Getting Stuff Done *with Laravel 4*

A journey through application design and development using PHP's hottest new framework

by Chuck Heintzelman

Getting Stuff Done with Laravel 4

A journey through application design and development using PHP's hottest new framework

Chuck Heintzelman

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

This version was published on 2014-02-02

Tweet This Book!

Please help Chuck Heintzelman by spreading the word about this book on Twitter!

The suggested hashtag for this book is #LaravelGSD.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#LaravelGSD

Contents

Thank You

I want to sincerely thank you for purchasing this book. I hope you find it engaging, entertaining, and most of all, useful.

Other Places to Learn Laravel

- The website[1]. This is always my first stop to look something up. Check out the forums there. It's chock-full of information.
- NetTuts[2]. There's some nice Laravel tutorials on the site.
- Laravel Testing Decoded[3] by Jeffery Way. This book is an awesome resource on how to test your Laravel code.
- Code Bright[4] by Dayle Rees. This book is both fun and informative.
- From Apprentice to Artisan[5] by Taylor Otwell. By the creator of Laravel … need I say more.
- Implementing Laravel[6] by Chris Fidao. This book focuses on implementing projects with Laravel. It covers structure and and common patterns. A great book.
- Laravel 4 Cookbook[7] by Christopher Pitt. This book contains various projects built in Laravel 4.
- Laravel in Action[8] by Maks Surguy. This book is now available from Manning Publications's Early Access program.

---

[1] http://laravel.com
[2] http://net.tutsplus.com/
[3] http://leanpub.com/laravel-testing-decoded
[4] http://leanpub.com/codebright
[5] https://leanpub.com/laravel
[6] https://leanpub.com/implementinglaravel
[7] https://leanpub.com/laravel4cookbook
[8] http://www.manning.com/surguy/

Revision History

Current version: 1.2

Version	Date	Notes
1.2	02-Feb-2014	Typos and updates for Laravel 4.1
1.1	28-Nov-2013	Typos and general cleanup.
1.0	2-Nov-2013	Fixed many typos and released on Leanpub.
0.9	27-Oct-2013	Finished and released on Leanpub.
0.8	20-Oct-2013	Added 4 chapters, released on Leanpub.
0.7	13-Oct-2013	Added 3 chapters, two appendices. Released on Leanpub.
0.6	6-Oct-2013	Added 8 chapters, finishing Part 3. Released on Leanpub.
0.5	29-Sep-2013	Added 7 chapters to Part 3 and released on Leanpub.
0.4	22-Sep-2013	Added 7 chapters to Part 3 and released on Leanpub.
0.3	15-Sep-2013	Cleanup draft through Part 2. Decided to release first version on Leanpub.
0.2	8-Sep-2013	Finish 1st draft of Part 2
0.1	31-Aug-2013	Finish 1st draft of Welcome and Part 1
0.0	3-Aug-2013	Start writing first draft

A special thank you

Here's a list of non-anonymous people who have helped me by finding typos and other issues in these pages.

- Peter Steenbergen
- Jeremy Vaught
- George Gombay
- Mike Bullock
- Kristian Edlund

Thank you very much!

Cover Image

[9] http://www.dreamstime.com/kemaltaner_info
[10] http://www.dreamstime.com/

About the Sample Version

Since you've downloaded the "Sample" version of the book, you're not receiving entire thing.

I'm providing the eight chapters from the **Welcome** section in their entirety. Plus the chapter on **SOLID Object Design**. This will give you the flavor of my writing and explain why this book will be useful.

Contents of Paid Version

If you look at The Leanpub Page[11] the entire Table of Contents is presented for you to examine.

Go there. Check it out. See if anything helps you decide to purchase the complete version of this book.

[11] https://leanpub.com/gettingstuffdonelaravel

Welcome

Welcome to *Getting Stuff Done with Laravel*. The **Welcome** part of the book explains what you'll get out of the book.

Here's how things are broadly organized.

Welcome
: The first part of the book explains what you'll get out of the book.

