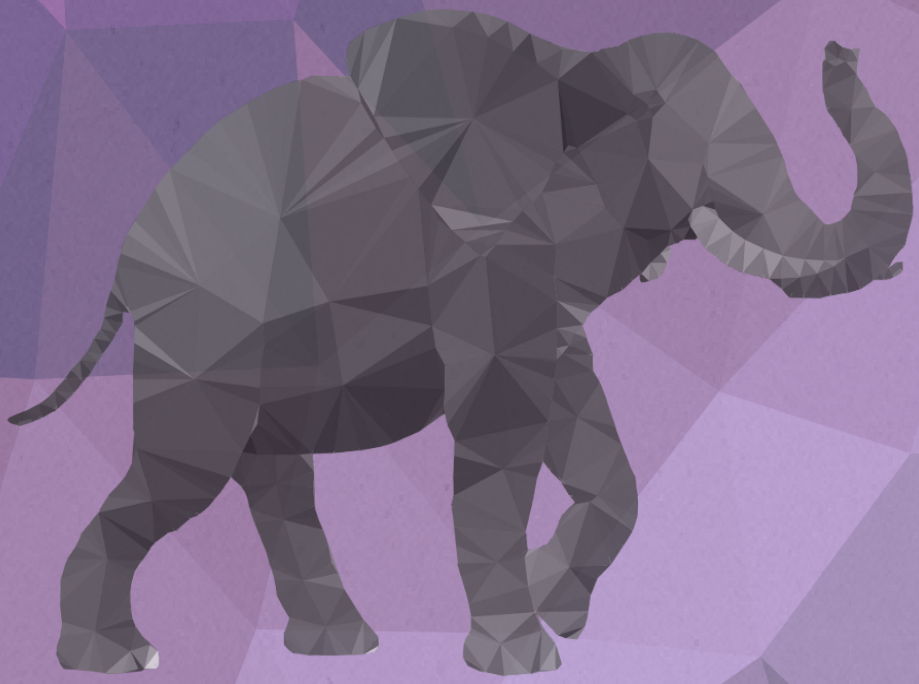
THE CLEAN ARCHITECTURE IN PHP



KRISTOPHER WILSON

The Clean Architecture in PHP

Kristopher Wilson

This book is for sale at http://leanpub.com/cleanphp

This version was published on 2016-04-23

Dedication

First and foremost, I dedicate this book to my wife, ***Ashley****. Thank you for allowing me to spend so much time staring at millions of dots on a screen.*

Secondly, to ***my parents****, who worked so hard to make sure their children had everything they needed and wanted, and for encouraging me to follow my dreams, however odd they may have been.*

Contents

Introduction

Figuring out how to architect a brand new application is a big deal. Doing it the wrong way can lead to a huge headache later. Testing can become hard – or maybe even impossible – and refactoring is an absolute nightmare.

While the methods outlined in this book aren't the only way to go about developing an application, they do provide a framework for developing applications that are:

1. Testable
2. Refactorable
3. Easy to work with
4. Easy to maintain

This book is for anyone wanting to build a medium to large sized application that must be around for a long time and/or be easily enhanced in the future. The methods outlined in this book aren't meant for all applications, and they might be downright over kill for some.

If your application is small, or an unproven, new product, it might be best to just get it out the door as fast as possible. If it grows, or becomes successful, later applying these principles may be a good idea to create a solid, long lasting product.

The principles outlined in this book involve a learning curve. Writing code this way will slow a developer down until the methods become familiar to them.

Organization

This book begins by discussing common problems with PHP code and why having good, solid, clean code is important to the success and longevity of an application. From there, we move on to discussing some principles and design patterns that allow us to solve problems with poor code. Using these concepts, we'll then discuss the Clean Architecture and how it further helps solve problems with bad code.

Finally, in the second half of the book, we dive into some real code and build an application following this architecture. When we're done with our case study application, we'll start swapping out components, libraries, and frameworks with new ones to prove out the principles of the architecture.

The Author

My name is Kristopher Wilson. I've been developing in PHP since around 2000. That sounds impressive on the surface, but most of those years involved writing truly terrible code. I would have benefited greatly from a book like this that outlines the principles of how to cleanly organize code.

I've done it all, from simple websites to e-commerce systems and bulletin boards. Mostly, I've concentrated on working on ERP (Enterprise Resource Planning) systems and OSS (Operational Support Systems) – software that runs the entire back office of large organizations, from manufacturing to telecommunications. I even wrote my own framework once. It was terrible, but that's another story.

