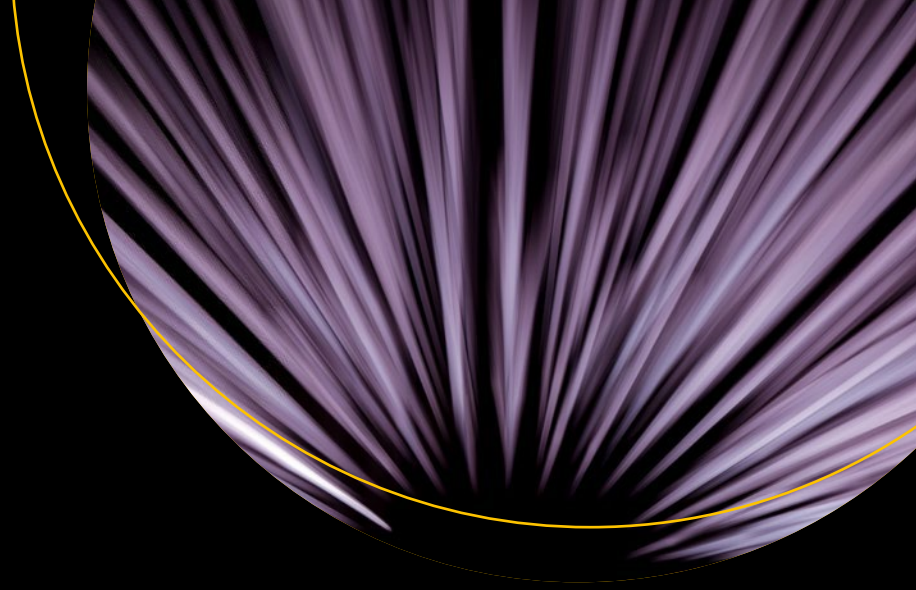


PHP 8 Objects, Patterns, and Practice

Mastering OO Enhancements, Design Patterns, and Essential Development Tools

—

Sixth Edition

—

Matt Zandstra

Apress®

PHP 8 Objects, Patterns, and Practice

Mastering OO Enhancements, Design Patterns, and Essential Development Tools

Sixth Edition

Matt Zandstra

Apress®

PHP 8 Objects, Patterns, and Practice: Mastering OO Enhancements, Design Patterns, and Essential Development Tools

Matt Zandstra
Brighton, UK

ISBN-13 (pbk): 978-1-4842-6790-5 ISBN-13 (electronic): 978-1-4842-6791-2
https://doi.org/10.1007/978-1-4842-6791-2

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Devin Avery on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484267905. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

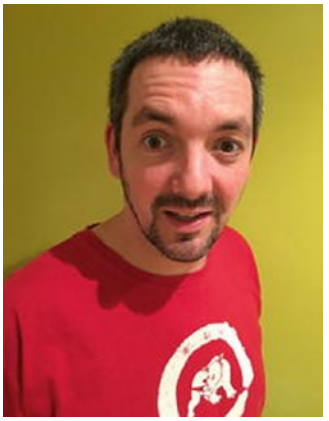
To Louise. Still the whole point.

Table of Contents

About the Author

Matt Zandstra has worked as a web programmer, consultant, and writer for over two decades. He is the author of *SAMS Teach Yourself PHP in 24 Hours* (three editions) and is a contributor to *DHTML Unleashed*. He has written articles for *Linux Magazine*, Zend, IBM DeveloperWorks, and *php|architect* magazine, among others. Matt was a senior developer/tech lead at Yahoo! and API tech lead at LoveCrafts. Matt works as a consultant advising companies on their architectures and system management and also develops systems primarily with PHP and Java. Matt also writes fiction.

About the Technical Reviewer



Paul Tregoing has worked in ops and development in a variety of environments for nearly 20 years. He worked at Yahoo! for 5 years as a senior developer on the frontpage team; there he generated his first PHP using Perl. Other employers include Bloomberg, Schlumberger, and the British Antarctic Survey, where he became intimate with thousands of penguins.