Part 1 - Design Philosophies and Principles
: This part talks about general principles of design we'll follow in creating the application.

Part 2 - Designing the Application
: This is the part where we design the actual application.

Part 3 - The Console Application
: Next, we make the application usable from the console.

Part 4 - The Web Application
: Now we'll take our creation and put a web skin on it. **Mwaa**, **haa**, **ha**.

Appendices
: Supplemental information. Like how to install Composer.

Chapter 1 - This book's purpose

This book will take you on a journey through Laravel. Hopefully, you'll go places you've never been and see you things you've never seen. It's a travelogue of sorts. We'll have a definite destination (the application we're creating) and I'll point out some amazing sights along the way. When you reach the end, drop me a note at chuckh@gmail.com[12]. I'm very interested in what you thought of the journey.

This book is meant to be experienced. To be used. Please follow along and build the application chapter by chapter. Each chapter leads to the next. The sections within a chapter flow forward. Each part of the book builds on the previous.

You could think of the sections in each chapter as cities. Then the chapters themselves are countries, and the book's parts are continents and … *Okay, enough with the labored traveling analogy.*

The focus throughout the book is the step-by-step creation of an application using Laravel 4.

This is not a typical technical manual

I've attempted to mimic the actual process of design and development as closely as possible. This means there are false starts, design changes, and refactoring along the way.

You've been warned <grin>.

What's not in this book

- Every aspect of Laravel. This is not a reference book on the entire framework.
- Caching, Events, or Logging. These are important topics, but the application we're creating doesn't require them.
- Queues, Authentication, Cookies, or Sessions. Again, important stuff, but we don't need it.
- Database. Yeah, it almost pains me to admit this. One of the greatest aspects of Laravel is it's *Fluent Query Builder* and *Eloquent ORM*. I mean, what great names. Names that the implementation fully lives up to. Sadly, I don't touch on this because … you guessed it … the application we're creating doesn't need it.

[12] mailto:chuckh@gmail.com

What's in this book

Mostly, me blabbing away about why I'm doing what I'm doing in creating the application. You may agree with me some of the time. You may argue with me some of the time. Sometimes you may think I'm a complete idiot. Hopefully, at times you'll think *"Oh yeah. Good one."* But in the end, you're getting the nuts-and-bolts of creating a real system that you can use.

Chapter 2 - Who are you?

Most books start with information about the author but the more important question really is "who are you?"

I'm making the following assumptions:

- You know more about computers than most people.
- You are a programmer.
- You know how to program in PHP. Maybe a little. Maybe a lot.
- You've heard of Laravel[13]. *(This isn't a deal-breaker because I'm going to tell you about it.)*
- You love programming or want to get back the passion that first drove you to make computers do your bidding.
- Your name is not Taylor Otwell because if it is, then I'm not worthy.

I'll do my best to make the material approachable to beginners, yet interesting and in-depth enough for intermediate programmers to find the material useful.

Bottom line

You want to learn more about Laravel.

[13] http://laravel.com

Chapter 3 - Who am I?

This is the typical rah-rah, ain't I great chapter. It's not going to teach you a single thing about Laravel. The smartest move you could make right now is to skip to the next chapter.

Hello. My name is Chuck Heintzelman and I write computer programs.

(That felt like I was in front a support group. I hope nobody said "Hi Chuck.")

Seriously. I've written programs since that day in 9th grade when I stayed home "sick" from school with a borrowed *BASIC Language Reference* manual and wrote out on paper a game that was like Asteroids[14] except instead of asteroids flying at you it was other ships firing long white blocks of death at you.

After long hours of debugging and waiting for the TRS-80 to load/save my program to it's "mass storage" (a cassette tape), the game finally worked. This was 33 years ago. Back in the day of computer dinosaurs, large ferocious beasts filling climate-controlled rooms. No, I've never *actually* used punched cards, but have seen them in use.