I live in Grand Rapids, Michigan with my wife and our four cats (our application to become a registered zoo is still pending). I'm one of the founders of the Grand Rapids PHP Developers (GrPhpDev)

group and am highly involved with organizing, teaching, and learning from the local community.

A Word about Coding Style

I strongly prefer and suggest the use of PSR-2 coding standards[1]. As a community, it makes it much easier to evaluate and contribute to one another's code bases if the dialect is the same. I also strongly suggest the use of DocBlocks and helpful comments on classes and methods.

However, for brevity, the code examples in this book make a few deviations from PSR-2 standards, namely involving bracket placement, and don't include many DocBlocks. If this is too jarring for the PSR-2 and DocBlock fan, like myself, I humbly apologize.

[1] https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-2-coding-style-guide.md

The Problem With Code

Writing code is easy. So easy that there are literally hundreds of books[2] claiming they can teach you to do it in two weeks, ten days, or even twenty-four hours. That makes it sound really easy! The internet is littered with articles on how to do it. It seems like everyone is doing it and blogging about it.

Here's a fun question: would you trust a surgeon or even dentist who learned their profession from a couple of books that taught them how to operate in two weeks? Admittedly, writing code is nothing like opening up and fixing the human body, but as developers, we do deal with a lot of abstract concepts. Things that only exist as a collection of 1s and 0s. So much so that I'd definitely want an experienced, knowledgeable developer working on my project.

The problem with code is that good code, code that serves it's purpose, has little or no defects, can survive and perform it's purpose for a long time, and is easy to change, is quite difficult to accomplish.

[2] http://norvig.com/21-days.html

Writing Good Code is Hard

> If it were easy, everyone would be doing it
>
> *-Somebody, somewhere in history*

Writing code is hard. Well, scratch that: writing code is easy. It's so easy everyone is doing it. Let me start this chapter over.

> If it were easy to be good at it, everyone would be good at it
>
> *-Me, I suppose*

Writing code, especially in PHP, but in many other languages as well, is incredibly easy. Think about the barrier to entry: all you have to do is go download PHP on your Windows machine, type:

```
php -S localhost:1337
```

Now all of your PHP code in that directory is suddenly available for you via a browser. Hitting the ground running developing PHP is easy. On Linux, it's even easier. Install with your distributions package manager and type the same command as above. You don't even have to download a zip file, extract it, worry about getting it in your path, etc.

Not only is getting a server up easy, but actually learning how to get things accomplished in PHP is incredibly easy. Just search the web for "PHP." Go ahead, I'll wait. From Google, I got 2,800,000,000 results. The internet is literally littered with articles, tutorials, and source code relating to PHP.

I chose my words very carefully. The internet is literally *littered* with PHP.

Writing Bad Code is Easy

Since PHP is incredibly easy to get started in, it makes sense that eventually it would gather a large following of developers. The good, the bad, and the ugly. PHP has been around since 1994 or so, as has been gathering developers ever since. At the time of this writing, that's twenty years worth of code being written in PHP.

Since then, an absolute horde of poorly written PHP has shown up on the web, in the form of articles, tutorials, StackOverflow solutions, and open source code. It's also fair to point out that some really stellar PHP has shown up as well. The problem is, writing code the good way (we'll talk about what that means soon) typically tends to be harder. Doing it the down and dirty, quick, and easy to understand way, is, well, easier.

The web has been proliferated with poorly written PHP, and the process of turning out poorly written PHP only naturally increases with the popularity and adoption of the language.

Simply put: it's just way too easy to write bad code in PHP, it's way too easy to find bad code in PHP, it's way too easy to suggest (via putting source code out there or writing tutorials) others write bad code, and it's way too easy for developers to never "level up" their skills.

So why is bad code bad? Let's discuss the results of bad code.

We Can't Test Anything

> We don't have time to write tests, we need to get working software out the door.
>
> -The Project Manager at a previous job

Who has time to write tests? Test are hard, time consuming, and they don't make anybody any money. At least according to project

managers. All of this is absolutely correct. Writing good tests can be challenging. Writing good test can be time consuming. Very rarely will you come across an instance in your life where someone cuts you a check specifically to write software tests.