He now works as a freelance engineer for various clients, small and large, building multitiered web apps using PHP, JavaScript, and many other technologies. Paul is a voracious consumer of science fiction and fantasy and harbors not-so-secret ambitions to try his hand at writing in the near future. He lives in Cambridge, United Kingdom, with his wife and children.

Acknowledgments

As always, I have benefited from the support of many people while working on this edition. But as always, I must also look back to the book's origins. I tried out some of this book's underlying concepts in a talk in Brighton, back when we were all first marveling at the shiny possibilities of PHP 5. Thanks to Andy Budd, who hosted the talk, and to the vibrant Brighton developer community. Thanks also to Jessey White-Cinis, who was at that meeting and who put me in touch with Martin Streicher at Apress.

Once again, this time around, the Apress team has provided enormous support, feedback, and encouragement. I am lucky to have benefited from such professionalism.

I'm very lucky to have had my friend and colleague, Paul Tregoing, working on this edition as Technical Reviewer. The fact that PHP itself was under active development throughout the writing of this book demanded extra vigilance. Code examples that were perfectly valid in early drafts were rendered incorrect by the language's fast evolution. Once again, this edition has greatly benefited from Paul's knowledge, insight, and attention to detail—many thanks Paul!

Thanks and love to my wife, Louise. The production of this book has coincided with three pandemic lockdowns, so thanks are also due to our children, Holly and Jake, for many much-needed distractions—often provided during Zoom meetings conducted in my office space (the corner of the kitchen table).

Thanks to Steven Metsker for his kind permission to reimplement in PHP a simplified version of the parser API he presented in his book, *Building Parsers with Java* (Addison-Wesley Professional, 2001).

I write to music, and, in previous editions of this book, I remembered the great DJ, John Peel, champion of the underground and the eclectic. The soundtrack for this edition was largely provided by BBC Radio 3's contemporary music show, *Late Junction*, played on a loop. Thanks to them for keeping things weird.

Introduction

When I first conceived of this book, object-oriented design in PHP was an esoteric topic. The intervening years have not only seen the inexorable rise of PHP as an object-oriented language but also the march of the framework. Frameworks are incredibly useful, of course. They manage the guts and the glue of many (perhaps, these days, most) web applications. What's more, they often exemplify precisely the principles of design that this book explores.

There is, though, a danger for developers here, as there is in all useful APIs. This is the fear that one might find oneself relegated to userland, forced to wait for remote gurus to fix bugs or add features at their whim. It's a short step from this standpoint to a kind of exile in which one is left regarding the innards of a framework as advanced magic and one's own work as not much more than a minor adornment stuck up on top of a mighty unknowable infrastructure.

Although I'm an inveterate reinventor of wheels, the thrust of my argument is not that we should all throw away our frameworks and build MVC applications from scratch (at least not always). It is rather that, as developers, we should understand the problems that frameworks solve and the strategies they use to solve them. We should be able to evaluate frameworks not only functionally but in terms of the design decisions their creators have made and to judge the quality of their implementations. And yes, when the conditions are right, we should go ahead and build our own spare and focused applications and, over time, compile our own libraries of reusable code.

I hope this book goes some way toward helping PHP developers apply design-oriented insights to their platforms and libraries and provides some of the conceptual tools needed when it's time to go it alone.

PART I

Objects

CHAPTER 1

PHP: Design and Management

In July 2004, PHP 5.0 was released. This version introduced a suite of radical enhancements. Perhaps first among these was radically improved support for object-oriented programming. This stimulated much interest in objects and design within the PHP community. In fact, this was an intensification of a process that began when version 4 first made object-oriented programming with PHP a serious reality.

In this chapter, I look at some of the needs that coding with objects can address. I very briefly summarize some aspects of the evolution of patterns and related practices.

