Chapter 1

Congratulations, Your Problem Has Already Been Solved

In This Chapter

- Introducing design patterns
- Knowing how design patterns can help
- Extending object-oriented programming
- Taking a look at some specific design patterns

As a programmer, you know how easy it can be to get lost in the details of what you're doing. And when you lose the overview, you don't plan effectively, and you lose the bigger picture. When that happens, the code you're writing in the trenches ends up working fine for a while, but unless you understand the bigger picture, that code really is a specialized solution to a particular problem.

And the heck of it is that problems rarely stay solved after you've handled them once. Developers typically regard their work as tackling individual problems by writing code and solving those problems. But the truth is that in any professional environment, developers almost always end up spending a lot more time on maintenance and adapting code to new situations than writing entirely new code.

So if you consider it, it doesn't make sense to think in terms of Band-Aid fixes to remedy the problems you face because you'll end up spending a great deal of time putting out fires and trying to extend code written for a specific problem so that it can handle other cases as well. It makes more sense to get a little overview on the process of code design and maintenance.

The idea behind this book is to familiarize you with a set of *design patterns* to simplify the programming process automatically. The plan is to get you some overview automatically, no additional effort required. A design pattern is a tested solution to a standard programming problem. When you're familiar with the design patterns presented in this book, you can face a programming issue and — Bam! — a solution will come to you more quickly. Instead of banging your head against the wall in desperation, you'll say, "What I need here is the Factory pattern." Or the Observer pattern. Or the Adapter pattern.

That's *not* to say, as some design books seem to suggest, that you should spend a great deal of time dealing with abstractions and planning before tackling a project. Adding unneeded layers of abstraction to the programming process is not a burden any programmer needs.

The whole beauty here is simply that someone has already faced the problem you're facing and has come up with a solution that implements all kinds of good design. And being familiar with design patterns can make the design process all but automatic for you.

How do you turn into a software design expert, the envy of all, with hardly any work on your part? Easy. You read this book and get familiar with the patterns I cover in depth. You don't have to memorize anything; you just get to know those patterns. Then when you encounter a real-world issue that matches one of those patterns, something deep inside you says, "Hey! That looks like you need the Iterator pattern." And all you have to do is look up that pattern in this book and leaf through the examples to know what to do. So without further ado, this chapter gets you started on your tour of these handy, helpful design patterns.

Just Find the Pattern that Fits

The charm of knowing about design patterns is that it makes your solution easily reusable, extendable, and maintainable. When you're working on a programming problem, the tendency is to program to the problem, not in terms of reuse, extensibility, maintainability, or other good design issues. And that's where most programmers should be putting in more work because they end up spending far more time on such issues than on solving the original problem in the long run.

For example, you may want to create Java objects that can, say, parse XML documents. And to do that, you create a proprietary parser class, and then instantiate objects of that class to create XML parser objects as needed. So far, so good, you think. But it turns out that there are dozens of XML parser classes out there written in Java that people are attached to, and they might want to use the special features of the XML parser class they're used to. If you'd used the Factory pattern, you would have written code that could use any XML parser class to create parser objects (instead of hardcoding a proprietary solution). And your code would be extendable, reusable, and easier to maintain.

In other words, *design patterns are solutions to programming problems that automatically implement good design techniques.* Someone has already faced the issues you're facing, solved them, and is willing to show you what the best techniques are. All without a lot of memorization on your part; all you have to do is recognize which design pattern fits which situation and lock it into place.

Sweet.

Enter the Gang of Four Book

The set of 23 standard design patterns was published by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides in their seminal 1995 book *Design Patterns: Elements of Reusable Object-Oriented Software* (Pearson Education, Inc. Publishing as Pearson Addison Wesley). They've come to be known in programming circles as the Gang of Four, or, more popularly, GoF.

A lot of water has passed under the bridge since the GoF book appeared, and it turns out that some of the original 23 patterns were not used as much as some of the others. You see them all in this book, but I emphasize the patterns that are used the most — and that includes some new, non-GoF patterns in Chapter 11 that have appeared since the GoF book debuted.

It's important to realize that there is more going on here than just memorizing design patterns. There are also specific design insights about object-oriented programming that are just as important, and I talk about them throughout the book. OOP is a terrific advance in programming. But too many programmers

blindly apply its design strategies without a lot of insight, and that can cause as many problems as it fixes. A large part of understanding how to work with design patterns involves understanding the OOP insights behind them — encapsulating what changes most, for example, or knowing when to convert from is-a inheritance to has-a composites (see Chapter 2 for more on what these terms mean) — and I talk about those insights a lot.

Getting Started: The Mediator Pattern

Figure 1-1 provides an example design pattern, the Mediator pattern, that shows what design patterns can do for you. Say that you've got a four-page Web site that lets users browse a store and make purchases. As things stand, the user can move from page to page. But there's a problem — the code in each page has to know when to jump to a new page as well as how to activate the new page. You've got a lot of possible connections and a lot of duplicate code in the various pages.