The Project Manager at my last job who, painfully, was also my boss, absolutely put his foot down to writing tests. Working software out the door was our one and only goal; making that paycheck. What's so incredibly ironic about this is that a robust test suite is the number one way to make it possible to write working software.

Writing tests is supremely important to having a stable, long lasting software application. The mountains of books, articles, and conference talks dedicated to the subject are a testament to that fact. It's also a testament to how hard it is to test, or, more correctly, how important it is to test effectively.

Testing is the single most important means of preventing bugs from happening in your code. While it's not bullet proof and can never catch everything, when executed effectively, it can become a quick, repetitive, and solid way to verify that a lot of the important things in your code – such as calculating taxes or commissions or authentication – is working properly.

There is a direct correlation between how poorly you write your code, and how hard it is to test that code. Bad code is hard to test. So hard to test, in fact, that it leads some to declare testing pointless. The benefits of having tests in place though, cannot be argued.

Why does bad code make tests so hard? Think about a taxing function in our software. How hard would it be to test that taxing functionality if it were spattered about controllers? Or worse yet, spattered about a random collection of `.php` files? You'd essentially have to CURL the application with a set of POST variables and then search through the generated HTML to find the tax rates. That's utterly terrible.

What happens when someone goes in and changes around the tax rates in the database? Now your known set of data is gone. Simple: use a testing database and pump it full of data on each test run. What about when designers change up the layout of the product page, and now your code to "find" the tax rate needs to change? Front-end design should dictate neither business nor testing logic.

It is nearly impossible to test poorly written code.

Change Breaks Everything

The biggest consequence of not being able to test the software is that change breaks everything. You've probably been there before: one little minute change seems to have dire consequences. Even worse, one small change in a specific portion of the application causes errors within a seemingly unrelated portion of the application. These are **regression bugs**, which is a bug caused after introducing new features, fixing other bugs, upgrading a library, changing configuration settings, etc.

When discovered, these regression bugs often lead to exclamations of "We haven't touched that code in forever!" When they're discovered, it's often unknown what caused them in the first place, due to the nature of changes usually happening in "unrelated" portions of the code. The time elapsed before discovering them is often large, especially for obscure portions of the application, or very specific circumstances needed to replicate the bug.

Change breaks everything, because we don't have the proper architecture in place to gracefully make those changes. How often have you hacked something in, and ignored those alarm bells going off in your brain? Further, how often has that hack come around to bite you later in the form of a regression bug?

Without good clean architecture conducive to change, and/or a comprehensive set of test suites, change is a very risky venture

for production applications. I've dealt with numerous systems that the knowledgeable developer declared "stable and working" that they were utterly terrified of changing, because, when they do, "it breaks." Stable, huh?

We Live or Die by the Framework

Frameworks are fantastic. If written well, they speed up application development tremendously. Usually, however, when writing software within a framework, your code is so embedded into that framework that you're essentially entering a long term contract with that framework, especially if you expect your project to be long lived.

Frameworks are born every year, and die every once-in-awhile, too (read: CodeIgniter, Zend Framework 1, Symfony 1). If you're writing your application in a framework, especially doing so the framework documented way, you're essentially tying the success and longevity of your application to that of the framework.

We'll discuss this much more in later chapters, and go into a specific instance where my team and I failed to properly prepare our application for the death of our chosen framework. For now, know this: there is a way to write code, using a framework, in such a way that switching out the framework shouldn't lead to a complete rewrite of the application.

We Want to Use All the Libraries

Composer[3] and Packagist[4] brought with them a huge proliferation of PHP libraries, frameworks, components, and packages. It's now easier than it ever has been to solve problems in PHP. The wide

[3] https://getcomposer.org/
[4] https://packagist.org/

range of available libraries, installed quickly and simply through Composer, make it easy to use other developer's code to solve your problems.

Just like the framework, though, using these libraries comes at a cost: if the developer decides to abandon them, you're faced with no choice but eventually replacing it with something else. If you've littered your code base with usages of this library, you now have a time consuming process to run through to upgrade your application to use some other library.

And of course, later, we'll describe how to gracefully handle this problem in a way that involves minimal rewriting, and hopefully minimal bugs if you've written a good test suite to verify your success.

Writing Good Code

Writing good code is hard.

The goal of this book is to solve these problems with bad code. We'll discuss how architecture plays a key role in both causing and solving these problems, and then discuss ways in which to correct or at least mitigate these issues, such that we can build strong, stable, and long-lasting software applications.

