Part I

Getting Acquainted with Design Patterns and PHP

1

Understanding Design Patterns

Usually when I pick up a book and see a chapter longer than five pages about a topic that I'm not the most familiar with, I tend to get scared. More than five pages may see me dropping the book and running away, flailing my arms and shouting about how tough these computers are! While this chapter may be longer than five pages, don't be discouraged. The term *Design Pattern* is just a fancy name for something that is not all that complex. A good portion of this chapter is taking what you may already know and use regularly and refining it to a more concise definition. So, let's jump in and see what Design Patterns really are.

What Are They?

The story of Steve that follows helps describe Design Patterns in a real-world context. I'm hoping that you're not too familiar with this story!

An All Too Common Example

Steve works at a large insurance firm. His most recent task was developing a way to show customer information to the call center representatives over a web interface. He designed a complex system that would allow the reps to search for a customer, enter call logs, update customer coverage information, and process payments. The system went into place smoothly, minus the few bumps and hiccups that a new installation in a production environment always runs into. Steve is happy, relaxed, and ready to sit back in the break room sipping his free coffee.

Overnight, the insurance company triples in size from its most recent investment. Not only is Steve called back to work on providing new scalability and enhancements to the call center software but there has also been buzz about adding some new features to the corporate site to support the new acquisition's customers. Steve's department is also increased to include two new developers, Andy and Jason.

The news comes down from the vice president that the corporate site needs to allow customers to process their payments after they have completed a successful, secure user log in. Additionally, the system needs to show how many times the customer has called in to the call center. Finally, it needs to show an audit log of every change the call center has made to the customer's account.

Steve knows that he can easily update the call center software to provide the audit log and then copy over the code, tweak it, and make use of the payment processing. However, the new programmers need to be tasked without much time to get up to speed on the new system. Steve's boss has assigned them the portions of the project that Steve is most familiar with. Since Steve is the rock-star PHP programmer with the most experience, his boss needs him to work on the other portions of the corporate site as soon as possible after which he'll then come around and make use of the new programmers' changes to the auditing on the call center software. In the end, it will be his responsibility to provide hooks for the new payment-processing portion of the user login screen.

Steve's code isn't bad, but it seems to be taking Jason a bit longer to follow through and port the payment-processing portion into the corporate site. Instead, he determines he could finish faster by writing it in his own method. He mentions this to Steve and continues on his way. Andy is also struggling. Since his Master's in computer science is newly acquired, he hasn't had much time to gain experience with the jumbled code that sometimes supports existing enterprises.

Through much struggle and late nights, the team is successful and deploys the new code changes. Andy feels like everything could have been architected better. Steve thinks that if the other programmers would have just copied and pasted his code, things would have gone must faster; Jason and Andy just needed to make a few tweaks and it would have been solid. Jason mentioned that he was confused about why some functionality was implemented in one way in one section of the code and in a different way in a different piece. That is what threw him off.

As the website continues to gain more visitors, the performance begins to suffer. Steve's boss suggests that the team take a few days and look at the code for optimization.

Jason discovers that the method that he wrote for payment processing is nearly the same as Steve's. Jason combines and tweaks the methods into one class. Steve is starting to see similarities between the authentication code that he wrote for the call center site and the classes he authored for the corporate site's user login. Andy is realizing that every PHP page they create has the same set of function calls at the top of it. He creates a bootstrap type class to bring this all into one location to reduce code duplication.

From outside this example, you can objectively see many things. Steve's code could have benefited from commonality in its approach. Andy's formal education in software design made him sometimes question PHP's ability to accomplish the tasks and question the architecture. Jason couldn't easily understand Steve's payment system, so he opted to create his own, causing code duplication. Finally, after the software analysis, the team started discovering patterns in their seemingly jumbled code base. This is the beginning of this team's foray into Design Patterns.

Design Patterns Are Solving the Same Problem

In the previous example, Steve's team stumbled into the first important part of the Design Pattern concept. Patterns are not intentionally created in software development. They are more often discovered through practice and application in real-world situations. The payment application system and the bootstrap type calls being consolidated into classes are examples of identifying patterns in programming.

It was once said that every single piece of music that could be written already has been. Now, new music creation is just the rearranging of those particular sets of notes to different tempos and speeds. It's the same with general software development, barring a few major groundbreaking exceptions. The same problems come up repeatedly and require common solutions. This is exactly what Design Patterns are: reusable solutions for these common problems.

No book mentioning Design Patterns would be complete without the reference to the *Gang of Four*: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, authors of the original Design Patterns book. After a considerable amount of time in the field, they started noticing particular patterns of design emerging from various development projects. Collectively, they gathered these ideas together to form the initial Design Patterns concept. Recognizing these as templates for future development, they were able to put them into an easy-to-understand reference with digestible segments for large, complex programming concepts.

While Design Patterns can encompass many things — from interface design to architecture, and even marketing and metrics — this particular book will focus on development language construction using Object Oriented Programming.

