VANQARD PRESS

PHP BRILLIANCE

ADVANCED CODING MOJO

THUNDER RAVEN-STOKER

PHP Brilliance

Advanced Coding Mojo

Thunder Raven-Stoker

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

This version was published on 2016-02-11

Tweet This Book!

Please help Thunder Raven-Stoker by spreading the word about this book on Twitter!

The suggested hashtag for this book is #phpbrilliance.

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=#phpbrilliance

I would like to dedicate this book to the loving memory of my Father, a man who was very much an engineer in the traditional sense. He was also a man who enjoyed ribbing me with the idea that "a programmer doesn't make owt! He just pushes buttons all day."

This one's for you, Dad, with all its incumbent button-pushing.

Contents

PHP Brilliance - Introducting the sampler

Hello and thank you for picking up the sample copy of my book, PHP Brilliance. This sample is brief and to the point, with the intention of showing you the kind of writing that appears within the book itself.

All that I have done here is to cherrypick a couple of chapters to give you a taste of what's in the book itself.

For a more extensive (but still not exhaustive) list of the topics covered, there are a couple of pages added to the back of this sample to give you a clearer idea of what to expect.

At the time of writing, this is still a work in progress and as I say to my readers, I'm hungry for your feedback. Any comments, questions or criticisms that you may have, please do send them in. My goal is to make this book the very best it can be, and I hope to achieve that with your help.

You can write to me directly at this address:

thunder@vanqard.com[1]

Please note, there's no letter "u" after the "q" in vanqard. Just saying.

I hope you enjoy what you read here. And in all things, have fun!

Best wishes, Thunder

[1] thunder@vanqard.com

Prelude

"*Somebody in this camp ain't what he appears to be. Right now that may be one or two of us. By spring, it could be all of us.*" - MacReady

More Pub Time

Before we begin the book proper, there is a key principle that we should look at first. At the time of writing, it is highly unlikely that you have ever heard of this principle before now. I would like to say that it comes to us as the result of some super secret research conducted over many years by a shady, off-the-books, clandestine government agency and that it somehow fell into my hands.

I would very much like to say that, in the interest of the greater good, I am now leaking the details of that principle into the public domain so that it might be read, understood and digested for the benefit of all mankind.

I can do no such things though.

Sadly, the reality of the situation is much more mundane.

I made it up, specifically to act as the unifying theme for the content that appears within these pages.

Introducing the “More Pub Time” principle

If I were to express the principle in a manner that is both short and to the point, it might be expressed thus:

> By pre-emptively acquiring the right knowledge, a programmer may accelerate their own personal development towards mastery and the production of robust, high quality software applications. In turn, common bug-prone scenarios may be avoided and highly valuable pub time preserved. - The More Pub Time Principle[2]

I’m not a big fan of jingoisms though, so let us proceed to break that down into something that makes a great deal more sense.

[2] https://morepubtime.com

What is the right kind of knowledge?

The principle as stated up there leaves itself wide open to being accused of baldly stating the obviously. If a developer has all of the right kind of knowledge in their head, then it should naturally follow that the quality of their application code will be of a high standard.

But what is the right kind of knowledge?

To answer that question, we need to qualify a few things first, including something of a definition for evaluating code quality itself. Fortunately for us, we can stand upon the shoulders of giants and examine the work that has already been done in this arena.

- The functional quality reflects how well a particular application serves its intended purpose.
- The structural quality reflects on the non-functional requirements and how well the application code supports the delivery of the application's functional requirements.

Fitness for purpose, as it is commonly known, isn't something that we will be considering here; building an online word processor that can compete with a desktop version will have very different functional requirements to trying to build out a social network specifically for Alaskan tropical fish enthusiasts.

The functional quality of any given software project will be judged by criteria that are very specific to that project. In other words, does it do what it is supposed to.

However, we certainly can examine the non-functional requirements of any given software project in order to judge the quality of the software and its constituent code. Typically, this is done by assessing four key factors.

Reliability

Can we trust our application code to do the right thing? It might sound a bit daft on the face of it but this is something that we absolutely must be able to do. Our users will rely on our application's code to deliver the functionality that they need, else they will stop being our users and go look elsewhere.

The companies that we work for will rely on our application's code to be able to deliver the necessary functionality either to increase sales, reduce costs or any other way that might improve the bottom line. Failure to provide for this kind of reliance might result in the company going bust or us getting fired.

The keyword here is failure. The reliability of an application, or an individual portion of code inside an application, isn't just down to how well the code responds when given the right data in a low stress environment - coding just for the happy path does not provide us with the reliability that we crave!

No, the reliability of a piece of code or an application as a whole comes down to how well that code responds to errors, failures and defects. If a caching layer is down due to hardware or networking issues, will our application choke and die? Or will it handle the situation gracefully?

If a user provides us with garbage in an HTML form, either innocently or maliciously, does our code validate and sanitise appropriately, chucking up the right kind of error message whilst rejecting the content of that form?

Modelling and assessing for reliability comes in many different flavours, far too many to discuss here, but they all have a common theme: Does the application do the right thing when conditions are favourable and does it still do something acceptable when conditions are unfavourable?

If the application chokes when it is fed with garbage data, if our web site falls over once the database's connection pool becomes saturated, then our quality in terms of reliability could be said to be very low. As we could be said to be out of a job.

Maintainability

I've ranked these four quality considerations in order of importance, which is why maintainability comes in second. Reliability is right there at the top of the list since we need our code to do what it is supposed to, and respond appropriately when it can't.

However, maintainability is a most definite second on the list when considered in order of importance and for very good reason. Whilst the other three members of this list express their focus on a given snapshot of the codebase, frozen in time at any given point, maintainability focusses on both the frozen snapshot and a time-ranged consideration.

This is important for a number of reasons. In an online world, and especially in an online world of applications written for the web using languages such as PHP, the applications themselves very rarely reach a state of completion.

Unlike their counterparts in the desktop world, there isn't a point in an online applications lifetime where the business declares that the product is done and it's time to get the DVDs pressed and the product on shop shelves.

This is especially true for development teams that use agile methodologies and lean processes, tagging and releasing incremental changes over time in order to improve the product offering for the targeted end user.

