CHAPTER 7

What Are Design Patterns? Why Use Them?

Most problems we encounter as programmers have been handled time and again by others in our community. Design patterns can provide us with the means to mine that wisdom. Once a pattern becomes a common currency, it enriches our language, making it easy to share design ideas and their consequences. Design patterns simply distill common problems, define tested solutions, and describe likely outcomes. Many books and articles focus on the details of computer languages, the available functions, classes and methods. Pattern catalogs concentrate instead on how you can move on from these basics (the "what") to an understanding of the problems and potential solutions in your projects (the "why" and "how").

In this chapter, I introduce you to design patterns and look at some of the reasons for their popularity.

This chapter will cover

- *Pattern basics*: What are design patterns?
- *Pattern structure*: The key elements of a design pattern.
- *Pattern benefits*: Why are patterns worth your time?

What Are Design Patterns?

> *In the world of software, a pattern is a tangible manifestation of an organization's tribal memory.*
>
> —Grady Booch in Core J2EE Patterns

> *[A pattern is] a solution to a problem in a context.*
>
> —The Gang of Four, Design Patterns: Elements of Reusable Object-Oriented Software

As these quotations imply, a design pattern is a problem analyzed with good practice for its solution explained.

Problems tend to recur, and as web programmers, we must solve them time and time again. How are we going to handle an incoming request? How can we translate this data into instructions for our system? How should we acquire data? Present results? Over time, we answer these questions with a greater or lesser degree of elegance and evolve an informal set of techniques that we use and reuse in our projects. These techniques are patterns of design.

Design patterns inscribe and formalize these problems and solutions, making hard-won experience available to the wider programming community. Patterns are (or should be) essentially bottom-up and not top-down. They are rooted in practice and not theory. That is not to say that there isn't a strong theoretical element to design patterns (as we will see in the next chapter), but patterns are based on real-world techniques used by real programmers. Renowned pattern-hatcher Martin Fowler says that he discovers patterns, he does not invent them. For this reason, many patterns will engender a sense of déjà vu as you recognize techniques you have used yourself.

A catalog of patterns is not a cookbook. Recipes can be followed slavishly; code can be copied and slotted into a project with minor changes. You do not always need even to understand all the code used in a recipe. Design patterns inscribe *approaches* to particular problems. The details of implementation may vary enormously according to the wider context. This context might include the programming language you are using, the nature of your application, the size of your project, and the specifics of the problem.

Let's say, for example that your project requires that you create a templating system. Given the name of a template file, you must parse it and build a tree of objects to represent the tags you encounter.

You start off with a default parser that scans the text for trigger tokens. When it finds a match, it hands on responsibility for the hunt to another parser object, which is specialized for reading the internals of tags. This continues examining template data until it either fails, finishes, or finds another trigger. If it finds a trigger, it too must hand on to a specialist— perhaps an argument parser. Collectively, these components form what is known as a recursive descent parser.

So these are your participants: a `MainParser`, a `TagParser`, and an `ArgumentParser`. You create a `ParserFactory` class to create and return these objects.

Of course, nothing is easy, and you're informed late in the game that you must support more than one syntax in your templates. Now, you need to create a parallel set of parsers according to syntax: an `OtherTagParser`, `OtherArgumentParser`, and so on.

This is your problem: you need to generate a different set of objects according to circumstance, and you want this to be more or less transparent to other components in the system. It just so happens that the Gang of Four define the following problem in their book's summary page for the pattern Abstract Factory, "Provide an interface for creating families of related or dependent objects without specifying their concrete classes."

That fits nicely. It is the nature of our problem that determines and shapes our use of this pattern. There is nothing cut and paste about the solution either, as you can see in Chapter 9, in which I cover Abstract Factory.

The very act of naming a pattern is valuable; it provides the kind of common vocabulary that has arisen naturally over the years in the older crafts and professions. Such shorthand greatly aids collaborative design as alternative approaches and their various consequences are weighed and tested. When you discuss your alternative parser families, for example, you can simply tell colleagues that the system creates each set using the Abstract Factory pattern. They will nod sagely, either immediately enlightened or making a mental note to look it up later. The point is that this bundle of concepts and consequences has a handle, which makes for a handy shorthand, as I'll illustrate later in this chapter.

Finally, it is illegal, according to international law, to write about patterns without quoting Christopher Alexander, an architecture academic whose work heavily influenced the original object-oriented pattern advocates. He states in *A Pattern Language* (Oxford University Press, 1977):

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

It is significant that this definition (which applies to architectural problems and solutions) begins with the problem and its wider setting and proceeds to a solution. There has been some criticism in recent years that design patterns have been overused, especially by inexperienced programmers. This is

often a sign that solutions have been applied where the problem and context are not present. Patterns are more than a particular organization of classes and objects, cooperating in a particular way. Patterns are structured to define the conditions in which solutions should be applied and to discuss the effects of the solution.

In this book, we will focus on a particularly influential strand in the patterns field: the form described in *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1995). It concentrates on patterns in object-oriented software development and inscribes some of the classic patterns that are present in most modern object-oriented projects.