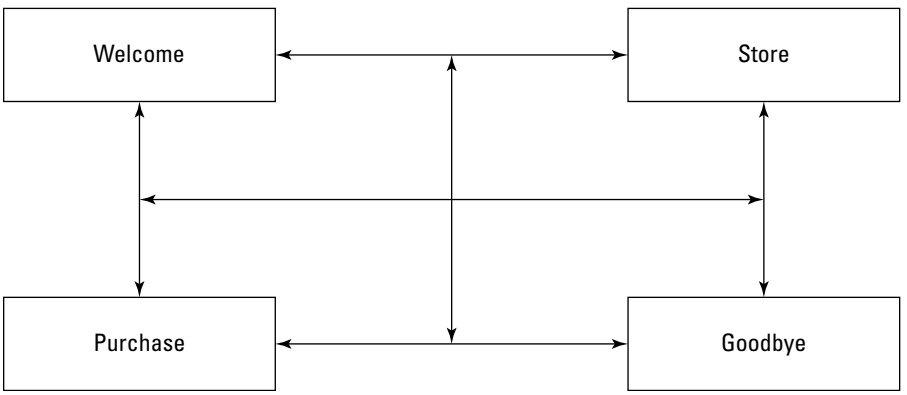

Figure 1-1: In this illustration of the basic navigation of a typical online store, the Mediator pattern mediates between objects.

You can use a mediator here to encapsulate all the navigation code out of the separate pages and place it into a mediator object instead. From then on, each page just has to report any change of state to the mediator, and the mediator knows what page to send the user to, as shown in Figure 1-2.

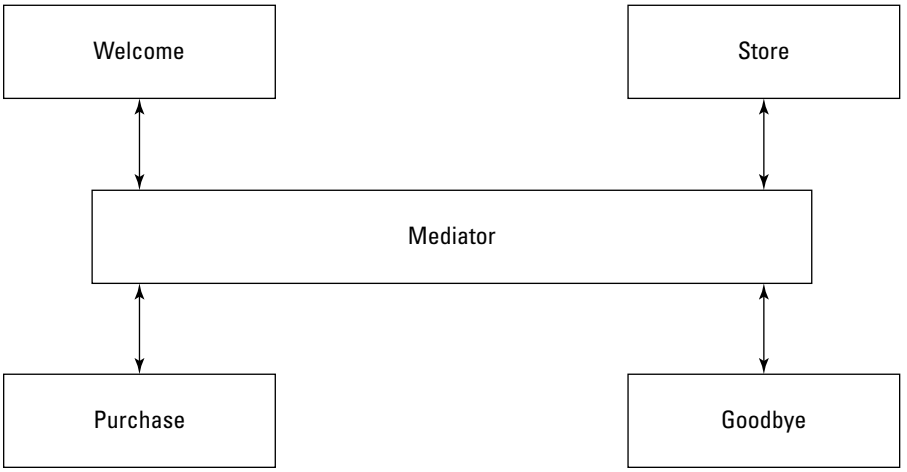

Figure 1-2: The mediator directs Web site traffic.

You can build the mediator to deal with the internals of each page so the various pages don't have to know the intimate details of the other pages (such as which methods to call). And when it's time to modify the navigation code that takes users from page to page, that code is all collected in one place, so it's easier to modify.

Adapting to the Adapter Pattern

Here's another design pattern, the Adapter pattern. Say that for a long time you've been supplied with a stream of objects and fit them into code that can handle those objects, as shown in Figure 1-3.

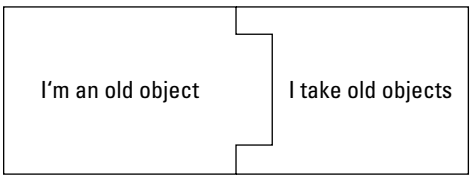

Figure 1-3: Everything seems to be working here.

But now say there's been an upgrade. The code isn't expecting those old objects anymore, only new objects, and the old objects aren't going to fit into the new code, as shown in Figure 1-4.

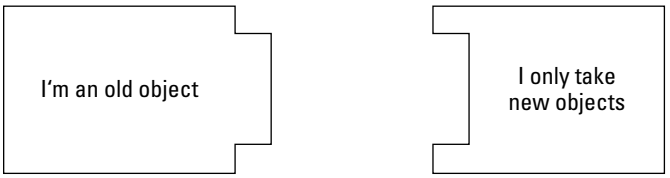

Figure 1-4: This isn't going to work.

If you can't change how the old objects are generated in this case, the Adapter pattern has a solution — create an adapter object that exposes the interface expected by the old object and the new code, and use the adapter to let the old object fit into the new code, as shown in Figure 1-5.

Figure 1-5: Old objects work with new objects via an adapter.

Problem solved. Who says design patterns are hard?