What is Architecture?

This chapter is only included in the full copy of the book.

You may purchase it from https://leanpub.com/cleanphp.

Coupling, The Enemy

This chapter is only included in the full copy of the book.

You may purchase it from https://leanpub.com/cleanphp.

Your Decoupling Toolbox

We've uncovered various ways in which poor design decisions can lead to code that is hard to maintain, hard to refactor, and hard to test. Now we're going to look at some guiding principles and design patterns that will help us alleviate these problems and help us write better code. Later, when we talk about architecture, we'll apply these principles further to discover how to create truly uncoupled, refactorable, and easily testable code.

SOLID Design Principles

Much like design patterns are a common language of constructs shared between developers, SOLID Design Principles are a common foundation employed across all types of applications and programming languages.

The **SOLID** Design Principles are a set of five basic design principles for object oriented programming described by Robert C. Martin[5]. These principles define ways in which all classes should behave and interact with one another, as well as principles of how we organize those classes.

The SOLID principles are:

1. **S**ingle Responsibility Principle
2. **O**pen/Closed Principle
3. **L**iskov Substitution Principle
4. **I**nterface Segregation Principle
5. **D**ependency Inversion Principle

Single Responsibility Principle

The **Single Responsibility Principle** states that objects should have one, and only one, purpose. This is a principle that is very often violated, especially by new programmers. Often you'll see code where a class is a jack of all trades, performing several tasks, within sometimes several thousand lines of code, all depending on what method was called.

[5] http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

To the new OOP developer, classes are often viewed at first as a collection of related methods and functionality. However, the SRP advocates against writing classes with more than one responsibility. Instead, it recommends condensed, smaller classes with a single responsibility.

What is a responsibility?

In his description of the Single Responsibility Principle, Robert Martin describes a responsibility as "a reason for change." Any time we look at a given class and see more than one way in which we might change it, then that class has more than one responsibility.

Another way to look at a responsibility is to look at the behavior of a class. If it has more than one behavior, it is violating SRP.

Let's look at a class that represents a `User` record stored in a database:

```
class User {
  public function getName() {}
  public function getEmail() {}

  public function find($id) {}
  public function save() {}
}
```

This `User` class has two responsibilities: it manages the state of the user, and it manages the retrieval from and persistence to the database. This violates SRP. Instead, we could refactor this into two classes:

```
class User {
  public function getName() {}
  public function getEmail() {}
}

class UserRepository {
  public function find($id) {}
  public function save(User $user) {}
}
```

The `User` class continues to manage the state of the user data, but now the `UserRepository` class is responsible for managing the retrieval and persistence to the database. These two concepts are now decoupled, and the two classes conform to SRP.

When we look at the `UserRepository` class, we can make a determination that retrieving and persisting data to the database are the same responsibility, as a change to one (such as changing where or how the data is stored) requires a change to the other.

Breaking up Classes

In order to apply the SRP principle to existing classes, or even when creating new classes, it's important to analyze the responsibility of the class. Take for instance a customer invoicing class, like the one we looked at in the previous chapter:

```
class InvoicingService {
  public function generateAndSendInvoices() {}
  protected function generateInvoice($customer) {}
  protected function createInvoiceFile($invoice) {}
  protected function sendInvoice($invoice) {}
}
```

Already it's plainly obvious that this class has more than one responsibility. Just looking at the method name of `generateAndSendInvoices()` reveals two. It's not always readily apparent from class and method names how many responsibilities there are, though. Sometimes it requires looking at the actual code within those methods. After all, the method could have simply been named `generateInvoices()`, hiding the fact that it was also responsible for delivering those invoices.

There are at least four separate responsibilities of this class:

1. Figuring out which invoices to create
2. Generating invoice records in the database
3. Generating the physical representation of the invoice (i.e.: PDF, Excel, CSV, etc)
4. Sending the invoice to the customer via some means

In order to fix this class to conform to SRP, we'll want to break it up into smaller, fine tuned classes, each representing one of the four responsibilities we identified above, plus the `InvoicingService` class that ties this all together.

```
class OrderRepository {
  public function getOrdersByMonth($month);
}

class InvoicingService {
  public function generateAndSendInvoices() {}
}

class InvoiceFactory {
  public function createInvoice(Order $order) {}
}
```

```
class InvoiceGenerator {
  public function createInvoiceFormat(
    Invoice $invoice,
    $format
  ) {}
}

class InvoiceDeliveryService {
  public function sendInvoice(
    Invoice $invoice,
    $method
  ) {}
}
```

Our four classes here represent the responsibilities of the previous `InvoicingService` class. Ideally, we'd probably even have more than this: we'll probably want strategy classes for each format needed for the `InvoiceGenerator` and strategy classes for each delivery method of the `InvoiceDeliveryService`. Otherwise, these classes end up having more than one responsibility as they're either generating multiple file formats, or utilizing multiple delivery methods.