Since then I've written programs in Fortran, COBOL (yeah, I know), Assembly Language, Basic, C, C++, C#, Java, Pascal, Perl, Javascript, and PHP. I've tinkered with many, many other languages, but have not written programs that people actually used.

I've created systems for Fortune 500 companies, as well as small Mom-and-Pop stores. Everything from mail order systems running in Xenix to web applications running in PHP. I've started several companies before the days of the Internet *(not before the **real** beginning of the Internet, just before the excitement starting in the mid-90s)*, and a few dot coms since then. And through it all I've did what I loved to do–write computer programs.

Whew! Okay, enough about how great I am.

Here's my point

Throughout my career I've never felt the need to create a book about programming until now. The sole reason I'm writing this is because of Laravel.

[14] http://en.wikipedia.org/wiki/Asteroids_(video_game)

Chapter 4 - What is Laravel?

Raise your hand if this sounds familiar

> *You've been tasked with adding a feature to your company's existing system. Unfortunately, the system was written in PHP 4 and whoever the original programmer was, you suspect they watched a few too many "Wordpress Gone Wild" videos.*
>
> *You've inherited this codebase with *gasp* no classes, a glut of global variables, and a structure not unlike a 50,000 piece jigsaw puzzle.*
>
> *You curse your job, the short sightedness of the management-team, and whatever possessed you to want to make money by programming in the first place.*
>
> *After all, programming should be fun. Right?*

We've all been there.

Enter Laravel.

(Cue the sound of kettle drums: duh-duh duh-duh duh-da-duh)

Laravel is a framework for PHP which makes programming fun again.

Come on man ... it's just a framework

Laravel is not a new language. It's only a framework. If you cut through all the hyperbole and look at its essence, Laravel is simply a PHP Framework.

Although, I do agree with the motto from the Laravel web site:

The PHP Framework for Web Artisans.

Ruby On Rails is *just a framework.* Yet look at the fandom behind it.

Laravel's not going to magically fix your PHP spaghetti code, but it provides you with a new, fast and elegant way to get stuff done. *(Note, the concept of Getting Stuff Done is a reoccurring theme in this book.)*

In short, Laravel provides you with an architecture that makes PHP programming a joy. You'll be able to refactor existing code in a way that is expressive, stylish, and will be easy to maintain and expand in the future.

Laravel is not a panacea. If your existing codebase sucks, it's going to be painful to get from **where it is now** to **where it should be**. That's the nature of our industry.

But, if you want to move to a framework that allows simple expressivity *(is that even a word?)* then Laravel is the answer.

Chapter 5 - How to justify Laravel

Here's the problem (or a problem) ...

> *You must work under the constraints your company places on you. Namely, that you must support the existing software and develop new code that plays nice with your existing systems. There's a mix of .NET, some Java, but most of the existing code is PHP.*

You've recently discovered Laravel and like it and want to use it for new development.

How can you justify switching to Laravel?

Let's put on our detective hat for a minute.

Hmmm. The detectives I know (from TV of course) seem to follow the money when looking for suspects and motives. So let's follow the money ...

Customers provide money to businesses in exchange for goods and services. The better the product, and the more customers really want the product, the more money they fork over to the business.

Managers want the business to thrive. They want as many customers as possible to give them as much money as possible as frequently as possible.

Think about it from management's perspective ...

- I want my customers to be happy.
- I want new customers.
- Customer happiness equates to expectations being met.
- I want my programmers to be able to deliver the requirements on time.
- I want the programming team to be agile. *(Whatever that means ... see the box below.)*
- I want to facilitate customer's requests in a timely manner.
- I want great developers delivering great products

What does Agile even mean?

You ever say or write a word so often that it loses all meaning? Almost like the Smurfs ... everything is smurfing, smurfable, smurferific. Agile seems to be one of those words. It's way

> past the buzzword stage. Everything is Agile this, Agile that. Are people talking about the iterative software process or something else? Something magical? I really don't know.