This is where the time-ranged consideration comes in. Code that is maintainable is code that is resilient to the inevitable changes that come along. It is the opposite of code that is brittle and/or fragile. It is code that is sufficiently malleable that it will bend and flex appropriately to accomodate the seemingly endless flood of change requests emanating from the business team without hiccup, downtime or the loss of that most valuable commodity: Pub time.

Security

Security is perhaps the most commonly misunderstood aspect of software quality. Naturally as developers we are keen to protect our application's data from the more malicious elements of the outside world but security has a much broader sphere of influence beyond the common, though no less critical, act of guarding against known attack vectors.

As far as security is concerned, it becomes necessary to consider the topic in much broader terms. One thing that is common to all applications is that they collect, manipulate and store data. It doesn't matter whether it's a game, a social network, a corporate intranet or a blogging site, it's the data that is king in every case.

As far as the security aspect of software quality is concerned then, we must widen the definition beyond preventing hack attacks to one that recognises the preservation of data integrity is paramount.

If the data within our application is "mostly ok" then we're in trouble since that implies that some of it is bad. We need all of the data in our application to be good data at all times, we need it be where it is supposed to be and not end up in places where it's not supposed to be.

As a result, assessing the security aspect of software quality entails a lot more than making sure we are protecting ourselves against SQL injection attacks, or hashing user passwords properly. It entails making sure we have the right kind of validation in place and that we've put that validation code in the right location. It entails making sure that a multi-stage data manipulation process can be rolled back to its original state should one of those stages fail. It entails a lot of things but the final outcome needs to be this: All of the right data in all of the right places and none of it anywhere else.

Making sure that we're ready for the user that has `;DROP TABLE users;@hotmail.com` as an email address is one thing, but if a member of our team creates an algorithm that accidentally sets every user's first name to the four character string "John" then we're already sailing down crap creek anyway and the paddle is nowhere to be seen.

Efficiency

I've known developers in the past who have placed an inordinately high degree of importance on creating highly efficient, fast running, low memory usage code. Ones that have declared code can only be beautiful when it's fast.

Thankfully, most of us already know better.

In the days, months and years before the explosion of cloud computing and virtualisation, the performance of an application could often be improved by an activity known as "throwing more tin at it". Adding another server to the rack or installing fast hard disks or more RAM was commonly cheaper than paying a developer's salary for the time required to optimise inefficient code.

Nowadays, we can still "throw more tin at it" but in a virtual sense by spinning up another instance.

Nevertheless, creating code that is performant and efficient is still very important if we hope to preserve our all-important pub time. Code that is already reliable, secure and maintainable can always be optimised for speed and efficiency.

But code that is written for speed first can rarely be optimised for reliability, security and maintainability after the fact.

That's not to say that speed and efficiency aren't important of course. An online application that takes thirty-seven minutes to turn a request into a response had

better be delivering absolutely critical information at the end of the cycle if it has even the slightest hope of retaining users.

The reality for the vast majority of online applications is that the request-response cycle needs to complete in a matter of seconds, not minutes. Fortunately for us in the online world, there are a number of techniques that we can employ in order to improve the *perceived* performance of an application and therefore improve the user experience.

Rather than despatching that account activation email as part of the sign-up procedure, we can queue it for an independent process such as a cron or a daemon to take care of.

Rather than parsing and processing an uploaded CSV as part of the request-response cycle, we can move the file to a watched directory and allow the status of the file's processing to be reported back to the front end by a series of subsequent, asynchronous calls.

Of course, we can optimise the actual performance and efficiency of our application code and not just the perceived performance and efficiency. The developer that performs a `SELECT * FROM users` and then loops through the entire recordset in memory just to find the row for the user that has just logged in probably needs his or her butt kicked first and foremost before being taught how to write a `WHERE` clause. Returning a single row in this situation is clearly much more efficient than pulling out the entire table.

Despite being at the bottom of this list, efficiency is still a very important factor in a list of four factors for assessing the quality of an application and its code.

> Software quality is a huge topic in its own right, with hundreds of books and academic papers published on the subject. Here, I've only just scratched the surface and only very lightly at that. Nevertheless, I hope I've been able to at least impart the vaguest impression of what is meant by quality code being reliable, maintainable, secure and efficient.

Which brings us closer to considering what the right kind of knowledge might be. Before we do that though, we should take a moment to consider the wrong kind of

knowledge.

Hello World!

The learning process is well documented. Not just in programming circles but in any kind of trade or craft, a practitioner will go through a number of distinctive phases. At the start comes the novice, picking up the fundamentals and learning the first principles. It might be horribly presumptuous of me to say this but I'd be prepared to wager that a great many of us began this journey by writing "Hello World" on the screen, all thanks to Brian Kernighan way back in 1973.

Advancing beyond the novice stage is the journeyman, building upon the fundamentals by acquiring additional knowledge and combining it with experience in order to hone their skills and further their personal development before readying themselves for the final stage, that of the master craftsman.

Traditionally, the title of master craftsman would be bestowed upon a practitioner by the guild that oversees their particular trade or craft. When ready, the journeyman would prepare and submit a piece of work as part of the application process and it would be the members of the guild who would judge said piece of work and establish whether they deem it to be worthy or not.

In the world of online application creation we aren't so fortunate to have such a structured approach to personal development; one cannot readily apprentice oneself to an established master craftsman. As such, our journeyman stage is less a joy-filled skip through a sun-drenched meadow and much more a stagger through a perilous swamp filled with traps and pitfalls.

It's also huge, thanks to the Internet. The journeyman developer is literally inundated with opportunities to advance their knowledge and abilities thanks to the plethora of materials available to him or her. There are books aplenty, there are courses to follow both online and in more traditional bricks-and-mortar learning establishments, there are articles and blog posts that are way to numerous to count.

Which is precisely where the danger lies; the largely unrestrained and unregulated publication of tutorials and training videos presents a very real danger to the journeyman coder. With such an unfathomably large volume of information available, how does the journeyman coder ever hope to be able to sift the good quality information from the bad, especially when the bad keeps coming at an ever-increasing rate?

