Extracted from:

Metaprogramming Ruby 2

Program Like the Ruby Pros

The Pragmatic Bookshelf

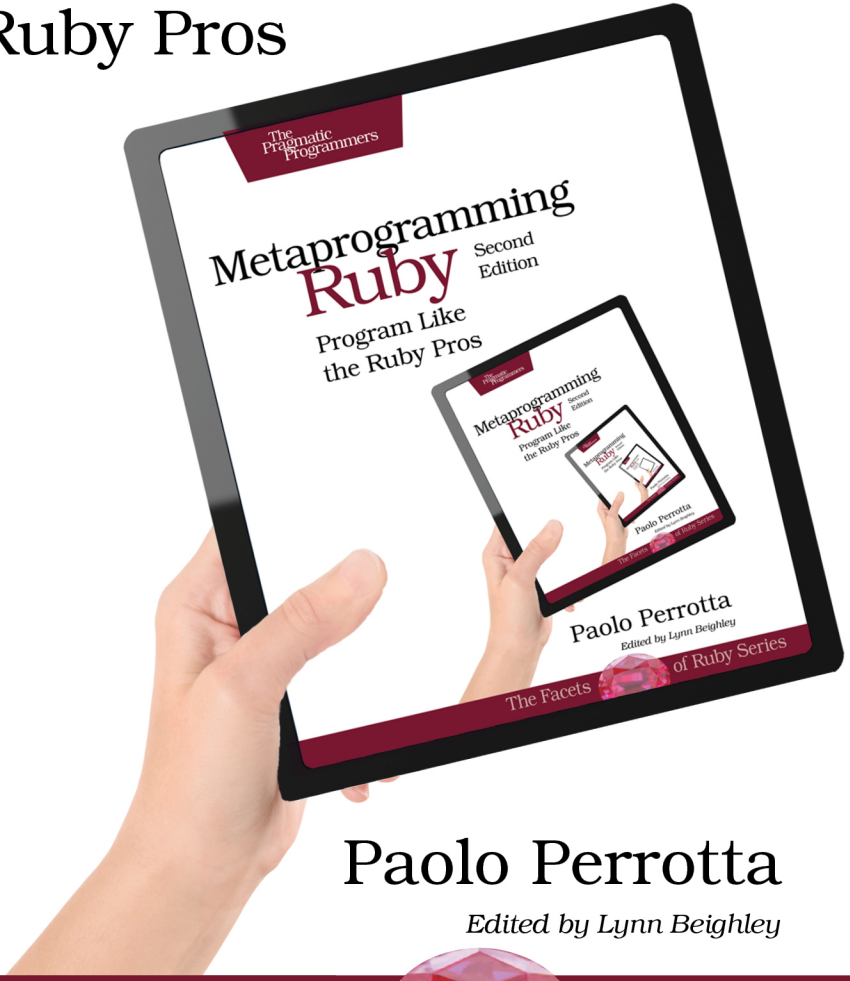
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Metaprogramming Ruby

Second Edition

Program Like the Ruby Pros



Paolo Perrotta

Edited by Lynn Beighley

The Facets of Ruby Series

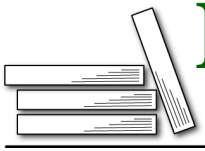
Metaprogramming Ruby 2

Program Like the Ruby Pros

Paolo Perrotta

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Lynn Beighley (editor)
Potomac Indexing, LLC (indexer)
Cathleen Small (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-94122-212-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2014

I was thirteen, and I was tired of hanging out at the local toy shop to play Intellivision games. I wanted my own videogame console. I'd been bugging my parents for a while, with no success.

Then I found an alternative: I could play games on a computer as well. So I asked my parents to buy me one of those new 8-bit computers—you know, to learn useful stuff. My dad agreed, and my mom took me to the shop and bought me a Sinclair ZX Spectrum.

Mom, Dad… Here is something that I should've told you more often in my life: thank you. This book is dedicated to the two of you. I'm hoping it will make you proud, just like your once-kid is proud of you. And while I'm here, I have something to confess about that life-changing day thirty years ago: I didn't really want to learn stuff. I just wanted to play.

In fact, that's what I've been doing all these years.

Will write code that writes code that writes code for food.

➤ *Martin Rodgers*

Introduction

Metaprogramming…it sounds cool! It sounds like a design technique for high-level enterprise architects or a faddish buzzword that has found its way into press releases.

In fact, far from being an abstract concept or a bit of marketing-speak, metaprogramming is a collection of down-to-earth, pragmatic coding techniques. It doesn't just sound cool; it *is* cool. Here are some things you can do with metaprogramming in the Ruby language:

- Say you want to write a Ruby program that connects to an external system —maybe a web service or a Java program. With metaprogramming, you can write a wrapper that takes *any* method call and routes it to the external system. If somebody adds methods to the external system later, you don't have to change your Ruby wrapper; the wrapper will support the new methods right away. That's magic.
- Maybe you have a problem that would best be solved with a programming language that's specific to that problem. You could go to the trouble of writing your own language, custom parser and all. Or you could just use Ruby, bending its syntax until it looks like a specific language for your problem. You can even write your own little interpreter that reads code written in your Ruby-based language from a file.
- You can aggressively remove duplication from your Ruby code while keeping it elegant and clean. Imagine twenty methods in a class that all look the same. How about defining all those methods at once, with just a few lines of code? Or maybe you want to call a sequence of similarly named methods. How would you like a single short line of code that calls all the methods whose names match a pattern—like, say, all methods that begin with *test*?
- You can stretch and twist Ruby to meet your needs, rather than adapt to the language as it is. For example, you can enhance any class (even a core class like `Array`) with that method you miss so dearly, you can wrap

logging functionality around a method that you want to monitor, you can execute custom code whenever a client inherits from your favorite class…the list goes on. You are limited only by your own, undoubtedly fertile, imagination.