If the above list is the management perspective, then Laravel is easily justified:

- Customers are happy when their needs are addressed and met.
- Customers are even happier when their expectations are exceeded.
- Laravel provides a framework that …
  - Makes it easy to extend functionality.
  - Follows best-practices in design.
  - Allows multiple programmers to collaborate efficiently.
  - Makes programmers happy. *(Remember managers: a happy programmer is a productive programmer.)*
  - Let's stuff get done faster.
  - Encourages unit testing, considering testing a core component of every application.

Laravel provides managers the ability for their programmers to get more done, quicker, and eliminates many of the obstacles inherent in web application development. I'll expand on this later.

Pretty easy to justify, ain't it?

Chapter 6 - Why programmers like Laravel

Let's cut to the chase … why would you, as a programmer, want to use Laravel as a framework?

Let me talk a bit about Framework Envy.

> *(Here I picture talking to a therapist. Him nodding sagely, taking a drag on his pipe and saying, "Talk about zee framework envy.")*

I'd been given projects written in PHP. These were bloated, PHP 4 projects written by a developer whose only concept of "**class**" was that it is something at school to be skipped. And I'd look across the street at the Ruby developers and silently wish for some natural disaster–earthquake, tornado, even lightning–to level their building.

Does this make me a bad person?

This was at a time when Ruby was all shiny and new. What made Ruby cool wasn't the language itself (although there are very nice aspects to the language). No, what made Ruby cool was Ruby on Rails.

All the developers were flocking to Ruby on Rails.

Why were they flocking to it?

Because it promised a way of development that was fun. And by fun, I mean powerful, expressive, and quick to implement. I credit RoR on creating an atmosphere making programming a delight again. The coding joy instilled by RoR is the exact same feeling as that initial impetus that made us all want to be programmers.

How sad was it that we were mired in the PHP world? Where any Tom, Dick or Henrietta was a "PHP Programmer" because they could hack a Wordpress install.

(See the next chapter about Wordpress - The Good, The Bad, The Ugly)

But, no, we were stuck with the requirements that our projects be in PHP. We couldn't be a cool kid like all those Ruby developers. They were cutting edge. They were the ones pushing the boundaries, making a name for themselves.

Along comes Laravel. It takes the best of Ruby on Rails and brings it to the PHP world. Suddenly, a PHP developer is dealing with routes to controllers instead of individual scripts. Concepts like DRY (Don't Repeat Yourself) now have more meaning. Suddenly, we have a "blade" template engine incorporating the essence of PHP in a way Smarty Templates only dreamed of. We have, quite literally, the potential of PHP Nirvana.

Does it sound like I think Laravel's awesome? I hope so.

Chapter 7 - Wordpress: The Good, The Bad, The Ugly

Wordpress revolutionized blogging. It brought blogging to the masses. Sure, there are other platforms like blogger and livejournal, but what Wordpress did was put out in the public domain a large, popular system written PHP.

With the advent of Wordpress, anybody could hack the PHP scripts to make the blogging platform do what they wanted it do it.

> "With great power comes great responsibility." – Uncle Ben (from Spiderman)

Unfortunately, the power Wordpress availed was not met with great responsibility. Scripts were hacked with no thought toward overall design or usability. To make matters worse, Wordpress started in the days of PHP 4, when the language didn't allow the constructs that allowed true programmers to create maintainable systems.

Wordpress was the best thing that happened to PHP, but it also was the worst thing that happened to the language.

It's a case of too much success in the hands of too few artisans.

This attached a stigma to PHP.

Softwarati[15]
: Self absorbed programming intellectuals who comment on languages.

For your consideration … a commonly heard quote by the Softwarati:

> "Oh. PHP's a ghetto language. Ugly, hardly maintainable, but it works … most of the time"

Thank goodness Laravel came along to kick those Softwarati in their upturned noses.

[15] Yes, this is a word I totally made up.

Chapter 8 - Conventions Used in this Book

There are several conventions used through this book.

Code is indented 2 spaces

Usually, I indent code 4 spaces but since this book is available in a variety of eBook formats some of the smaller screens have horizontal space at a premium.

```
1  for ($i = 0; $i < 10; $i++)
2  {
3    echo "I can count to ", $i, "\n";
4  }
```

This is a tip

It is used to highlight a particularly useful piece of information.