As a case in point, as recently as December 2015 I encountered two exceedingly poor quality tutorials linked to in developer forums that I frequent. The first showed how to "easily populate variables using the `extract()` function", which is a particularly dangerous "trick" to learn without making the reader fully aware of the dangers involved.

However, even that pales into insignificance when compared to the second tutorial, which illustrated how to use AJAX to send search terms from an html form to a PHP backend that would return the search results back to the front end after querying the database. On the face of it, this seems an entirely reasonable thing to post a tutorial about and as a topic, it has certainly been covered any number of times before.

However, in this particular tutorial, the example code illustrated the creation of a SQL query by iterating over the `$_POST` superglobal and using the values directly and without any form of validation or sanitisation.

For any time-served developer this should be a horrifying state of affairs; that a tutorial posted in 2015 pays no heed to preventing SQL injection attacks and thus proving the proverb "A little knowledge is a dangerous thing".

Yet it is easy to forget that new coders take their very first steps onto a coding career path every single day. Any such hapless coder copying and pasting this tutorial code into their own endeavours would become vulnerable to those same SQL injection attacks, at least until such time that they acquire the right kind of knowledge that would let them guard against these attacks.

Which brings us nicely back around to considering just what does constitute the right knowledge. Regrettably, at the time of writing there isn't one definitive answer to that question. It is also rather dubious that there ever could be.

One of the very great joys that a career in software development brings along with it is the fact that the learning process never ends. No matter how advanced a developer becomes in their mastery of the topic, there is always more to learn. On the one hand, this is because the topic itself is so vast. On the other, it is because the technology progresses relentlessly.

Complete mastery of application software development is entirely unobtainable for these two distinct reasons. At any given moment, there is already too much to fit into a single human brain. And today's latest technique has every chance of being superseded tomorrow.

Despite this, it is certainly my strongly held belief that there is certainly a catalogue of knowledge that every Senior Developer and above needs to have tucked under their belt. Such a catalogue of knowledge can cover a variety of topics, from very specific morsels such as always ensuring that you escape, validate and sanitise your inputs appropriately to much broader concepts such as the key software development principles, rules, laws and even design patterns.

But why?

The reason itself can quite simply be stated as this; to build better software. More specifically, to build better software expressly in terms of its structural quality so that it is reliable, maintainable, secure and efficient.

Fantastic! So now that we've nailed down the definitions of software quality and the right knowledge needed to achieve software quality…

What the heck is Pub Time?

If we are to preserve our invaluable pub time by pre-emptively learning the right stuff in order to achieve high quality software, we need to know what the chuffing heck is pub time?

This is the more jocular aspect of *The More Pub Time Principle*, which is precisely why it'll never be accepted by academicians as a valid software principle but in brief it goes like this:

> Pub time is that time expended in any manner more conducive to a happier and more fulfilling life than staying behind in the office hunting down and fixing software bugs that were entirely avoidable in the first place.

For me, that quite often entails a trip to pub with my friends to have a catch-up over a drink but it isn't, by definition, limited to going to the pub. Pub time can entail taking your beloved out for a romantic meal. Or it can mean getting home in time to read bedtime stories to the little ones. Or going to watch a play. Or hitting the gym for a workout. Or firing up your games console in order to blow the heads off of alien invaders.

In fact, engaging in any activity that improves the quality of your life and its enjoyment can be considered as "pub time", in comparison to late nights wasted hunting down and fixing perfectly avoidable software defects.

Conclusion

The journeyman stage of a software developer's career is somewhat ill-defined, and this is certainly all the more true for the developer working in the online world.

Much of the web is built upon open source languages such as PHP, Ruby and Python to name but three. Each of these languages have a dedicated following and are supported by fantastic communities and as such, there is a tremendous wealth of learning resources available both online and off.

One thing that the Internet will never be able to achieve though is the elimination of the journeyman stage. It simply is not possible for a learner, of any craft or topic, to jump straight from novice to master without having spent time in the journeyman phase. This isn't a limitation of technology that might be addressed in the future, it's a limitation of the human mind. Knowledge acquisition is done through the process of learning, leaving us at the mercy of our current biological limitations.

What we absolutely can do though is shorten the path that the journeyman takes. When faced with the boggy swamp that separates the land of the novice from the land of the masters, we can pick out a sure, swift and safe path to follow - one that takes us to all the good places to visit and leads us safely past the traps and the pitfalls.

This is entirely what *The More Pub Time Principle* aims to do in providing its rather nebulous definition:

> By pre-emptively acquiring the right knowledge, a programmer may accelerate their own personal development towards mastery and the production of robust, high quality software applications. In turn, common bug-prone scenarios may be avoided and highly valuable pub time preserved.

There are already a number of initiatives currently in play intending to help the journeyman developer find the faster, safer route through the quagmire. One

rather excellent resource is the *PHP: The Right Way*[3] project, which I would highly recommend to PHP developers at every level. Another resource is this book.

The goal of PHP Brilliance as a book is to put all of the right kinds of information in front of you. From architectural concerns such as MVC vs Service Oriented Architecture vs Microservices to a thorough examination of the SOLID principles, from looking at abstraction and inheritance in a new light to considering the frankenstein method of utilising closures.

The goal of PHP Brilliance as a state of mind is to ensure that you are equipped to lead the team in creating beautiful code that is reliable, maintainable, secure and efficient.

Life is short so please do take this advice to heart: Preserve your all-too-precious pub time, and do enjoy the journey along the way.

Last but by no means least please remember; drink responsibly.

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

Foundations

"*I dunno what the hell's in there, but it's weird and pissed off, whatever it is.*" - Clark

Object Oriented Thinking

Before we can get onto the really meaty stuff, it's important for us to take a pause and check on our understanding of what object oriented programming is. Even though this isn't a book for those first starting out in their development career, it becomes all to easy to form a fixed opinion of OO stuff when we first come across the idea as developers.

Further to this, there's a plethora of online tutorials that all take pretty much the same approach to explaining it and, as such, these tend to propagate some rather fixed thinking in this area.

If we're going to build our palaces and castles in beautiful and elegant PHP code, we must turn our attention first to what we are intending to build upon. In order to raise glorious edifices of logical magnificence that not just blend into the world wide web's skyline but to actually be part of what defines that skyline, we must focus on the groundwork.

