PHP Objects, Patterns, and Practice, Second Edition

ISBN-13 (pbk): 978-1-59059-909-9

ISBN-10 (pbk): 1-59059-909-8

ISBN-13 (electronic): 978-1-4302-0466-4

ISBN-10 (electronic): 1-4302-0466-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Tom Welsh
Technical Reviewer: Tolan Blundell
Editorial Board: Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Jason Gilmore, Kevin Goff, Jonathan Hassell, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Richard Dal Porto
Copy Editors: Heather Lang, Benjamin Berg
Associate Production Director: Kari Brooks-Copony
Senior Production Editor: Laura Cheu
Compositor: Kinetic Publishing Services, LLC
Proofreader: Nancy Riddiough
Indexer: Becky Hornyak
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 1

PHP: Design and Management

When PHP 5 was released early in 2004, a mong the most important features it introduced was enhanced 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 the evolution of patterns and related practices in the Java world. I look at signs that indicate a similar process is occurring among PHP coders.

I also outline the topics covered by this book.

I will look at

- *The evolution of disaster*: A project goes bad.
- *Design and PHP*: How object-oriented design techniques are taking 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.

Trouble strikes when you go back to the code to begin a new phase. Now you have a larger team, some more users, 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, though 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 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 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 or 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 Web sites. In many ways, though, 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. You may think that this is a redundancy, since Java is an object-oriented language. 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 '70s. Perhaps aptly, the idea originated in the field of architecture, and not computer science. By the early '90s, 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*, by the affectionately nicknamed *Gang of Four*, was published in 1995, and 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 '90s that design patterns seeped into the consciousness of the coding community at large. Patterns quickly infected the computer sections of High Street bookstores, and the first "hype or tripe" flame wars began on mailing lists and 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 is the militant wing of the design movement, then the moderate tendency is well represented by one of the best books about programming I have ever read: *The Pragmatic Programmer* by Andrew Hunt and David Thomas, which was published in 2000.

XP is 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 '80s, but it was codified in Martin Fowler's catalog of refactorings, *Refactoring: Improving the Design of Existing Code*, 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 think in objects and PHP at the same time.

Nevertheless, design disasters like 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, though, the interest was clear. Leon Atkinson wrote a piece about PHP and patterns for Zend in 2001 (http://www.zend.com/zend/trick/tricks-app-patt-php.php), and Harry Fuecks launched his journal at http://www.phppatterns.com (now largely mothballed, it seems) 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. The Zend 2 Engine provided greatly improved object support, as you shall see. Equally important, it sent a signal that objects and object-oriented design were now central to the PHP project.

At the time of this writing (September 2007), we are moving closer to a beta release of PHP 6, which promises to consolidate PHP's standing as an object-friendly language, with likely new features such as namespaces. In fact, PHP 6 namespaces are already available in development form, and I cover them in Chapter 5.

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 upon 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 2 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 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, though, 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 2 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 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 it serves to illustrate.

Part 3 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 4, 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 4 is about the practice of managing your code. The tools 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 4 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. The PHP community maintains PEAR, a repository of quality-controlled packages that can be stitched into projects with ease. I look at the trade-offs between implementing a feature yourself and deploying a PEAR package.

While I'm on the topic of PEAR, I look at the installation mechanism that makes the deployment of a package as simple as a single command. Best suited for stand-alone packages, this mechanism can be used to automate the installation of your own code. I show you how to do it.

Documentation can be a chore, and along with testing, it is probably the easiest part of a project to jettison when deadlines loom. I argue that this is probably a mistake, and show you PHPDocumentor, a tool that helps you turn comments in your code into a set of hyperlinked HTML documents that describe every element of your API.

Almost every tool or technique discussed in this book directly concerns or is deployed using PHP. The one exception to this rule is Concurrent Versions System (CVS). CVS is a version control system that enables many programmers to work together on the same codebase

without overwriting one another's work. CVS lets you grab snapshots of your project at any stage in development, see who has made which changes, and split the project into mergeable branches. CVS will save your project one day.

Two facts seem inevitable. First, bugs often recur in the same region of code, making some work days an exercise in déjà vu. Second, often improvements break as much as, or more than, they fix. Automated testing can address both of these issues, providing an early warning system for problems in your code. I introduce PHPUnit, a powerful implementation of the so-called xUnit test platform designed first for Smalltalk but ported now to many languages, notably Java. I look in particular at PHPUnit's features and more generally at the benefits, and some of the costs, of testing.

PEAR provides a build tool that is ideal for installing self-enclosed packages. For a complete application, however, greater flexibility is required. Applications are messy. They may need files to be installed in nonstandard locations, or want to set up databases, or need to patch server configuration. In short, applications need *stuff* to be done during installation. Phing is a faithful port of a Java tool called Ant. Phing and Ant interpret a build file and process your source files in any way you tell them to. This usually means copying them from a source directory to various target locations around your system, but as your needs get more complex, Phing scales effortlessly to meet them.

What's New in the Second Edition

The first edition of this book was published late in 2004, when PHP 5 was still avalable only as beta software. Since then, PHP has continued to evolve and mature. This new edition has been reviewed and thoroughly updated to take account of changes and new opportunities. I use the more recent PDO (PHP Data Objects) extension, in place of the `PEAR::DB` package, for example.

Many of the chapters have been expanded to cover more ground, and I have added two extra chapters. Chapter 13 covers database patterns, taking in some techniques for mapping relational data to the more organic structures that typify object relations. Chapter 18 covers testing with PHPUnit. Both chapters focus on themes that were touched on in the first edition, but with the luxury of more space comes the freedom for further exploration.

Summary

This is a book about object-oriented design and programming. It is also about tools for managing a PHP codebase from collaboration through to deployment.

These two themes address the same problem from different but complementary angles. The aim is to build systems that achieve their objectives and lend themselves well to collaborative development.

A secondary goals lies in the aesthetics of software systems. As programmers, we build machines that have shape and action. We invest many hours of our working day, and many days of our lives, writing these shapes into being. We want the tools we build, whether individual classes and objects, software components, or end products, to form an elegant whole. The process of version control, testing, documentation, and build does more than support this objective, it is part of the shape we want to achieve. Just as we want clean and clever code, we want a codebase that is designed well for developers and users alike. The mechanics of sharing, reading, and deploying the project should be as important as the code itself.