The Gang of Four book is important, because it inscribes key patterns, but also because it describes the design principles that inform and motivate these patterns. We will look at some of these principles in the next chapter.

■ **Note** The patterns described by the Gang of Four and in this book are really instances of a pattern language, that is, a catalog of problems and solutions organized together so that they complement one another, forming an interrelated whole. There are pattern languages for other problem spaces such as visual design and project management (and architecture, of course). When I discuss design patterns here, I refer to problems and solutions in object-oriented software development.

A Design Pattern Overview

At heart, a design pattern consists of four parts: the name, problem, solution, and consequences.

Name

Names matter. They enrich the language of programmers; a few short words can stand in for quite complex problems and solutions. They must balance brevity and description. The Gang of Four claims, "Finding good names has been one of the hardest parts of developing our catalog."

Martin Fowler agrees, "Pattern names are crucial, because part of the purpose of patterns is to create a vocabulary that allows developers to communicate more effectively"(*Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2002).

In *Patterns of Enterprise Application Architecture*, Martin Fowler refines a database access pattern I first encountered in *Core J2EE Patterns* by Deepak Alur, Dan Malks, and John Crupi (Prentice Hall, 2003). Fowler defines two patterns that describe specializations of the older pattern. The logic of his approach is clearly correct (one of the new patterns models domain objects, while the other models database tables, a distinction that was vague in the earlier work). It was hard to train myself to think in terms of the new patterns. I had been using the name of the original in design sessions and documents for so long that it had become part of my language.

The Problem

No matter how elegant the solution (and some are very elegant indeed), the problem and its context are the grounds of a pattern. Recognizing a problem is harder than applying any one of the solutions in a pattern catalog. This is one reason that some pattern solutions can be misapplied or overused.

Patterns describe a problem space with great care. The problem is described in brief and then contextualized, often with a typical example and one or more diagrams. It is broken down into its specifics, its various manifestations. Any warning signs that might help in identifying the problem are described.

The Solution

The solution is summarized initially in conjunction with the problem. It is also described in detail often using UML class and interaction diagrams. The pattern usually includes a code example.

Although code may be presented, the solution is never cut and paste. The pattern describes an approach to a problem. There may be hundreds of nuances in implementation. Think about instructions for sowing a food crop. If you simply follow a set of steps blindly, you are likely to go hungry come harvest time. More useful would be a pattern-based approach that covers the various conditions that may apply. The basic solution to the problem (making your crop grow) will always be the same (plant seeds, irrigate, harvest crop), but the actual steps you take will depend on all sorts of factors such as your soil type, your location, the orientation of your land, local pests, and so on.

Martin Fowler refers to solutions in patterns as "half-baked." That is, the coder must take away the concept and finish it for himself.

Consequences

Every design decision you make will have wider consequences. This should include the satisfactory resolution of the problem in question, of course. A solution, once deployed, may be ideally suited to work with other patterns. There may also be dangers to watch for.

The Gang of Four Format

As I write, I have five pattern catalogs on the desk in front of me. A quick look at the patterns in each confirms that not one uses the same structure as the others. Some are more formal than others; some are fine-grained, with many subsections; others are discursive.

There are a number of well-defined pattern structures, including the original form developed by Christopher Alexander (the Alexandrian form), the narrative approach favored by the Portland Pattern Repository (the Portland form). Because the Gang of Four book is so influential, and because we will cover many of the patterns they describe, let's examine a few of the sections they include in their patterns:

- *Intent*: A brief statement of the pattern's purpose. You should be able to see the point of the pattern at a glance.
- *Motivation*: The problem described, often in terms of a typical situation. The anecdotal approach can help make the pattern easy to grasp.
- *Applicability*: An examination of the different situations in which you might apply the pattern. While the motivation describes a typical problem, this section defines specific situations and weighs the merits of the solution in the context of each.
- *Structure/Interaction*: These sections may contain UML class and interaction diagrams describing the relationships among classes and objects in the solution.
- *Implementation*: This section looks at the details of the solution. It examines any issues that may come up when applying the technique and provides tips for deployment.

- *Sample Code*: I always skip ahead to this section. I find that a simple code example often provides a way into a pattern. The example is often chopped down to the basics in order to lay the solution bare. It could be in any object-oriented language. Of course, in this book, it will always be PHP.
- *Known Uses*: Real systems in which the pattern (problem, context, and solution) occur. Some people say that for a pattern to be genuine, it must be found in at least three publicly available contexts. This is sometimes called the "rule of three."
- *Related Patterns*: Some patterns imply others. In applying one solution, you can create the context in which another becomes useful. This section examines these synergies. It may also discuss patterns that have similarities in problem or solution and any antecedents: patterns defined elsewhere on which the current pattern builds.

Why Use Design Patterns?

So what benefits can patterns bring? Given that a pattern is a problem defined and solution described, the answer should be obvious. Patterns can help you to solve common problems. There is more to patterns, of course.