Our applications need to not only suffer the slings and arrows of outrageous fortune but also stand firm against the whims and change requests of our business teams and product managers. These change requests are the earthquakes and floods that our application development must withstand. We know that they're going to come. We can brace ourselves effectively against the flood. Just so long as our foundations are rock solid.

So let's go back to the basics and examine what we already know.

All too frequently, a tutorial will take the notion of an object as being a representation of a real world "thing" and how the developer is supposed to hang on to this notion as the author goes on to explain how real world things have a particular set of characteristics and attributes that go on to define what the thing is and what it does.

The benefit of this approach is that the examples given are already familiar to the reader and as such allows him or her to connect the concepts with current knowledge and experience. Anyone setting out to learn object oriented PHP will know what a car is. Or that a dog is a type of animal. For anyone approaching object oriented

development from a procedural background, something that is certainly prevalent in the PHP arena, this relationship between code and real-world objects can help the developer reach that "penny drop" moment sooner; that point where he or she will suddenly "get it".

The danger here though, and it's a pitfall that many of us have fallen into, is that the developer starts to cling on to this idea of linking the objects they create with real world examples. The next project that they take on, they'll starting hunting down the "nouns" in the project brief and planning their objects around them. This here is a `User`, that over there is a `Product`. It's a perfectly valid start to the process of identifying and designing the objects that will be the key players in our new application. But it is only a start. Unfortunately, that is commonly where the tutorials end.

If you're going to be guiding and mentoring the more junior members of your team, you're going to see some quite iffy code along the way. Just to make sure that we're on the same page, so to speak, I would like to set out the path that we're going to take in order to reach *object oriented thinking*. It's starts with a shiny new junior, a likeable chap that we'll call Joe. The route that Joe has taken through the PHP learning landscape in order to arrive at our office OO ready is not an uncommon one. I'd like to say that it's entirely fictional but that wouldn't be quite true. You see, Joe's path was actually the path that I took albeit with some hearty doses of artistic license added here and there. I have no shame.

From the outset, Joe learnt to script in PHP; building out the pages of the sites that he built with one script for each. Here an `index.php`, there an `aboutus.php`. Things such as database access and variable assignments could be done at the top of the file, then down below the page itself is built up with html and peppered with inline php constructs. At the top, the program logic, at the bottom, the output.

After a while, Joe's realised that he's duplicating significant amounts of code across his scripts. This is the point where he starts breaking chunks out into separate files; header, footer, routines for accessing the database, others for building html tables. This of course is all in accordance with the tutorials that he has been following regarding the use of the `include` and `require` functions.

Before long, he's creating libraries of commonly used functions which he can port from one project to another. Big old PHP files with names such as `database.php`,

`html.php` and other such collections of useful functions gathered together in a single file.

What happens then when Joe starts reading up about object oriented development? He's introduced to a `Car` class that not only has properties for things like the wheels and the engine, but also methods (functions!) for when the car needs to do things like `move()`, `turn()` and `stop()`.

Joe thinks it'd be a great idea to wrap his carefully crafted library of functions in `class` statements. This now is the point where Joe could take one of two paths. Does he instantiate a `Database` object in order to use those transplanted library methods? Or does he add the `static` keyword to the function declarations so that he can call the class methods statically?

Well, instantiating the object doesn't really look to be terribly useful. Let's go with the static.

Now Joe's coding regularly features things like this:

```
<?php
include("Config.php");
include("Database.php");
$conn = Database::connect($dbname, $dbuser, $dbpass);
```

Youch.

Eventually our developer will make the transition from wrapping his function libraries in class statements to identifying the nouns in his system and building objects around those. This is very much in line with the tutorials that he has followed. What we see next from Joe is the predominant but natural outcome of those tutorials and their habit of finishing a topic early.

Joe's classes have become enormous. The classes at the centre of the application, whether this be a `User` class or a `Product` class, are truly huge, spanning thousands of lines of code and with methods so long the start and end of them cannot be viewed onscreen at the same time. Scroll, scroll, scroll.

What's going on here?

The developer has fallen back on his procedural code knowledge once more in order to code up object methods that, rather than performing a single function, run through

an entire process from start to finish. Perhaps the most obvious example of this is an object method, probably located in a class called 'User' and most likely named something along the lines of 'create' or 'register'.

I'll grant you that for many web applications, the user registration process can be a convoluted one, performing validation against a number of submitted form fields, creating a user record along with the login credentials, possibly also storing an address and linking the newly created user to it as well as hooking up any number of configuration settings. What has happened here is that the developer has taken a procedural process and simply transplanted it into an object method. What used to live as a single php page for receiving and processing a user registration has now been transplanted wholesale into a single method in the `User` class. Not a literal copy and paste operation you understand, but a selective extraction and remodelling of the code to squish it in between those opening and closing braces.

Joe starts by validating the parameters that were passed in to the register() method. If that's all fine and dandy, he moves onto performing the database ops necessary to get the data into storage and extract the ids. This part may result in just one query being run, or it could be many; the basic user details could be accompanied by a row of default preference settings in one table, a physical home address in another. If that all proceeds ok, Joe then sends out the welcome-cum-verify email before finally returning a `true` or `false` back to the original invoker of the method.

In just that one method, we have a minimum of three fracture points - places where the process can fail - leading to a brittle design that can fall over a number of ways and be difficult to maintain at the same time. The validation stage, the database stage and the email sending stage.

Now is a good time to introduce a key principle to object oriented thinking. I don't recall where I first encountered this one but it has stayed with me ever since. It goes like this: *An object should either know things or do things, but never both.*

So many of the applications that we build are going to have a `User` model object. If such a model object represents what we know about a particular user, and we know that user registration is a *process* then it naturally follows that our `User` model class cannot have a `register()` method.

The more that you think about this, the more it makes sense. After all, why should all of our instances of the `User` class be lugging around a method whose purpose is

to create the user record in the first place. When would such an instance have need of the `register()` method again?

If you were to take that *knows things or does things* principle and apply it to the model layer of your most recent application, how many model classes would it suggest that you change? How many of the entities in your model both know the details of the thing that it represents (i.e. hold the data for) and provides ways to manipulate that data beyond the act of setting and getting it?