This is a warning

It is used to warn you about something to be careful of.

This is an information block

Used to reiterate an important piece of information

This is something to do

When there's code, or other actions you should take, it's always proceeded by this symbol.

Trailing ?> is used when opening tag is used.

When coding, I always drop the trailing ?> in a file. But the editor I'm writing this book in makes everything look wonky when I do it. So, within this book, if I open a PHP block with the PHP tag, I always close it in the code too. For example:

```
<?php
class SomethingOrOther {
    private $dummy;
}
?>
```

PHP Opening and Closing Tags

In the code examples sometimes the opening PHP tag (`<?php`) is used when it's not needed (such as when showing a portion of a file.) Sometimes the closing PHP tag ('?>') is used when it's not needed.

```
<?php
  function somethingOrOther()
  {
    $this->callSetup();
  }
?>
```

With real PHP Code I *always* omit the closing tag at the end of the file. I'll leave it to you to determine whether or not the tags are needed. Be aware the opening and closing tags in the code examples should not be taken verbatim.

What OS am I using?

I'm writing this manual, the code, etc., using Linux Mint 16[16] which based on Debian and Ubuntu. It's basically the same as Ubuntu 13.10[17].

[16] http://www.linuxmint.com/
[17] http://www.ubuntu.com/

Part 1 - Design Philosophies and Principles

There's not much code in this part of the book. Sorry, it's all about the design at this point. Here I'll discuss general design principles used in building the application.

You may be thinking, "just take me to the code." For the most part, I concur. It's often quickest and easiest just to jump in the code and learn by doing. If you understand the concepts: SOLID Object Design, Interface as Contract, Dependency Injection, Decoupling, and Inversion of Control, then skip to Part 2 to begin designing the application.

Chapter 16 - SOLID Object Design

There is a set of principles in object-oriented design called **SOLID**. It's an acronym standing for:

- **[S]**ingle Responsibility
- **[O]**pen/Closed Principle
- **[L]**iskov Substitution Principle
- **[I]**nterface Segregation Principle
- **[D]**ependency Inversion Principle

Together, they represent a set of best practices which, when followed, makes it more likely software you develop will be easier to maintain and extend over time.

This chapter explains each principle in greater detail.

Do me a SOLID?

If a programmer comes up to you and says, "Can you do me a SOLID?" Make sure you understand what you're being asked.

Single Responsibility Principle

The first letter, 'S' in the SOLID acronym, stands for Single Responsibility. The principle states:

> *"Every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility."*

Another way to think about this is that a class should have one, and only one, reason to change.

Let's say you have a class that compiles and emails the user a marketing report. What happens if the delivery method changes? What if the user needs to get the report via text? Or on the web? What if the report format changes? Or the report's data source changes? There are many potential reasons for this class to change.

State the class Purpose without AND

A simple way to implement the Single Responsibility Principle is to be able to define what the class does without using the word AND

Let's illustrate this with some code ...

```php
// Class Purpose: Compile AND email a user a marketing report.
class MarketingReport {
  public function execute($reportName, $userEmail)
  {
    $report = $this->compileReport($reportName);
    $this->emailUser($report, $userEmail);
  }
  private function compileReport($reportName) { ... }
  private function emailUser($report, $email) { ... }
}
```

In this case we'd break the class into two classes and push the decision as to what report and who/how to send it up higher in the food chain.

```php
// Class Purpose: Compiles a marketing report.
class MarketingReporter {
  public function compileReport($reportName) { ... }
}

interface ReportNotifierInterface {
  public function sendReport($content, $destination);
}

// Class Purpose: Emails a report to a user
class ReportEmailer implements ReportNotifierInterface {
  public function sendReport($content, $email) { ... }
}
```

Notice how I snuck that interface in there? Aye, I be a slippery bugger with them thar interfaces.