A problem in software design consists of three parts:

- The "what" is considered the business and functionality requirements.
- The "how" is the particular design that you use to meet those requirements.
- The "work" is the actual implementation, or the "how" put into actual application and practice.

Design Patterns fit into the "how" of this process, and as a result, this book describes the "how" of solving these problems as well as portions of the "work" necessary to make these solutions successful. You can picture PHP as the vehicle behind the "why" of the problem solving. Once you know "what" the software needs to do, and you've designed "how" it can do it, the "work" becomes a lot easier with a lot less refactoring.

I can't stress this enough: the PHP language, your grasp of it and the way you understand its intricacies is not the focus of this book. Instead, I bring common, time tested methodologies into focus, describe them, and relate them to PHP.

Patterns naturally start to come out of software development, as you saw in the example. However, having a full playbook that references existing patterns can make the architecture planning faster and the choices better. As an added bonus, programmers coming from different software realms may recognize the pattern and just have to adapt to the specifics of the language. Having a clear set of patterns in your application may also help new members of your team grasp your project, lowering your ramp-up time.

Design Patterns Are Around You All the Time

You've seen how Steve's team was able to grasp basic patterns in their software and create reusable items. You may also be able to draw parallels to your own software development now. How many times have you created the same user login and authentication system using your user class? Do you have a `db()` function sitting around somewhere that you favor? These are examples of how you've already been using patterns.

Even more detailed and closer to the root patterns are examples found in your favorite PEAR or other framework libraries. For example, using PEAR DB is an example of putting a Design Pattern into use (notably the factory method). The Zend Framework also uses various different patterns such as the Singleton and the Adapter patterns.

The Common Parts of a Design Pattern

The Gang of Four pioneered a documentation standard for describing Design Patterns. They used this in their book for each of the patterns that they mentioned. Authors after them have copied this exact format and continued to propagate this form of documentation. I was a little bit less verbose with you because I felt a lot of the sections either reiterated the sections above them or were just there for structure's sake. The introduction to this book mentions the four main parts of each pattern's documentation: the name, the problem and solution, the Universal Modeling Language (UML) diagram, and the code example.

The Name

The name is actually more important in Design Patterns than you may initially guess it is. Proper descriptive naming conventions can go a long way toward explaining the behavior and relationship of the pattern to the project and other patterns.

In the example for this chapter, you saw how Jason mentioned to Steve that he was going to rewrite a portion of the payment-processing system. Since Steve was the senior programmer, he may not have necessarily agreed with the approach that Jason was using, but he certainly could have suggested some patterns to be used in that rewrite's architecture. This way, the entire team would both be familiar with the underlying concepts of the payment system, with Jason specializing in the exact implementation.

Problem and Solution

As mentioned previously, Design Patterns are what emerge from solving the same problem with the same general solution. This section of the description covers the main problem or problems in your project and then shows how this particular Design Pattern is one of the better solutions.

As you may have noticed, I didn't use the phrase "the best solution" because no one can say this definitively. Even if you find what you believe is the best Design Pattern for a particular problem, you're going to have to apply a certain amount of tweaking to it in order for it to fit perfectly into your project.

UML Diagram

The UML diagram will show the general structure of the pattern. In some cases, it may be necessary to generate more than one diagram to show additional implementations of the pattern or to illustrate a complex concept in easier-to-understand segments.

> **What is UML?**
>
> *Unified Modeling Language (UML)* diagrams should be a staple in your programming arsenal. UML is a standard way to diagram programming actions, objects and use cases. This helps communicate your design when building complex software in PHP. For a quick refresher on UML, visit the `http://Wikipedia.org/wiki/Unified_Modeling_Language` page on Wikipedia.

You may find that the building blocks for generating your own UML diagrams for your project can be loosely based on these generic pattern diagrams. Of course, your method names, class names, and attributes will vary and be more complex than those in the example.

The Code Example

Hands-on PHP programmers are finally rejoicing: the code examples. These are going to be relatively simple examples of the Design Pattern concept put into PHP code. The bonus here in having a PHP-based Design Patterns book is that you don't necessarily need to know another language to see an example of this pattern. (Other books focusing on Enterprise Design Patterns have used Java or C examples, somewhat taking away the effectiveness of the example to a sole-language programmer.)

I continue to reiterate: the code examples are simply that. They are not meant to be plug and play. They may not contain error logging or handling, auditing, or wholly secure programming techniques. This is not to say that I don't appreciate high-quality, secure programming (previous teammate programmers of mine can confirm that I'm a stickler for details), but it would distract from the main concept that I'm trying to explain.

What Design Patterns Are Not

It's important to rein in the explanation of Design Patterns by also talking about what they don't encompass. Up until now, you may have noticed that I've created a pretty large umbrella of coverage for the Design Pattern definition.

Design Patterns Are Not Plug and Play

If you're expecting to flip to the Design Pattern pages of this book and see full examples that you can quickly copy and paste for your next project, you will be sadly disappointed. Design Patterns are not just a simple plug and play solution to your programming project.