In most cases, the primary residents of our model layer will be objects that represent the data that lives inside our application. In this sense, these are the objects that *know* things. For instance, suppose we have an application that's going to be handling lots of `User` instances. We ought to be confident that each instance *knows* the name, date of birth and email address of the `User` that it represents. In all likelihood, a full blown application will have `User` models that hold a lot more detail than that but this will serve us as a good starting point for the time being. None of these instances should be holding methods that *go beyond* managing the individual pieces of data that they represent. The methods of our model classes should be entirely introspective. Setters and getters are naturally of this ilk but what about the methods that we can identify as being processors?

What do I mean by processors?

Processors are methods that do things. A method that validates user input is a processor. A method that triggers the sending of an email is a processor. In almost every case, unless a processor is specifically introspective, it can be moved out into a new object that's designed to handle, to *encapsulate* that process.

For the registration procedure, ideally what we are looking for is a whole range of objects all collaborating in the user account creation process. Each object will have a tightly defined area of focus, performing a single task and performing it well. Having each tiny piece operating as a part of the whole is our goal here. We're looking for a range of validator objects responsible for checking each part (a password validator can confirm that the offered password has the right number and range of characters, a date validator can confirm that a submitted date of birth is in the right format, and perhaps importantly, is within the correct range (over 18s only?).

When we take this approach, we're neatly separating the logic that performs validation away from the logic that performs record creation. Continuing in an

ideal fashion, our process for record creation should be nicely squirrelled away and separate from the objects that represent those data records in the first place.

One thing that you may very well notice in this book is how often I'll draw your attention to apparently circular references. The chapter on inheritance refers you forward to the chapter on the Liskov Substitution Principle. The chapter on the Liskov Substitution Principle refers you back to the chapter on Inheritance and also sideways to the part on *favouring composition over inheritance*. There are so many ways in which one topic will either rely on or reinforce another that the boundaries start to blur. Wherever the crossovers occur, I will endeavour to point them out to you.

Now that you've just read that aside (you did, didn't you?), I'm going to make my first mention of the Single Responsibility Principle. The Single Responsibility Principle is, to my mind, the absolute single most important one of the five principles that go in to make up the set of SOLID principles. It also pleases me greatly that it's the first one in the set. Familiarity with the SRP can only help to reinforce the idea that *our objects should either know things or do things, but never both.* If we have objects in our system that know things and do things at the same time, it's a reasonably safe bet that we're already violating the Single Responsibility Principle. When we get to that chapter, I hope to make it clear as to why this will be.

Returning to Joe then, we know that his tutorials taught him to build his objects based around the *nouns* of his system. We also know that those self same beginner tutorials didn't tell him when to stop adding methods to his objects. The good news is though that we're now in a much better position to enlighten him as to when he's putting too much into a single class.

Regrettably it's not so easy to draw a line between the knowing and the doing. Adding processors to an object that's only supposed to be knowing things is all too easy to do. Worse still, it usually begins with the tiniest little thing and before you know it, the slow but inexorable creep towards bloated classes has begun. How then are you supposed to watch out for this, outside of an all-out code audit?

Taking a finger in the air approach, you should start to feel uncomfortable whenever any of the following signs appear in an object method.

Conditional statements such as an if statement or a switch appear in a method, and those conditionals are not used in performing validation but are selecting different logic paths to follow based on a incoming parameter. Try to restrict the use of 'if' statements to validation only. In the event that you're creating branched processes in your code because of the value of a particular property, you're almost certainly going to be better served by creating an independent object for each branched process and utilising something like the Strategy pattern, or the Chain of Responsibility pattern in order to handle the processing.

You can't see the start and end of a particular method at the same time. If a single method occupies more than a single screenful in your editor of choice, you have a problem. Look carefully at those methods to see if you can't at least break them down - the chances are good that they're doing more than one task. As a general rule of thumb, I'd suggest ensuring that your methods contain no more than twenty lines of active code.

There are lots of comments *inside* your methods. Nicely documented code is a good thing, but if you find a method that feels the need to explain every step it's taking, it's either taking too many steps or the author thinks you're a numpty. The best methods are nice and short, with easy to follow code and a terse but helpful explanation of the intent in the docblock above it.

These are the types of things that we need to be looking out for when we're reviewing the code of our more junior team members, and indeed the code that we produce ourselves. Detecting code smells is a knack that comes with both knowledge and experience but just by being aware of these three things, you're already well on your way. Nevertheless, code smells are certainly rife in PHP. Somehow it just seems to be something that we in the PHP community have grown up with, although of course there are plenty of examples to be found in other languages too. Even so, they are certainly something that we need to guard against. Much of the advice in the upcoming chapters is geared towards not such much how to avoid code smells, but how to take the right approach to creating an application whose objects don't stink.

Martin Fowler has an excellent bliki post[4] on code smells which succinctly explains what they are but in doing so he makes mention of anaemic objects that might benefit from having behaviours added to them. I'm not in full agreement with the brevity of this post since I really am very keen to put forward the notion that any single

[4] http://martinfowler.com/bliki/CodeSmell.html

class, and the objects that are instantiated from them, should have a laser-focussed intent and purpose. If you're interested, you can find a list of the more common code smells online, but I would actually be keen to suggest that you look them up after we're done with the first part of the book. Reading about them afterwards is much more likely to reinforce what you will have already read at that point.

Anyway, let's bring this back on topic. If we were to continue along the path of tightening the focus of our imaginary `User` class, what questions should we be asking? We've already considered the possibility of removing the `register()` method since we've determined that it doesn't belong within instances of our `User` class. How about password handling? This is perhaps the second most prevalent *wrong thing* to be found within a user model. For sure, we may want to accept and hold the hashed value of a user's password but do we actually want to incorporate the hashing mechanism within the user class itself?

The immediate answer seems to be yes, since it's something that we'll be doing *only* in conjunction with the user's own data. Nevertheless, we need to consider all of the things that we might want to do with passwords. For starters, we'll need to be able to accept a password from the user *when they register* for an account, which then needs to be hashed appropriately. Obviously. we will also need to be able to check a password when they log in, generating a hash of of the password that they've given us and checking it against the hash that we've already stored. Already, we have two processors for the most basic of operations.