I also outline the topics covered by this book. I will look at the following:

- *The evolution of disaster*: A project goes bad
- *Design and PHP*: How object-oriented design techniques took root in the PHP community
- *This book*: Objects, Patterns, Practice

The Problem

The problem is that PHP is just too easy. It tempts you to try out your ideas and flatters you with good results. You write much of your code straight into your web pages, because PHP is designed to support that. You add utility functions (such as database access code) to files that can be included from page to page, and before you know it, you have a working web application.

You are well on the road to ruin. You don't realize this, of course, because your site looks fantastic. It performs well, your clients are happy, and your users are spending money.

M. Zandstra, *PHP 8 Objects, Patterns, and Practice*, https://doi.org/10.1007/978-1-4842-6791-2_1

Trouble strikes when you go back to the code to begin a new phase. Now you have a larger team, some more users, and a bigger budget. Yet, without warning, things begin to go wrong. It's as if your project has been poisoned.

Your new programmer is struggling to understand code that is second nature to you, although perhaps a little byzantine in its twists and turns. She is taking longer than you expected to reach full strength as a team member.

A simple change, estimated at a day, takes three days when you discover that you must update 20 or more web pages as a result.

One of your coders saves his version of a file over major changes you made to the same code some time earlier. The loss is not discovered for three days, by which time you have amended your own local copy. It takes a day to sort out the mess, holding up a third developer who was also working on the file.

Because of the application's popularity, you need to shift the code to a new server. The project has to be installed by hand, and you discover that file paths, database names, and passwords are hard-coded into many source files. You halt work during the move because you don't want to overwrite the configuration changes the migration requires. The estimated two hours becomes eight as it is revealed that someone did something clever involving the Apache module ModRewrite, and the application now requires this to operate properly.

You finally launch phase 2. All is well for a day and a half. The first bug report comes in as you are about to leave the office. The client phones minutes later to complain. Her report is similar to the first, but a little more scrutiny reveals that it is a different bug causing similar behavior. You remember the simple change back at the start of the phase that necessitated extensive modifications throughout the rest of the project.

You realize that not all of the required modifications are in place. This is either because they were omitted to start with or because the files in question were overwritten in merge collisions. You hurriedly make the modifications needed to fix the bugs. You're in too much of a hurry to test the changes, but they are a simple matter of copy and paste, so what can go wrong?

The next morning, you arrive at the office to find that a shopping basket module has been down all night. The last-minute changes you made omitted a leading quotation mark, rendering the code unusable. Of course, while you were asleep, potential customers in other time zones were wide awake and ready to spend money at your store. You fix the problem, mollify the client, and gather the team for another day's firefighting.

This everyday tale of coding folk may seem a little over the top, but I have seen all these things happen over and over again. Many PHP projects start their life small and evolve into monsters.

Because the presentation layer also contains application logic, duplication creeps in early as database queries, authentication checks, form processing, and more are copied from page to page. Every time a change is required to one of these blocks of code, it must be made everywhere that the code is found, or bugs will surely follow.

Lack of documentation makes the code hard to read, and lack of testing allows obscure bugs to go undiscovered until deployment. The changing nature of a client's business often means that code evolves away from its original purpose until it is performing tasks for which it is fundamentally unsuited. Because such code has often evolved as a seething, intermingled lump, it is hard, if not impossible, to switch out and rewrite parts of it to suit the new purpose.

Now, none of this is bad news if you are a freelance PHP consultant. Assessing and fixing a system like this can fund expensive espresso drinks and DVD box sets for six months or more. More seriously, though, problems of this sort can mean the difference between a business's success and failure.

PHP and Other Languages

PHP's phenomenal popularity meant that its boundaries were tested early and hard. As you will see in the next chapter, PHP started life as a set of macros for managing personal home pages. With the advent of PHP 3 and, to a greater extent, PHP 4, the language rapidly became the successful power behind large enterprise websites. In many ways, however, the legacy of PHP's beginnings carried through into script design and project management. In some quarters, PHP retained an unfair reputation as a hobbyist language, best suited for presentation tasks.