More Robust Classes

By following the Single Responsibility Principle you'll end up with robust classes. Since you're focusing on a single concern, there's less likelihood of any code changes within the class breaking functionality outside the class.

Open/Closed Principle

The second letter in the SOLID acronym stands for the Open/Closed principle. This principle states:

> *"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."*

In other words, once you've coded a class you should never have to change that code except to fix bugs. If additional functionality is needed then the class can be extended.

This principle forces you to think "how could this be changed?" Experience is the best teacher here. You learn to design software setting up patterns that you've commonly used in earlier situations.

Don't Be Too Strict

I've found strict adherence to this principle can result in software that is over-engineered. This occurs when programmers try to think of every potential way a class could be used. A little bit of this thinking is a good thing, but too much can result in more complex classes or groups of classes that have no true need of being so complex.

Sorry if this offends die-hard fans of this principle, but hey, I just calls 'em like I sees 'em. It's a principle, not a law.

Nevertheless, let's work out a common example. Let's say you're working on an account biller, specifically the portion that processes refunds.

```
1  class AccountRefundProcessor {
2    protected $repository;
3
4    public function __construct(AccountRepositoryInterface $repo)
5    {
6      $this->repository = $repo;
7    }
8    public function process()
9    {
10     foreach ($this->repository->getAllAccounts() as $account)
11     {
12       if ($account->isRefundDue())
13       {
14         $this->processSingleRefund($account);
15       }
16     }
17   }
18 }
```

Okay, let's look at the above code. We're using Dependency Injection thus the storage of accounts is separated off into its own repository class. Nice.

Unfortunately, you arrive at work one day and your boss is upset. Apparently, management has decided that accounts due large refunds should have a manual review. Uggh.

Okay, what's wrong with the code above? You're going to have to modify it, so it's not closed to modifications. You could extend it with a subclass, but then you'll have to duplicate most of the `process()` code.

So you set out to refactor the refund processor.

```
interface AccountRefundValidatorInterface {
  public function isValid(Account $account);
}
class AccountRefundDueValidator implements AccountRefundValidatorInterface {
  public function isValid(Account $account)
  {
    return ($account->balance > 0) ? true : false;
  }
}
class AccountRefundReviewedValidator implements
  AccountRefundValidatorInterface {
  public function isValid(Account $account)
  {
    if ($account->balance > 1000)
    {
      return $account->hasBeenReviewed;
    }
    return true;
  }
}
class AccountRefundProcessor {
  protected $repository;
  protected $validators;

  public function __construct(AccountRepositoryInterface $repo,
    array $validators)
  {
    $this->repository = $repo;
    $this->validators = $validators;
  }
  public function process()
  {
    foreach ($this->repository->getAllAccounts() as $account)
    {
      $refundIsValid = true;
```

```
36        foreach ($this->validators as $validator)
37        {
38          $refundIsValid = ($refundIsValid and $validator->isValid($account));
39        }
40        if ($refundIsValid)
41        {
42          $this->processSingleRefund($account);
43        }
44      }
45    }
46  }
```

Now `AccountRefundProcess` takes an array of validators during construction. The next time business rules change you can whip out a new validator, add it to the class construction, and you're golden.

Liskov Substitution Principle

The "L" in SOLID stands for Liskov substitution principle. This principle states:

> *"In a computer program if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may be substituted for objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)."*

Huh? I'd hazard a guess that this part of the SOLID design causes more confusion than just about any other.

It's really all about **Substitutability**, but if we used the 'S' from Substitutability our acronym becomes SOSID instead of SOLID. Can you imagine the conversations?

Programmer 1: "We should follow the S.O.S.I.D. principles when designing classes."

Programmer 2: "Sausage?"

Programmer 1: "No, SOSID."

Programmer 2: "Saucy?"

Programmer 1: Calls up Michael Feathers on the phone ... "Hey, we need a better acronym."

Simply stated, the Liskov Substitution principle means that your program should be able to use sub-types of classes wherever they use a class.