Experience tells us that we are also going to have to provide some sort of password reset mechanism, since some of our users are likely to fall squarely into the *can't remember passwords for toffee* camp. Do we also need to implement a lock-out mechanism after three failed login attempts?

Answering these questions leads us to the conclusion that actually, building password handling logic into our `User` class is maybe not such a good idea afterall. Instead, we can wrap up all of these password related methods inside a new `PasswordManager` class, instances of which can either be injected into our `User` instances at creation time, or lazily loaded on request dependent upon our appetite for tight/loose coupling between a user instance and a password management processor.

Simply by hiving off two very common processes, that of registration and password management, we've not only improved the focus of the `User` class dramatically, we have also created two additional classes with tightly defined areas of responsibility.

That in itself is as awesome as a large and tasty pint of ice cold beer. Well, maybe almost as awesome - nothing really comes close to a large and tasty pint of ice cold beer. Ever.

So where does this leave us now? All being well, we have progressed from the stage where the most basic tutorials leave off. We are now a little better equipped to guard against thinking of our primary classes as silos for the ever expanding lists of processor methods that our application appears to need.

This is rather the key point that I want to make at this stage. All too often, it's terribly easy to get stuck with the real world nouns idea when thinking about the objects that will come into play within our applications, when what we really need to develop is an ability to think of application objects in an abstract sense. There isn't a tangible real world equivalent for a `PasswordManager` but if we can successfully keep that notion of objects only being able to know things or do things at the forefront, we're a long way down the trail of instinctively knowing what should go where.

Summary

For our first chapter, I've rather concentrated on the idea of grouping our application's objects into two distinct camps; the knowers and the doers. This is very much the key theme that I would like to introduce at this point. In a very general sense, our knowers are likely to be the principle citizens that reside in our model layer. They hold and represent the data that lies at the heart of our application. These are the users, the products, the orders and the invoices. They present an interface which is designed to set, retrieve, manipulate or transform the individual elements of data that they are responsible for.

Then there are the doers. The objects within our system that cause things to happen and whose interfaces comprise methods that we can call in order to trigger those things. These doers might mask the simplest of processes, such as the hashing of a password, or they might be a facade onto much more complex procedures, governing the various stages of user registration for example.

Clearly then, I'm really quite keen on this idea. Largely because I've witnessed the positive effects that it can have. It's not so much the idea in itself per se, more that the end results speak for themselves. Smaller, tighter, leaner and meaner objects are so much more efficient and maintainable than the alternative: classes treated like silos

into which we've dumped great quantities of superficially related methods almost as if we considered the class name to be little more than a namespace for a coagulated library of code.

Don't Talk To Strangers

In the previous chapter, we started to look at how even the accidental implementation of *couriers* will, in all likelihood, make portions of your application fragile and susceptible to bugs. The example we looked at passed a database connection through a controller in order to reach its final destination inside a model instance.

The issue wasn't that the code doesn't work, or that it breaks occasionally (although it might if you haven't wrapped the establishment of a database connection with the appropriate safeguards and error-handling). No, the issue is the fact that when the target code, in this case the model class, is revisited at some point down the line, you also have a string of other classes to consider if the reception of the database connection changes in any significant way.

It's very much a case of "*I fixed this bit here, but now that bit over there is broken!*"

This chapter focuses on a very similar problem and concerns itself with something known by many names, one of which is the *Law of Demeter*. Devised by Ian Holland way back in 1987, the *Law of Demeter* isn't even a law, but a guideline. I suppose we can forgive the fine folks that came up with that particular moniker though, since "The Guideline of Demeter" is somehow less *punchy*.

Emerson Macedo[5] boils down the *Law of Demeter* into three succinct statements, which are thus:

1. Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
2. Each unit should only talk to its friends; don't talk to strangers.
3. Only talk to your immediate friends.

In a way though, these three statements are a little *too much* on the succinct side. Without appropriate context or prior knowledge of this particular guideline, it isn't particularly easy to intuit either their meaning or their intended application.

But hey, that's what this chapter is for. Let's get going.

[5] http://emerleite.com/

The Principle of Least Knowledge

That sub-heading up there gives us another one of the names that this particular guideline is known by and at least this time it's a little more indicative of what this topic is actually about. This new name for it ties in nicely with those three bullet points that we looked at on the first page of this chapter.

The idea behind *the Principle of Least Knowledge* is that wherever you happen to be coding inside your application, the code that you write should only express knowledge of its immediate surroundings. When you follow this particular guideline, you're automatically promoting the notion of *loose coupling* within your codebase, which in turn leads to a much higher degree of *maintainability*.

What on earth does this all mean though?

Clearly, we need one of those examples around about now.

```
class OrderController
{
    public function addOrderAction()
    {
        ...
        try {
            $user->getAccount()
                 ->getBalance()
                 ->deductAmount($orderTotal);
        } catch (InsufficientFundsException $e) {
            // handle exception
        }
        ...
}
```

Here we are again with that woefully used and abused `addOrderAction` method, looking for a simple way for our customer to pay their damned bill.

In this case, it looks like we are passing the value of the order through to a method called `deductAmount()`, catching the exception that might be thrown if the user doesn't have enough funds in their account to pay for the order.

This is fine, surely? We've got error handling in there, and even if you can't see it in such a small snippet of code, there's lazy loading going on in the background with the `UserAccount` and `AccountBalance` objects.

Lizzie's certainly familiar with this sort of approach; they did it all the time at her old place and the old place managed to turn out some reasonably useful applications. The latter doesn't change the fact that Lizzie's last place was rubbish though.

Don't tell her I said that, we're lucky to have her here with us.

Back to the code. Just what is the problem with it, given the fact that it works when all is well, and catches the exception when they're not? The answer to that lies not with the fact that it's perfectly operational today, but that it presents us with a potential problem in the future.

Nothing will impact so negatively upon our valuable pub time than code constructs that provide fertile ground for defects to geminate within.

With the preservation of pub time firmly set at the forefront of our minds, it's time to look at yet another name that this particular principle is known by.

The one dot principle.