About this time (around the turn of the millennium), new ideas were gaining currency in other coding communities. An interest in object-oriented design galvanized the Java community. Since Java is an object-oriented language, you may think that this is a redundancy. Java provides a grain that is easier to work with than against, of course, but using classes and objects does not in itself determine a particular design approach.

The concept of the design pattern as a way of describing a problem, together with the essence of its solution, was first discussed in the 1970s. Perhaps aptly, the idea originated in the field of architecture, not computer science, in a seminal work by Christopher

Alexander: *A Pattern Language* (Oxford University Press, 1977). By the early 1990s, object-oriented programmers were using the same technique to name and describe problems of software design. The seminal book on design patterns, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional, 1995), by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (henceforth referred to in this book by their affectionate nickname, the *Gang of Four*), is still indispensable today. The patterns it contains are a required first step for anyone starting out in this field, which is why most of the patterns in this book are drawn from it.

The Java language itself deployed many core patterns in its API, but it wasn't until the late 1990s that design patterns seeped into the consciousness of the coding community at large. Patterns quickly infected the computer sections of Main Street bookstores, and the first flame wars began on mailing lists and in forums.

Whether you think that patterns are a powerful way of communicating craft knowledge or largely hot air (and, given the title of this book, you can probably guess where I stand on that issue), it is hard to deny that the emphasis on software design they have encouraged is beneficial in itself.

Related topics also grew in prominence. Among them was Extreme Programming (XP), championed by Kent Beck. XP is an approach to projects that encourages flexible, design-oriented, highly focused planning and execution.

Prominent among XP's principles is an insistence that testing is crucial to a project's success. Tests should be automated, run often, and preferably designed before their target code is written.

XP also dictates that projects should be broken down into small (very small) iterations. Both code and requirements should be scrutinized at all times. Architecture and design should be a shared and constant issue, leading to the frequent revision of code.

If XP was the militant wing of the design movement, then the moderate tendency is well represented by one of the best books about programming that I have ever read: *The Pragmatic Programmer: From Journeyman to Master* by Andrew Hunt and David Thomas (Addison-Wesley Professional, 1999).

XP was deemed a tad cultish by some, but it grew out of two decades of object-oriented practice at the highest level, and its principles were widely cannibalized. In particular, code revision, known as refactoring, was taken up as a powerful adjunct to patterns. Refactoring has evolved since the 1980s, but it was codified in Martin Fowler's catalog of refactorings, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley Professional), which was published in 1999 and defined the field.

Testing, too, became a hot issue with the rise to prominence of XP and patterns. The importance of automated tests was further underlined by the release of the powerful JUnit test platform, which became a key weapon in the Java programmer's armory. A landmark article on the subject, "Test Infected: Programmers Love Writing Tests" by Kent Beck and Erich Gamma (http://junit.sourceforge.net/doc/testinfected/testing.htm), gives an excellent introduction to the topic and remains hugely influential.

PHP 4 was released at about this time, bringing with it improvements in efficiency and, crucially, enhanced support for objects. These enhancements made fully object-oriented projects a possibility. Programmers embraced this feature, somewhat to the surprise of Zend founders Zeev Suraski and Andi Gutmans, who had joined Rasmus Lerdorf to manage PHP development. As you shall see in the next chapter, PHP's object support was by no means perfect. But with discipline and careful use of syntax, one could really begin to think in objects and PHP at the same time.

Nevertheless, design disasters such as the one depicted at the start of this chapter remained common. Design culture was some way off and almost nonexistent in books about PHP. Online, however, the interest was clear. Leon Atkinson wrote a piece about PHP and patterns for Zend in 2001, and Harry Fuecks launched his journal at www.phppatterns.com (now defunct) in 2002. Pattern-based framework projects such as BinaryCloud began to emerge, as well as tools for automated testing and documentation.

The release of the first PHP 5 beta in 2003 ensured the future of PHP as a language for object-oriented programming. Zend Engine 2 provided greatly improved object support. Equally important, it sent a signal that objects and object-oriented design were now central to the PHP project.