Design Patterns are not the actual implementation or even the algorithm for solving the problem. For example, you may create a design such that every house you construct has windows in the south to let in more heat and light. You are not actually doing the constructing with exact measurements and locations of the windows. You just hand over your design to the builder (programmer in our case), and they implement it.

Another analogous way to view Design Patterns is to compare them to musical notes on a scale. You may know all the notes in a minor scale, but playing them exactly in order and in the same tempo does not make an enjoyable song. You can't open up a scale book, grab the scale, play it on guitar, and expect everyone to think you're an amazing song writer. It would be quite boring and wouldn't solve the problem your music is made for: to demonstrate a specific set of emotions via art. In this way, Design Patterns are like those scales in the book. While they are the building blocks of a great solo, it is up to you to apply them, tweak them, and create a great song.

Design Patterns are Maintainable But Not Always Most Efficient

Design patterns don't always lend themselves to the greatest efficiency and speed in applications either. The goal of a Design Pattern is to help you design a solution in an easily repeatable and reusable way. This means the Pattern may not be specifically tailored to your situation but will have greater code maintainability and understandability.

Design Patterns are a Vehicle, Not a Refactoring End

A particular supervisor of mine just finished reading a book by Joshua Kerievsky and came to me with his newly acquired knowledge. He told me that we need to refactor our code base to use all Design Patterns. We had a discussion about what refactoring really meant, especially in our context.

While respecting Kerievsky and not disagreeing, I do feel that a greater distinction should be maintained when coupling Design Patterns to refactoring. Refactoring approaches both creating a more efficient code base and improving the maintainability and clarity of the code. Design Patterns are a great vehicle for your refactoring approach, but shouldn't be the destination. While I'm in favor of starting a project with a highly detailed set of Design Pattern architecture specs, I don't want to force something into a pattern for patterns' sake. Imagine if the first rock bands in existence threw a piano into the mix just because everyone else in music was doing it, and they thought they had to. You wouldn't have that classic guitar-driven rock music that we've come to love!

Design Pattern Demonstration

Most examples of Design Patterns historically have been very sparse and theoretical so as not to have the reader confuse the core concepts with language-agnostic features. Readers who have studied Design Patterns, or even Object Oriented Programming before will be very familiar with the ever-present square, circle, and oval object examples.

The debate rages on about Design Pattern books using simple objects like squares or people in their examples. Purists say you should detail the Design Pattern concepts and practice and give the simplest examples possible so as not to distract from the actual implementation of the pattern. (These are the people that hated story problems in math class because of all the extra information!) In my experience, self-taught PHP programmers prefer to see more thorough examples of the concept in code form. (They probably learned a lot by copy and paste coding when they first started.)

The Design Patterns in this book do contain small to medium-sized examples of PHP code to demonstrate the pattern. This dual-phased approach combines the actual conceptual explanation of the pattern for those who need that particular structure with the example-based pattern demonstration for those who are more hands-on learners.

The reference pages of this book will be more satisfying to the purists, while the case study section at the end will satisfy the code-example-hungry readers. For more information on the references pages, skip to the next section to see how they will be laid out.

Why Use Design Patterns in PHP?

PHP has a very easy beginner's learning curve with the backing of an enterprise-ready engine. Chances are that you ventured into PHP by inserting a few lines of code into an existing HTML document. Simply change the extension from `.html` to `.php`, add your quick snippet of code, deploy it to a PHP server, and you're a bona-fide PHP programmer. Up until the advent of the Zend Certified Engineer (ZCE) certification, there was no real measurement of a PHP programmer's prowess. Even after becoming a ZCE, programmers can still lack some of the essentials for developing enterprise-ready, architecturally sound application software.

As if the example in the beginning of the chapter weren't enough encouragement, more business-class players are coming on board with PHP. PHP's humble roots have left it somewhat devoid of the limelight of major enterprise-level programming languages. However, the hard work of Zend as well as the adoption of PHP by large Internet companies (such as Yahoo! and Amazon) has shown that PHP is enterprise ready. With the introduction of enterprise-level software requirements, enterprise-level methodology is to follow.

PHP now has support for a lot of the building blocks behind the concepts you're going to study. Perhaps during the era of PHP3 or PHP/FI, applying these styles of patterns may have been more difficult if not impossible. Don't get me wrong; there are always patterns in language; it's just that this book and its examples wouldn't have been nearly as useful!

Summary

This chapter discussed the prevalence of patterns in your normal programming by using an everyday programming example. By extending your understanding of patterns, you can make correlations to actual Design Patterns. Examining the realm that Design Patterns encompass, and what they do not, provided a more concise definition. Finally, the case was made for using Design Patterns in PHP by pointing out PHP's support for building base Design Patterns as well as mentioning PHP's position among some of the greater enterprise partners.

Now that you have an understanding of what Design Patterns are, let's move on to discovering what PHP already has available to help you out.