The first thing to note here is that this particular name was devised for those languages that separate objects from their fields with the dot (period) symbol, which in real terms, is pretty much all of them *except* PHP and Perl.

You know the sort of thing. If you've ever taken even the briefest look at Java, you will have encountered the following line:

```
System.out.println("Hello, World");
```

Of course, in PHP we use the arrow (->) to separate an object from its fields but the idea is just the same; if you're utilising more than one dot to access a property or method, the chances are exceedingly high that you're not following the *Principle of Least Knowledge* correctly.

The second thing to note is that this particular version of the name for the guideline/idiom/principle is the least accurate of the bunch so far. Determining whether we are conforming to or violating the *Law of Demeter* is a little bit more involved than a process of counting the dots.

Still, we haven't even gotten as far as considering why it's a bad thing, so let's get straight onto that now by recalling the offending "line" of code.

```
$user->getAccount()
     ->getBalance()
     ->deductAmount($orderTotal);
```

In PHP terms, that's three "dots" right there…

Could we fix that code by adhering to a "single dot" approach like this?

```
$account = $user->getAccount();
$balance = $account->getBalance();
$balance->deductAmount($orderTotal);
```

The answer is emphatically no.

All that we have achieved here is to move each of those three troublesome "dots" onto their own lines. The key problem remains, there's just too much knowledge here.

Not only is it aware that invoking the `getAccount()` method will return an object that presents the `getBalance()` method, it's also aware that the return value of the `getBalance()` method offers the `deductAmount()` method.

That's just too much knowledge.

In the language of the Law of Demeter, this line of code is reaching through the intermediate objects in order to invoke the `deductAmount()` method at the end of the chain. The way that we've written that code, even after we decomposed the chain down to three individual statements, expresses intimate knowledge of how that chain is structured all the way through to the final method call.

What we have here is another instance of *tight coupling*, which is something that we should always be trying to avoid. Tight coupling reduces the quality of the application code that we write by making it harder to maintain. Changes made to one piece of tightly coupled code require us to review and potentially modify all the other members of the tightly coupled relationship.

To put this into perspective, imagine a situation where, six months down the line, the business team pops up with a new requirement: instead of requiring our users to pre-fund their accounts, we'll be offering credit facilities to a select few of them.

Just to make things a little more complicated, let's imagine that the pre-funders are going to get to enjoy a discount on their orders but the users that pay on credit are going to be subject to a small surcharge.

This is a significant change for the business, but the way things are currently, it's massive change for the codebase, all thanks due to our tightly coupled design.

When this particular story lands in Lizzie's lap, it's apparent that she will be working primarily within the `UserAccount` class to handle the credit versus pre-funded account scenarios.

Unfortunately, thanks to our tightly coupled code, she's also going to have to check through the entire codebase to determine whether other parts of the application are affected as well. For one thing, we know that the `addOrderAction()` method will have to be modified since it most certainly doesn't accommodate the notion of juggling credit facilities currently.

What if the company offered a monthly subscription service? A notion that's certainly not beyond the realms of possibility but if the answer is to be a yes, then poor Lizzie's faced with the prospect of digging through the cron scripts as well. We'd better make sure there's plenty of coffee in the kitchen in this case.

Had we been aware of the Law of Demeter from the outset, all of this extra work might have been avoided. Instead of having our action method reach through multiple objects in order to trigger an order payment, our controller code might have looked like this.

```
class OrderController
{
    public function addOrderAction()
    {
        ...
        try {
            $user->payForOrder($order);
        } catch (OrderPaymentException $e) {
            // handle error
        }
        ...
    }
}
```

The critical line of code is this one:

```
$user->payForOrder($order);
```

In one fell swoop we've eliminated all of that extra knowledge that the action method shouldn't have had in the first place. Even though I've left you to imagine how this method acquires the `$user` and `$order` instances in the first place, the simple act of passing the `$order` instance to the user's `payForOrder()` method would have served the pre-credit account era just as well as the post-credit account one.

In other words, knowledge of the inner workings of the order payment process has been removed from where it doesn't belong. Our action method no longer has any kind of idea as to what happens to that `Order` instance on the other side of the `payForOrder()` invocation.

Now, the only knowledge it has of that process is that if an exception is thrown, it has to deal with it, which is exactly as it should be.

> If you've already encountered the *Law of Demeter* previously, you might have noticed how there are some distinct parallels between the example that we've been working through here and the subject of a paper by David Bock entitled The Paperboy, The Wallet and The Law of Demeter[a] in which he describes the process of paying the paperboy not by having the customer hand over his wallet, but by having the paperboy request payment and allowing the customer to organise how the bill gets settled. In other words, the paperboy has no need of ever seeing the wallet or even to know that it exists in the first place. I heartily recommend reading this paper. It's quite short and beautifully illustrates the point.
>
> [a] http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf

But does this mean that the logic that handles order payments now has to be moved into the `User` class?

Not at all. Given the code that we've already seen, and the nature of the story that Lizzie has to deal with, we know that it's the `UserAccount` objects that are going to be dealing with order payments.

What we're adding to the `User` class is a proxy method, like so:

```
class User
{
    ...
    public function payForOrder(Chargeable $order)
    {
        try {
                $this->getAccount()->payForOrder($order);
        } catch (InsuffientFundsException $e) {
            // throw new OrderPaymentException();
        }
    }
}
```

> For the sake of the *One Dot Principle*, we are safe to exclude the arrow that comes after any references to `$this`. I'll come to that bit shortly.

A proxy method is a pretty simple construct. In this instance it does just two things. Firstly, it makes sure that it's receiving a parameter that conforms to our `Chargeable` interface, whatever that may be. Secondly, it passes that `Chargeable` item on to this user's `UserAccount` instance.

> Remember that I mentioned that the `UserAccount` instance would be lazily loaded? If this is the case, then acquiring it via a `getAccount()` method is a sensible way to achieve this. The alternative would be to inject a pre-fabricated `UserAccount` instance via the constructor, in which case we would probably access it as `$this->account` instead.

This is the point where we start encountering the downsides to achieving loose coupling by following the *Law of Demeter*; a proliferation of proxy methods.