This is a lot of classes, almost seemingly a silly number of classes. What we've given up, however, is one very large, monolithic class with multiple responsibilities. Each time we need to make a change to one of those responsibilities, we potentially risk introducing an unintended defect in the rest of the seemingly unrelated code.

Why does SRP matter?

Why are we concerned with making sure a class only has one responsibility? Having more than one responsibility makes those responsibilities coupled, even if they are not related. This can make it harder to refactor the class without unintentionally breaking

something else, whereas having a separate class for each responsibility shields the rest of the code from most of the risk.

It's also much easier to test a class with only one responsibility: there's only one thing to test, although with a potential for many different outcomes, and there's much less code involved in that test.

Generally, the smaller the class, the easier it is to test, the easier it is to refactor, and the less likely it is to be prone to defects.

Open/Closed Principle

The **Open/Closed Principle** states that classes should be open to extension, but closed to modification. This means that future developers working on the system should not be allowed or encouraged to modify the source of existing classes, but instead find ways to extend the existing classes to provide new functionality.

The Strategy Pattern introduced in the previous chapter of Design Patterns provides a great example of how the Open/Closed Principle works. In it, we defined strategies for mechanisms to deliver invoices to customers. If we wanted to add a new delivery method, perhaps one via an EDI (Electronic Data Interchange), we could simply write a new adapter:

```
class EdiStrategy implements DeliveryInterface {
    public function send(Invoice $invoice) {
        // Use an EDI library to send this invoice
    }
}
```

Now the invoice process has the ability to deliver invoices via EDI without us having to make modifications to the actual invoicing code.

The OCP in PHPUnit

The PHPUnit testing framework provides a great example of how a class can be open to extension, but closed for modification. The `PHPUnit_Extensions_Database_TestCase` abstract class requires that each individual test case provide a `getDataSet()` method, which should return an instance of `PHPUnit_Extensions_Database_DataSet_IDataSet`, an interface. PHPUnit provides several implementations of this interface, including a `CsvDataSet`, `XmlDataSet`, `YamlDataSet`, etc.

If you decided you wanted to provide your data sets as plain PHP arrays, you could write your own data set provider class to do so, simply by implementing the `IDataSet` interface. Using this new class, we could provide the `TestCase` class with a data set, parsed from PHP arrays, that works and acts like any of the built-in PHPUnit data sets.

```
class MyTest extends DatabaseTestCase {
  public function getDataSet() {
    return new ArrayDataSet([]);
  }
}
```

The internal code of PHPUnit has not been modified, but now it is able to process pure PHP arrays as data sets. [6]

https://github.com/mrkrstphr/dbunit-fixture-arrays.

Why does OCP matter?

The benefit of the Open/Closed Principle is that it limits the direct modification of existing source code. The more often code is changed, the more likely it is to introduce unintended side effects

[6] If you want to see a complete example of this concept in action, checkout

and cause defects. When the code is extended as in the examples above, the scope for potential defects is limited to the specific code using the extension.

Liskov Substitution Principle

The **Liskov Substitution Principle** says that objects of the same interface should be interchangeable without affecting the behavior of the client program [7].

This principle sounds confusing at first, but is one of the easiest to understand. In PHP, interfaces give us the ability to define the structure of a class, and then follow that with as many different concrete implementations as we want. The LSP states that all of these concrete implementations should be interchangeable without affecting the behavior of the program.