Standing In for Other Objects with the Proxy Pattern

Here's another pattern, the Proxy design pattern. Say that you've got some local code that's used to dealing with a local object as shown in Figure 1-6:

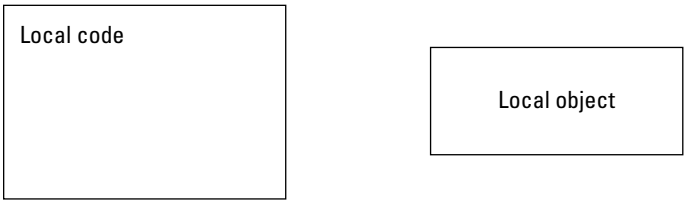

Figure 1-6: Code and an object, working together in the same neighborhood.

But now say that you want to deal with some remote object, somewhere else in the world. How can you make the local code think it's dealing with a local object still when in fact it's working with that remote object?

With a *proxy.* A proxy is a stand-in for another object that makes the local code think it's dealing with a local object. Behind the scenes, the proxy connects to the remote object, all the while making the local code believe it's working with a local object, as you can see in Figure 1-7.

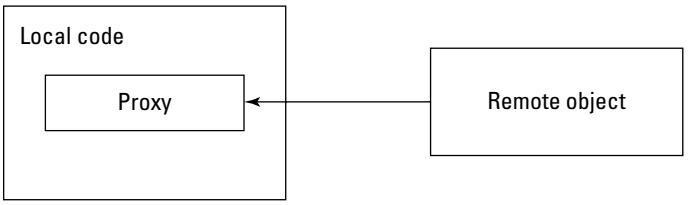

Figure 1-7: Trick your local code and remote object into working together.

You see the Proxy pattern at work in Chapter 9 in an example that lets you connect to a remote object over the Internet anywhere in the world, with just a few lines of code.

Taking a Look at the Observer Pattern

You're most likely familiar with a number of the patterns in this book, such as the Observer pattern. This pattern, like many others, is already implemented in Java.

The Observer design pattern is about passing notifications around to update a set of objects when some important event has occurred. You can add new observer objects at runtime and remove them as needed. When an event occurs, all registered observers are notified. Figure 1-8 shows how it works; an observer can register itself with the subject.

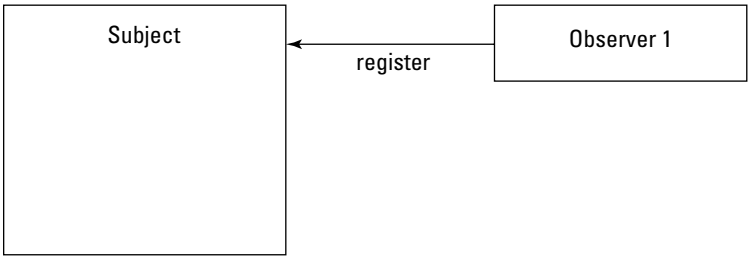

Figure 1-8: The Observer pattern lets observers register with subjects.

And another observer, Observer 2, can register itself as well, as shown in Figure 1-9.

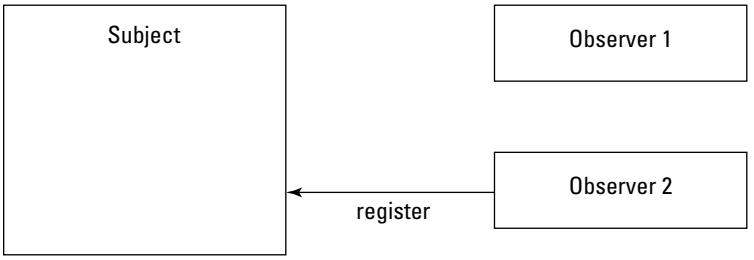

Figure 1-9: More than one observer can register with a subject.

Now the subject is keeping track of two observers. When an event occurs, the subject notifies both observers. (See Figure 1-10.)

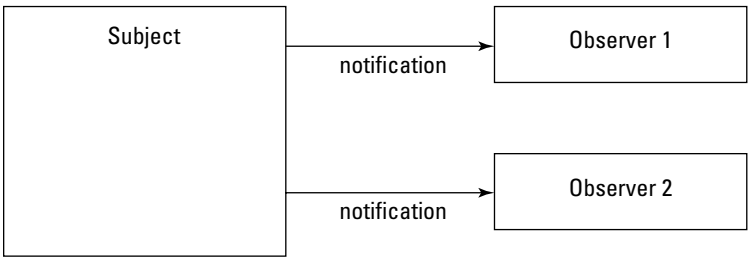

Figure 1-10: When events occur in the subject, registered observers are notified.

Does this sound familiar in Java? If Java event listeners came to mind, you'd be right. Event listeners can register with objects like push buttons or windows to be informed of any events that occur.

That's just one example of the kind of design pattern you've probably already seen implemented in Java. When such examples come up, I include Java example code showing how a particular design pattern is already built into Java. The example code might ring a few bells.

This book is written to be easy to use and understand. You're not going to see chalkboard diagrams of complex abstractions that you have to plow through. The chapters in this book are aimed at programmers, to be useful for programmers; even if you don't read all of them, you're going to benefit. The design insights and patterns covered here are becoming standard throughout the programming world, and they are helpful on an everyday level. Hopefully, the next time you face a tough coding issue, you'll suddenly find yourself saying: Aha! this is a job for the Facade pattern.