In this rather simple scenario, it isn't really a problem. The addition of the `payForOrder()` method to the public interface of our `User` model is both short and to the point. Any other developer on our team ought to be able to look at that code as

see that an order can be paid for just by sending `Chargeable` compliant instance to the `payForOrder()` method.

Where things do start to get a bit messy is the point where we are adding too many proxy methods. It doesn't take too long before we end up polluting the public interface of our `User` model with methods that are only vaguely related to our concept of what a `User` actually is. With too much of this going on, our `User` model will end up looking like an implementation of the *Facade Pattern*.

We're getting ahead of our selves though. If we didn't want to start going down the route of adding proxy methods, how do we solve the issue in our controller's action method.

One possibility would be to cut out the middleman and work with the `UserAccount` instance directly, given us code that looks something like this:

```
public function addOrderAction()
{
    ...
    try {
      $userAccount->payForOrder($order);
    } catch ( OrderPaymentException $e) {
      // handle exception
    }
    ...
}
```

If you read that aside earlier on in this chapter you might be tempted to exclaim "Aha! But isn't that just like handing the wallet over to the paperboy?" to which the answer is, thankfully, no. If our action method is behaving like the paperboy, it's still successfully ignorant of the actual payment process, which is still nicely abstracted away behind the `payForOrder()` method.

What we're actually looking at here is the action method working directly with an immediate collaborator. Admittedly, our controller might be better served by handing off the entire order/payment process to a service and letting that service co-ordinate the appropriate interactions with the model, but that's an architectural concern that we'll look at in *Part Five* of the book.

Connecting code with its immediate collaborators is exactly what the *Law of Demeter* wishes for us to do. Let's take another look at those three bullet points that we started this chapter with.

1. Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
2. Each unit should only talk to its friends; don't talk to strangers.
3. Only talk to your immediate friends.

By revising our code to cut out the middle man (the `User` instance), we've brought our three "friends" closer together; the `Order` and `UserAccount` instances, and the `OrderController` itself. Even so, we're still promoting loose coupling between the players within this system; the knowledge of how to handle a `Chargeable` item remains firmly where it should be and nowhere where it shouldn't.

This in turn frees Lizzie up to play merry hell inside the `UserAccount` object itself. As long as she receives a `Chargeable` object and emits an exception when the value of that object can't be charged to the user's account, it's not going to matter too much what the actual logic implementation looks like. No other piece of code has that knowledge, so no other piece of code will get broken due to changes in the way that that knowledge is implemented.

This is the situation that we want to find ourselves in, one that promotes loose coupling and one that helps us to properly encapsulate the knowledge of potentially complicated operations into the places where they should be.

The *Law of Demeter* helps us to achieve both of these desirable outcomes.

Summary

The *Law of Demeter* wants us to keep our friends close and our enemies as far away as possible, erecting barricades and boundary fences along the way. Tight coupling is one such enemy and, whilst it's certainly unavoidable 100% of the time, erecting the proper boundaries allows us to avoid the miserable loss of pub time due to *action at a distance* effects.

In our erroneous early example, our controller code was reaching through both the `$user` instance and the `$userAccount` instance to get at the `deductAmount()`

method of the `$balance` instance. Hopefully now it should be clear how this can lead to problems. The developer that is working on the `UserAccount` or the `Balance` class isn't necessarily going to be aware that the controller code was accessing the `$balance` instance's methods directly.

With the appropriate barrier in place, the `payForOrder(Chargeable $order);` method, the controller code becomes ignorant of any changes to the way that order payments are settled, which is precisely how we want it.

One thing that we still have to guard against though is the proliferation of proxy methods that can lead to a bloated, muddied interface.

But with loose coupling and encapsulation promoted through the application of *The Law of Demeter*, we're golden.

Don't talk to strangers. Unless they're offering to pick up your bar tab.

Fin

Thank you for taking the time to read through my free sample. I hope that you found it interesting enough to go on and purchase the book itself.

With the rest of the book, you'll encounter more of the same kind of thinking. Some of it you will already be familiar with. Other parts, you won't. Some of the ideas that I present in the coming chapters are controversial. Others are simply the kind of things that *any* PHP developer worth his salt should know.

You might have noticed that this book isn't a straight up "How To". Even when a chapter starts with the most basic introduction, it isn't the goal of that chapter to present a linear tutorial that ends with a working solution. Rather, I present each concept as more of a sight-seeing tour. When you reach a certain level of ability with your coding, being able to see the wood and the trees at the same time is a much more valuable quality than being able to recall a straight-up tutorial.

This idea is what underpins the notion of PHP Brilliance: The ability to take a more holistic view of the programming problems before us, in recognition of the fact that so many programming topics are knitted together, interlinked, inter-dependent. Encapsulation forms part of the consideration surrounding the Single Responsibility Principle, which in turn is one of the SOLID principles, which contains the Liskov Substitution Principle, which in itself has a direct bearing on abstraction, inheritance, and the notion of programming to an interface rather than an implementation.

The concepts that lead to high quality, enterprise grade application development simply cannot be boxed up into self-contained units.

But let's cut to the quick. When a developer achieves a comfortable mastery of the topics that are covered in this book, she becomes a much more valuable employee. When I interview developers for positions on project teams, the one that knows this stuff is the one most likely to be offered the job. In effect, it's like a wishlist of the knowledge that I hope my next interview candidate will have.

The reason for this is relatively simple. I've constructed this book to be representative of the knowledge that I believe all senior developers and above should be comfortable

with. If you're applying for a top level technical position and you're comfortable with all of this, you're already ahead of the pack.

In case you might be interested, here's a (non-exhaustive) list of the topics included within the book.

- The four central tenets: a fresh look at encapsulation, abstraction, inheritance and polymorphism.
- Interfaces, namespaces, traits and closures
- Loose coupling and dependency injection
- Programming principles, patterns and anti-patterns
- The five SOLID principles
- Dependency Injection Containers and super factories.
- Architectural concerns: MVC and its siblings, Service oriented architecture, APIs and Microservices, the Architectural Fortress and other architectural designs
- What's new in PHP7 and how we can use it to excel.

Thank you again for taking the time to read this far.

Thunder

https://phpbrilliance.com[6]

[6] https://phpbrilliance.com