A Design Pattern Defines a Problem

How many times have you reached a stage in a project and found that there is no going forward? The chances are you must backtrack some way before starting out again.

By defining common problems, patterns can help you to improve your design. Sometimes, the first step to a solution is recognizing that you have a problem.

A Design Pattern Defines a Solution

Having defined and recognized the problem (and made certain that it is the right problem), a pattern gives you access to a solution, together with an analysis of the consequences of its use. Although a pattern does not absolve you of the responsibility to consider the implications of a design decision, you can at least be certain that you are using a tried-and-tested technique.

Design Patterns Are Language Independent

Patterns define objects and solutions in object-oriented terms. This means that many patterns apply equally in more than one language. When I first started using patterns, I read code examples in C++ and Smalltalk and deployed my solutions in Java. Others transfer with modifications to the pattern's applicability or consequences but remain valid. Either way, patterns can help you as you move between languages. Equally, an application that is built on good object-oriented design principles can be relatively easy to port between languages (although there are always issues that must be addressed).

Patterns Define a Vocabulary

By providing developers with names for techniques, patterns make communication richer. Imagine a design meeting. I have already described my Abstract Factory solution, and now I need to describe my strategy for managing the data the system compiles. I describe my plans to Bob:

> ME: I'm thinking of using a Composite.
>
> BOB: I don't think you've thought that through.

OK, Bob didn't agree with me. He never does. But he knew what I was talking about, and therefore why my idea sucked. Let's play that scene through again without a design vocabulary.

> ME: I intend to use a tree of objects that share the same type. The type's interface will provide methods for adding child objects of its own type. In this way, we can build up complex combinations of implementing objects at runtime.
>
> BOB: Huh?

Patterns, or the techniques they describe, tend to interoperate. The Composite pattern lends itself to collaboration with Visitor:

> ME: And then we can use Visitors to summarize the data.
>
> BOB: You're missing the point.

Ignore Bob. I won't describe the tortuous nonpattern version of this; I will cover Composite in Chapter 10 and Visitor in Chapter 11.

The point is that without a pattern language, we would still use these techniques. They *precede* their naming and organization. If patterns did not exist, they would evolve on their own anyway. Any tool that is used sufficiently will eventually acquire a name.

Patterns Are Tried and Tested

So if patterns document good practice, is naming the only truly original thing about pattern catalogs? In some senses, that would seem to be true. Patterns represent best practice in an object-oriented context. To some highly experienced programmers, this may seem an exercise in repackaging the obvious. To the rest of us, patterns provide access to problems and solutions we would otherwise have to discover the hard way.

Patterns make design accessible. As pattern catalogs emerge for more and more specializations, even the highly experienced can find benefits as they move into new aspects of their fields. A GUI programmer can gain fast access to common problems and solutions in enterprise programming, for example. A web programmer can quickly chart strategies for avoiding the pitfalls that lurk in PDA and smart phone projects.

Patterns Are Designed for Collaboration

By their nature, patterns should be generative and composable. This means that you should be able to apply one pattern and thereby create conditions suitable for the application of another. In other words, in using a pattern you may find other doors opened for you.

Pattern catalogs are usually designed with this kind of collaboration in mind, and the potential for pattern composition is always documented in the pattern itself.

Design Patterns Promote Good Design

Design patterns demonstrate and apply principles of object-oriented design. So a study of design patterns can yield more than a specific solution in a context. You can come away with a new perspective on the ways that objects and classes can be combined to achieve an objective.

PHP and Design Patterns

There is little in this chapter that is specific to PHP, which is characteristic of our topic to some extent. Many patterns apply to many object-capable languages with few or no implementation issues.

This is not always the case, of course. Some enterprise patterns work well in languages in which an application process continues to run between server requests. PHP does not work that way. A new script execution is kicked off for every request. This means that some patterns need to be treated with more care. Front Controller, for example, often requires some serious initialization time. This is fine when the initialization takes place once at application startup but more of an issue when it must take place for every request. That is not to say that we can't use the pattern; I have deployed it with very good results in the past. We must simply ensure that we take account of PHP-related issues when we discuss the pattern. PHP forms the context for all the patterns that this book examines.

I referred to object-capable languages earlier in this section. You can code in PHP without defining any classes at all (although with PEAR's continuing development you will probably manipulate objects to some extent). Although this book focuses almost entirely on object-oriented solutions to programming problems, it is not a broadside in an advocacy war. Patterns and PHP can be a powerful mix, and they form the core of this book; they can, however, coexist quite happily with more traditional approaches. PEAR is an excellent testament to this. PEAR packages use design patterns elegantly. They tend to be object-oriented in nature. This makes them more, not less, useful in procedural projects. Because PEAR packages are self-enclosed and hide their complexity behind clean interfaces, they are easy to stitch into any kind of project.

Summary

In this chapter, I introduced design patterns, showed you their structure (using the Gang of Four form), and suggested some reasons why you might want to use design patterns in your scripts.

It is important to remember that design patterns are not snap-on solutions that can be combined like components to build a project. They are suggested approaches to common problems. These solutions