In other words if you have a Rectangle class, and your program uses a Rectangle class and you have a Square class derived from the Rectangle. Then the program should be able to use the Square class anywhere it uses a Rectangle.

Still confused? I know I was confused initially because I wanted to say "Duh? This is pretty obvious", yet I feared there was something I didn't understand. I mean, why have this as a major tenet of SOLID design if it's so obvious?

Turns out there are some subtle aspects to this principle stating preconditions of subtypes should not be strengthened and postconditions should not be weakened. Or is it the other way around? And you can't have subtypes throw exceptions that aren't the same (or derived from) exceptions the supertype throws.

Wow! What else? There are other details about Liskov's principle I won't get into here and I fear I've really spent too much time on it.

Liskov's principle doesn't matter.

What?

Yeah! I said it. (Technically the principle does matter, so I'm lying to you, but I'll try to justify my lie.)

In PHP, if you follow three guidelines then you're covered 99% of the time with Liskov.

1 - Use Interfaces

Use interfaces. Use them everywhere that makes sense. There's not much overhead in adding interfaces and the result is that you have a "pure" level of abstraction that almost guarantees Liskov Substitution is followed.

2 - Keep implementation details out of interfaces

When you're using interfaces, don't have the interface expose any implementation details. Why would you have a `UserAccountInterface` provide any details on storage space left?

3 - Watch your type-checking code.

When you write code that operates differently on certain types, you may be breaking Liskov's Substitution Principle (and probably several other SOLID principles).

Consider the following code:

```
class PluginManager {
  protected $plugins; // array of plugins

  public function add(PluginInterface $plugin)
  {
    if ($plugin instanceof SessionHandler)
    {
      // Only add if user logged on.
      if ( ! Auth::check())
      {
        return;
      }
    }
    $this->plugins[] = $plugin;
  }
}
```

What's incorrect about that piece of code? It breaks Liskov because the task performed is different depending on the object type. This is a common enough snippet of code and in a small application, in the spirit of getting stuff done, I'd be perfectly happy to let it slide.

But ... why should an `add()` method get picky about what's added. If you think about it, `add()`'s being a bit of a control freak. It needs to take a deep breath and give up control. *(Hmmm, Inversion of Control)*

Interface Segregation Principle

The 'I' in SOLID stands for the Interface Segregation Principle. Interface Segregation states:

> *"No client should be forced to depend on methods it does not use."*

In plain English this means don't have bloated interfaces. If you follow the "Single Responsibility Principle" with your interfaces, then this usually isn't an issue.

The app we're designing here will be a small application and this principle probably won't apply too much. Interface Segregation becomes more important the larger your application becomes.

Often this principle is violated with drivers. Let's say you have a cache driver. At it's simplest a cache is really nothing more than temporary storage for a data value. Thus a `save()` or `put()` method would be needed and a corresponding `load()` or `get()` method. But often what happens is the designer of the cache wants it to have extra bells and whistles and will design an interface like:

```
interface CacheInterface {
  public function put($key, $value, $expires);
  public function get($key);
  public function clear($key);
  public function clearAll();
  public function getLastAccess($key);
  public function getNumHits($key);
  public function callBobForSomeCash();
}
```

Let's pretend we implement the cache and realize that the `getLastAccess()` is impossible to implement because your storage doesn't allow it. Likewise, `getNumHits()` is problematic. And we have no clue how to `callBobForSomeCash()`. *Who is this Bob guy? And does he give cash to anyone that calls?* When implementing the interface we decide to just throw exceptions.

```
class StupidCache implements CacheInterface {
  public function put($key, $value, $expires) { ... }
  public function get($key) { ... }
  public function clear($key) { ... }
  public function clearAll() { ... }
  public function getLastAccess($key)
  {
    throw new BadMethodCallException('not implemented');
  }
  public function getNumHits($key)
  {
    throw new BadMethodCallException('not implemented');
  }
  public function callBobForSomeCache()
  {
    throw new BadMethodCallException('not implemented');
  }
}
```

Ugh. Ugly right?