So if we had an interface for greetings, with various implementations:

```
interface HelloInterface {
  public function getHello();
}

class EnglishHello implements HelloInterface {
  public function getHello() {
    return "Hello";
  }
}

class SpanishHello implements HelloInterface {
  public function getHello() {
    return "Hola";
```

[7] http://www.objectmentor.com/resources/articles/lsp.pdf

```
  }
}

class FrenchHello implements HelloInterface {
  public function getHello() {
    return "Bonjour";
  }
}
```

These concrete `Hello` classes should be interchangeable. If we had a client class using them, the behavior shouldn't be affected by swapping them for one another:

```
class Greeter {
  public function sayHello(HelloInterface $hello) {
    echo $hello->getHello() . "!\n";
  }
}

$greeter = new Greeter();
$greeter->sayHello(new EnglishHello());
$greeter->sayHello(new SpanishHello());
$greeter->sayHello(new FrenchHello());
```

While the output may be different, which is desired in this example, the behavior is not. The code still says "hello" no matter which concrete instance of the interface we give it.

LSP in PHPUnit

We already discussed an example of the Liskov Substitution Principle when we discussed the Open/Closed Principle. The `ArrayDataSet` class we defined as an instance of PHPUnit's `IDataSet` is returned from the `getDataSet()` method of DbUnit's `DatabaseTestCase` abstract class.

```
class MyTest extends DatabaseTestCase {
  public function getDataSet() {
    return new ArrayDataSet([]);
  }
}
```

The PHPUnit `DatabaseTestCase` class expects that the `getDataSet()` method will return an instance of `IDataSet`, but doesn't necessarily care what implementation you give it, so long as it conforms to the interface. This is also referred to as design by contract, which we'll talk about in much more detail in Dependency Injection.

The key point of the Liskov Substitution Principle is that the behavior of the client code shall remain unchanged. Regardless of what implementation of `IDataSet` we return from `getDataSet()`, it will result in the data set being loaded into the database for unit tests to be run against. It doesn't matter if that data came from CSV, JSON, XML, or from our new PHP array class: the behavior of the unit tests remain the same.

Why does LSP matter?

In order for code to be easily refactorable, the Liskov Substitution Principle is key. It allows us to modify the behavior of the program, by providing a different instance of an interface, without actually modifying the code of the program. Any client code dependent upon an interface will continue to function regardless of what implementation is given.

In fact, as we've already seen, the Liskov Substitution Principle goes hand-in-hand with the Open/Closed Principle.

Interface Segregation Principle

The **Interface Segregation Principle** dictates that client code should not be forced to depend on methods it does not use[8]. The principle intends to fix the problem of "fat" interfaces which define many method signatures. It relates slightly to the Single Responsibility Principle in that interfaces should only have a single responsibility. If not, they're going to have excess method baggage that client code must also couple with.

Consider for a moment the following interface:

```
interface LoggerInterface {
  public function write($message);
  public function read($messageCount);
}
```

This interface defines a logging mechanism, but leaves the details up to the concrete implementations. We have a mechanism to write to a log in `write()`, and a mechanism to read from the log file in `read()`.

Our first implementation of this interface might be a simple file logger:

[8] http://www.objectmentor.com/resources/articles/isp.pdf

```
class FileLogger implements LoggerInterface {
  protected $file;

  public function __construct($file) {
    $this->file = new \SplFileObject($file);
  }

  public function write($message) {
    $this->file->fwrite($message);
  }

  public function read($messageCount)
  {
    $lines = 0;
    $contents = [];

    while (!$this->file->eof()
      && $lines < $messageCount) {

      $contents[] = $this->file->fgets();

      $lines++;
    }

    return $contents;
  }
}
```

As we continue along, though, we decide we want to log some critical things by sending via email. So naturally, we add an `EmailLogger` to fit our interface:

```
class EmailLogger implements LoggerInterface {
  protected $address;

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

  public function write($message) {
    // hopefully something better than this:
    mail($this->address, 'Alert!', $message);
  }

  public function read($messageCount)
  {
     // hmm...
  }
}
```

Do we really want our application connecting to a mailbox to try to read logs? And how are we even going to sift through the email to find which are logs and which are, well, emails?

It makes sense when we're doing a file logger that we can easily also write some kind of UI for viewing the logs within our application, but that doesn't make a whole lot of sense for email.

But since `LoggerInterface` requires a `read()` method, we're stuck.

This is where the Interface Segregation Principle comes into play. It advocates for "skinny" interfaces and logical groupings of methods within interfaces. For our example, we might define a `LogWriterInterface` and a `LogReaderInterface`:

```
interface LogWriterInterface {
  public function write($message);
}

interface LogReaderInterface {
  public function read($messageCount);
}
```

Now `FileLogger` can implement both `LogWriterInterface` and `LogReaderInterface`, while `EmailLogger` can implement only `LogWriterInterface` and doesn't need to bother implementing the `write()` method.