Over the years, PHP 5 continued to evolve and improve, incorporating important new features such as namespaces and closures. During this time, it secured its reputation as the best choice for server-side web programming.

PHP 7, released in December 2015, represented a continuation of this trend. In particular, it provided support for both parameter and return type declarations—two features that many developers (together with previous editions of this book) had been clamoring for over the years. There were many other features and improvements including anonymous classes, improved memory usage, and boosted speed. Over the years, the language grew steadily more robust, cleaner, and more fun to work with from the perspective of an object-oriented coder.

In December 2020, almost exactly five years after the release of PHP 7, PHP 8 is due for release. While some of the implementation details may change (and have changed a little during the writing of this book), the features are already available at the time of this writing (August 2020). I cover many of them in detail here. They include improvements to type declarations, streamlined property assignment, and many other new features. The headline addition is, perhaps, support for attributes (often called annotations in other languages).

About This Book

This book does not attempt to break new ground in the field of object-oriented design; in that respect, it perches precariously on the shoulders of giants. Instead, I examine, in the context of PHP, some well-established design principles and some key patterns (particularly those inscribed in *Design Patterns*, the classic Gang of Four book). Finally, I move beyond the strict limits of code to look at tools and techniques that can help to ensure the success of a project. Aside from this introduction and a brief conclusion, the book is divided into three main parts: objects, patterns, and practice.

Objects

I begin Part 1 with a quick look at the history of PHP and objects, charting their shift from afterthought in PHP 3 to core feature in PHP 5.

You can still be an experienced and successful PHP programmer with little or no knowledge of objects. For this reason, I start from first principles to explain objects, classes, and inheritance. Even at this early stage, I look at some of the object enhancements that PHP 5, PHP 7, and PHP 8 introduced.

The basics established, I delve deeper into our topic, examining PHP's more advanced object-oriented features. I also devote a chapter to the tools that PHP provides to help you work with objects and classes.

It is not enough, however, to know how to declare a class, and to use it to instantiate an object. You must first choose the right participants for your system and decide the best ways for them to interact. These choices are much harder to describe and to learn than the bald facts about object tools and syntax. I finish Part 1 with an introduction to object-oriented design with PHP.

Patterns

A pattern describes a problem in software design and provides the kernel of a solution. "Solution" here does not mean the kind of cut-and-paste code that you might find in a cookbook (excellent though cookbooks are as resources for the programmer). Instead, a design pattern describes an approach that can be taken to solve a problem. A sample implementation may be given, but it is less important than the concept that it serves to illustrate.

Part 2 begins by defining design patterns and describing their structure. I also look at some of the reasons behind their popularity.

Patterns tend to promote and follow certain core design principles. An understanding of these can help in analyzing a pattern's motivation and can usefully be applied to all programming. I discuss some of these principles. I also examine the Unified Modeling Language (UML), a platform-independent way of describing classes and their interactions.

Although this book is not a pattern catalog, I examine some of the most famous and useful patterns. I describe the problem that each pattern addresses, analyze the solution, and present an implementation example in PHP.

Practice

Even a beautifully balanced architecture will fail if it is not managed correctly. In Part 3, I look at the tools available to help you create a framework that ensures the success of your project. If the rest of the book is about the practice of design and programming, Part 3 is about the practice of managing your code. The tools that I examine can form a support structure for a project, helping to track bugs as they occur, promoting collaboration among programmers, and providing ease of installation and clarity of code.

I have already discussed the power of the automated test. I kick off Part 3 with an introductory chapter that gives an overview of problems and solutions in this area.

Many programmers are guilty of giving in to the impulse to do everything themselves. Composer, together with Packagist, its main repository, offers access to thousands of dependency managed packages that can be stitched into projects with ease. I look at the trade-offs between implementing a feature yourself and deploying a Composer package.

While I'm on the topic of Composer, I look at the installation mechanism that makes the deployment of a package as simple as a single command.