That's the crux of the Interface Segregation Principle. Instead, you should create smaller interfaces such as:

```
interface CacheInterface {
  public function put($key, $value, $expires);
  public function get($key);
  public function clear($key);
  public function clearAll();
}
interface CacheTrackableInterface {
  public function getLastAccess($key);
  public function getNumHits($key);
}
interface CacheFromBobInterface {
  public function callBobForSomeCash();
}
```

Make sense?

Dependency Inversion Principle

The final 'D' in SOLID stands for Dependency Inversion Principle. This states two things:

> *"A. High-level modules should not depend on low-level modules. Both should depend on abstractions."*
>
> *"B. Abstractions should not depend upon details. Details should depend upon abstractions."*

Great so what does this mean?

High-level
: High-level code is usually more complex and relies on the functioning of low-level code.

Low-level
: Low-level code performs basic, focused operations such as accessing the file system or managing a database.

There's a spectrum between low-level and high-level. For example, I consider session management low-level code. Yet session management relies on other low-level code for session storage.

It's useful to think of high-level and low-level in relation to each other. Session management is definitely lower than application specific functionality such as logging a user in, but session management is high-level to the database access layer.

I've heard people say ... "Oh, Dependency Inversion is when you implement Inversion of Control." or "No. Dependency Injection is what Dependency Inversion is." Both answers are partially correct, because both answers can implement Dependency Inversion.

The Dependency Inversion Principle could be better stated as a technique of decoupling class dependencies. By decoupling, both the high-level logic and low-level objects don't rely directly on each other, instead they rely on abstractions.

In the PHP world ... Dependency Inversion is best achieved through what? You guessed it: interfaces.

Isn't it interesting how all these design principles work together? And how often these principles bring into play that most unused of PHP constructs: the interface?

The Simple Rule of Dependency Inversion

Make objects your high-level code uses **always** be through interfaces. And have your low-level code implement those interfaces, and you've nailed this principle.

An Example

User authentication is a good example. At the highest level is the code that authenticates a user.

```
1  <?php
2  interface UserInterface {
3    public function getPassword();
4  }
5
6  interface UserAuthRepositoryInterface {
7    /**
8     * Return UserInterface from the $username
9     */
10   public function fetchByUsername($username);
11 }
12
13 class UserAuth {
14   protected $repository;
15
16   /**
17    * Inject repository dependency
18    */
19   public function __construct(UserAuthRepositoryInterface $repo)
```

```
20    {
21      $this->repository = $repo;
22    }
23
24    /**
25     * Return true if the $username and $password are valid
26     */
27    public function isValid($username, $password)
28    {
29      $user = $this->repository->fetchByUsername($username);
30      if ($user and $user->getPassword() == $password)
31      {
32        return true;
33      }
34      return false;
35    }
36  }
37  ?>
```

So we have the high-level class `UserAuth`, relying on the abstractions of `UserAuthRepositoryInterface` and `UserInterface`. Now, implementing those two abstractions are almost trivial.

```
1   <?php
2   class User extends Eloquent implements UserInterface {
3     public function getPassword()
4     {
5       return $this->password;
6     }
7   }
8   class EloquentUserRepository implements UserAuthRepositoryInterface {
9     public function fetchByUsername($username)
10    {
11      return User::where('username', '=', $username)->first();
12    }
13  }
14  ?>
```

Easy, peasey, lemon-squeezy.

Now, to use the `UserAuth` we would either construct it with the `EloquentUserRepository` or bind the interface to the `EloquentUserRepository` to automatically inject it.

Do not confuse the authentication examples used in this chapter with the Laravel `Auth` facade. These are just examples to illustrate the principle. Laravel's implementation, though similar to these, is far better.

Thanks for checking out the Sample

I hope you enjoyed it.

In fact, I'm hoping you enjoyed it enough to click on over to Getting Stuff Done with Laravel[18] and investing in the rest of the chapters.

[18] https://leanpub.com/gettingstuffdonelaravel