Further, if we needed to sometimes rely on a logger that can read and write, we could define a `LogManagerInterface`:

```
interface LogManagerInterface
  extends LogReaderInterface, LogWriterInterface {
}
```

Our `FileLogger` can then implement the `LogManagerInterface` and fulfill the needs of anything that has to both read and write log files.

Why does ISP matter?

The goal of the Interface Segregation Principle is to provide decoupled code. All client code that uses the implementation of an interface is coupled to all methods of that interface, whether it uses them or not, and can be subject to defects when refactoring within that interface occur, unrelated to what implementations it actually uses.

Dependency Inversion Principle

The **Dependency Inversion Principle** states that [9]:

> A. High level modules should not depend upon low level modules. Both should depend upon abstractions.

and that:

> B. Abstractions should not depend upon details. Details should depend upon abstractions.

This principle is very core to The Clean Architecture, and we'll discuss how it fits in great detail in that leading chapter.

Imagine a class that controls a simple game. The game is responsible for accepting user input, and displaying results on a screen. This `GameManager` class is a high level class responsible for managing several low level components:

```
class GameManager {
  protected $input;
  protected $video;

  public function __construct() {
    $this->input = new KeyboardInput();
    $this->video = new ScreenOutput();
  }

  public function run() {
    // accept user input from $this->input
```

[9] http://www.objectmentor.com/resources/articles/dip.pdf

```
    // draw the game state on $this->video
  }
}
```

This `GameManager` class is depending strongly on two low level classes: `KeyboardInput` and `ScreenOutput`. This presents a problem in that, if we ever want to change how input or output are handled in this class, such as switching to a joystick or terminal output, or switch platforms entirely, we can't. We have a hard dependency on these two classes.

If we follow some guidelines of the Liskov Substitution Principle, we can easily devise a system in which we have a `GameManager` that allows for the input and outputs to be switched, without affecting the output of the `GameManager` class:

```
class GameManager {
  protected $input;
  protected $video;

  public function __construct(
    InputInterface $input,
    OutputInterface $output
  ) {
    $this->input = $input;
    $this->video = $output;
  }

  public function run() {
    // accept user input from $this->input
    // draw the game state on $this->video
  }
}
```

Now we've inverted this dependency to rely on `InputInterface`

and `OutputInterface`, which are abstractions instead of concretions, and now our high level `GameManager` class is no longer tied to the low level `KeyboardInput` and `ScreenOutput` classes.

We can have the `KeyboardInput` and `ScreenOutput` classes extend from these interfaces, and add additional ones, such as `JoystickInput` and `TerminalOutput` that can be swapped at run time:

```
class KeyboardInput implements InputInterface {
  public function getInputEvent() { }
}

class JoystickInput implements InputInterface {
  public function getInputEvent() { }
}

class ScreenOutput implements OutputInterface {
  public function render() { }
}

class TerminalOutput implements OutputInterface {
  public function render() { }
}
```

We're also utilizing what's known as Dependency Injection here, which we'll talk about in the next chapter, conveniently called Dependency Injection.

If we can't modify the input and output classes to conform to our interfaces, if they're maybe provided by the system, it would then be smart to utilize the Adapter Pattern we previously discussed to wrap these existing objects and make them conform to our interface.

Why does DIP matter?

In general, to reach a decoupled code base, one should get to a point where dependency only flows inward. Things that change

frequently, which are the high level layers, should only depend on things that change rarely, the lower levels. And the lower levels should never depend on anything that changes frequently, which is the higher level layers.

We follow this philosophy to make it easier for change to happen in the future, and for that change to have as little impact upon the existing code as possible. When refactoring code, we only want the refactored code to be vulnerable to defects; nothing else.

Applying SOLID Principles

The SOLID principles work tightly together to enforce code that is easy to extend, refactor, and test, which ultimately leads to less defects and quicker turn around time on new features.

Just as we continued building on our Design Patterns in this chapter, we'll continue building on the principles of SOLID as we discuss Inversion of Control and the Clean Architecture later. The SOLID principles are the founding principles that make the Clean Architecture work.

The Clean Architecture in PHP

Thank you for evaluating this sample of The Clean Architecture in PHP.

To purchase the rest of this book, please visit https://leanpub.com/cleanphp.