Preface

A former colleague of mine used to say that thick books about design patterns were evidence of an inadequate programming language. What he meant was that, because design patterns are the common idioms of code, a good programming language should make them very easy to implement. An ideal language would so thoroughly integrate the patterns that they would almost disappear from sight.

To take an extreme example, in the late 1980s I worked on a project that produced object-oriented code in C. Yes, C, not C++. We pulled off this feat by having each "object" (actually a C structure) point to a table of function pointers. We operated on our "objects" by chasing the pointer to the table and calling functions out of the table, thereby simulating a method call on an object. It was awkward and messy, but it worked. Had we thought of it, we might have called this technique the "object-oriented" pattern. Of course, with the advent of C++ and then Java, our object-oriented pattern disappeared, absorbed so thoroughly into the language that it vanished from sight. Today, we don't usually think of object orientation as a pattern—it is too easy.

But many things are still not easy enough. The justly famous Gang of Four book (*Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides) is required reading for every software engineer today. But actually implementing many of the patterns described in *Design Patterns* with the languages in widespread use today (Java and C++ and perhaps C#) looks and feels a lot like my 1980s-vintage handcrafted object system. Too painful. Too verbose. Too prone to bugs.

The Ruby programming language takes us a step closer to my old friend's ideal, a language that makes implementing patterns so easy that sometimes they fade into the background. Building patterns in Ruby is easier for a number of reasons:

- Ruby is dynamically typed. By dispensing with static typing, Ruby dramatically reduces the code overhead of building most programs, including those that implement patterns.
- Ruby has code closures. It allows us to pass around chunks of code and associated scope without having to laboriously construct entire classes and objects that do nothing else.
- Ruby classes are real objects. Because a class in Ruby is just another object, we can do any of the usual runtime things to a Ruby class that we can do to any other object: We can create totally new classes. We can modify existing classes by adding or deleting methods. We can even clone a class and change the copy, leaving the original alone.
- Ruby has an elegant system of code reuse. In addition to supporting garden-variety inheritance, Ruby allows us to define mixins, which are a simple but flexible way to write code that can be shared among several classes.

All of this makes code in Ruby compressible. In Ruby, as in Java and C++, you can implement very sophisticated ideas, but with Ruby it becomes possible to hide the details of your implementations much more effectively. As you will see on the pages that follow, many of the design patterns that require many lines of endlessly repeated boilerplate code in traditional static languages require only one or two lines in Ruby. You can turn a class into a singleton with a simple `include Singleton` command. You can delegate as easily as you can inherit. Because Ruby enables you to say more interesting things in each line of code, you end up with less code.

This is not just a question of keyboard laziness; it is an application of the DRY (Don't Repeat Yourself) principle. I don't think anyone today would mourn the passing of my old object-oriented pattern in C. It worked for me, but it made me work for it, too. In the same way, the traditional implementations of many design patterns work, but they make you work, too. Ruby represents a real step forward in that you become able to do work only once and compress it out of the bulk of your code. In short, Ruby allows you to concentrate on the real problems that you are trying to solve instead of the plumbing. I hope that this book will help you see how.

Who Is This Book For?

Simply put, this book is intended for developers who want to know how to build significant software in Ruby. I assume that you are familiar with object-oriented programming, but you don't really need any knowledge of design patterns—you can pick that up as you go through the book.

You also don't need a lot of Ruby knowledge to read this book profitably. You will find a quick introduction to the language in Chapter 2, and I try to explain any Ruby-specific language issues as we go.

How Is This Book Organized?

This book is divided into three parts. First come a couple of introductory chapters, starting with the briefest outline of the history and background of the whole design patterns movement, and ending with a quick tour of the Ruby language at the "just enough to be dangerous" level.

Part 2, which takes up the bulk of these pages, looks at a number of the original Gang of Four patterns from a Ruby point of view. Which problem is this pattern trying to solve? What does the traditional implementation of the pattern—the implementation given by the Gang of Four—look like in Ruby? Does the traditional implementation make sense in Ruby? Does Ruby provide us with any alternatives that might make solving the problem easier?

Part 3 of this book looks at three patterns that have emerged with the introduction and expanded use of Ruby.

A Word of Warning

I cannot sign my name to a book about design patterns without repeating the mantra that I have been muttering for many years now: Design patterns are little spring-loaded solutions to common programming problems. Ideally, when the appropriate problem comes along, you should trigger the design pattern and your problem is solved. It is that first part—the bit about waiting for the appropriate problem to come along—that some engineers have trouble with. You cannot say that you are correctly applying a design pattern *unless you are confronting the problem that the pattern is supposed to solve.*

The reckless use of every design pattern on the menu to solve nonexistent problems has given design patterns a bad name in some circles. I would contend that Ruby makes it easier to write an adapter that uses a factory method to get a proxy to the builder, which then creates a command, which will coordinate the operation of adding two plus two. Ruby will make that process easier, but even in Ruby it will not make any sense.

Nor can you look at program construction as a simple process of piecing together some existing design patterns in new combinations. Any interesting program will always have unique sections, bits of code that fit that specific problem perfectly and no other. Design patterns are meant to help you recognize and solve the common problems that arise repeatedly when you are building software. The advantage of design patterns is that they let you rapidly wing your way past the problems that someone has already solved, so that you can get on to the hard stuff, the code that is unique to your situation. Design patterns are not the universal elixir, the magic potion that will fix all of your design problems. They are simply one technique—albeit a very useful technique—that you can use to build programs.

About the Code Style Used in This Book

One thing that makes programming in Ruby so pleasant is that the language tries to stay out of your way. If there are several sensible ways of saying something, Ruby will usually support them all:

```
# One way to say it

if (divisor == 0)
  puts 'Division by zero'
end

# And another

puts 'Division by zero' if (divisor == 0)

# And a third

(divisor == 0) && puts 'Division by zero'
```

Ruby also tries not to insist on syntax for syntax's sake. Where possible, it will let you omit things when the meaning is clear. For example, you can usually omit the parentheses around the argument list when calling a method:

```
puts('A fine way to call puts')
puts 'Another fine way to call puts'
```

You can even forget the parentheses when you are defining the argument list of a method and around the conditional part of an `if` statement:

```
def method_with_3_args a, b, c
  puts "Method 1 called with #{a} #{b} #{c}"
  if a == 0
    puts 'a is zero'
  end
end
```

The trouble with all of these shortcuts, convenient as they are in writing real Ruby programs, is that when liberally used, they tend to confuse beginners. Most programmers who are new to Ruby are going to have an easier time with

```
if file.eof?
  puts( 'Reached end of file' )
end
```

or even

```
puts 'Reached end of file' if file.eof?
```

than with

```
file.eof? || puts('Reached end of file')
```

Because this book is more about the deep power and elegance of Ruby than it is about the details of the language syntax, I have tried to strike a balance between making my examples actually look like real Ruby code on the one hand while still being beginner friendly on the other hand. In practice, this means that while I take advantage of some obvious shortcuts, I have deliberately avoided the more radical tricks. It is not that I am unaware of, or disapprove of, the Ruby syntactical shorthand. It is just that I am more interested getting the conceptual elegance of the language across to readers who are new to Ruby. There will be plenty of time to learn the syntactical shortcuts after you have fallen hopelessly in love with the language.