Metaprogramming gives you the power to do all these things. Let's see how this book will help you learn about it.

About This Book

Part I, *Metaprogramming Ruby*, is the core of the book. Chapter 1, *The M Word*, on page ?, walks you through the basic idea behind metaprogramming. The following chapters tell the story of a week in the life of a newly hired Ruby programmer and his or her more experienced colleague:

- Ruby's object model is the land in which metaprogramming lives. Chapter 2, *Monday: The Object Model*, on page ?, provides a map to this land. This chapter introduces you to the most basic metaprogramming techniques. It also reveals the secrets behind Ruby classes and *method lookup*, the process by which Ruby finds and executes methods.
- Once you understand method lookup, you can do some fancy things with methods: you can create methods at runtime, intercept method calls, route calls to another object, or even accept calls to methods that don't exist. All these techniques are explained in Chapter 3, *Tuesday: Methods*, on page ?.
- Methods are members of a larger family also including entities such as blocks and lambdas. Chapter 4, *Wednesday: Blocks*, on page ?, is your field manual for everything related to these entities. It also presents an example of writing a *domain-specific language*, a powerful conceptual tool that Ruby coders tend to love. This chapter also comes with its own share of tricks, explaining how you can package code and execute it later or how you can carry variables across scopes.
- Speaking of scopes, Ruby has a special scope that deserves a close look: the scope of class definitions. Chapter 5, *Thursday: Class Definitions*, on page ?, talks about this scope and introduces you to some of the most powerful weapons in a metaprogrammer's arsenal. It also introduces *singleton classes*, the last concept you need to make sense of Ruby's most perplexing features.
- Finally, Chapter 6, *Friday: Code That Writes Code*, on page ?, puts it all together through an extended example that uses techniques from all the

previous chapters. The chapter also rounds out your metaprogramming training with two new topics: the somewhat controversial eval method and the callback methods that you can use to intercept events in the object model.

Part II of the book, *Metaprogramming in Rails*, is a case study in metaprogramming. It contains short chapters that focus on different areas of Rails, the flagship Ruby framework. By looking at Rails' source code, you'll see how master Ruby coders use metaprogramming in the real world to develop great software, and you'll also understand how some metaprogramming techniques evolved in the last few years.

Three appendixes close the book. Appendix 1, *Common Idioms*, on page ?, is a grab-bag of common techniques that are not explained anywhere else in the book. Appendix 2, *Domain-Specific Languages*, on page ?, is a quick look at a programming approach that is common among Ruby developers. Appendix 3, *Spell Book*, on page ?, is a catalog of all the spells in the book, complete with code examples.

"Wait a minute," I can hear you saying. "What the heck are *spells*?" Oh, right, sorry. Let me explain.

Spells

This book contains a number of metaprogramming techniques that you can use in your own code. Some people might call these *patterns* or maybe *idioms*. Neither of these terms is very popular among Rubyists, so I'll call them *spells* instead. Even if there's nothing magical about them, they *do* look like magic spells to Ruby newcomers.

You'll find references to spells everywhere in the book. I reference a spell with the convention *Class Macro (?)* or *String of Code (?)*, for example. The number in parentheses is the page where the spell receives a name. If you need a quick reference to a spell, you'll find it in Appendix 3, *Spell Book*, on page ?.

Quizzes

Every now and then, this book also throws a quiz at you. You can skip these quizzes and just read the solution, but you'll probably want to solve them on your own just because they're fun.

Some quizzes are traditional coding exercises; others require you to get off your keyboard and think. All include a solution, but most quizzes have more than one possible answer. Please, feel free to go wild and experiment.

Notation Conventions

This book is chock full of code examples. To show you that a line of code results in a value, I print that value as a comment on the same line:

```
-1.abs          # => 1
```

If a code example is supposed to print a result rather than return it, I show that result after the code:

```
puts 'Testing... testing...'

Testing... testing...
```

In most cases, the text uses the same code syntax that Ruby uses: `MyClass.my_method` is a class method, `MyClass::MY_CONSTANT` is a constant defined within a class, and so on. There are a couple of exceptions to this rule. First, I identify instance methods with the *hash* notation, like the Ruby documentation does (`MyClass#my_method`). This is useful to distinguish class methods and instance methods. Second, I use a hash prefix to identify singleton classes (`#MySingletonClass`).

Ruby has a flexible syntax, so few universal rules exist for things like indentation and the use of parentheses. Programmers tend to adopt the syntax that they find most readable in each specific case. In this book, I try to follow the most common conventions. For example, I skip parentheses when I call a method without parameters (as in `my_string.reverse`), but I tend to use parentheses when I pass parameters (as in `my_string.gsub("x", "y")`).

Some of the code in this book comes straight from existing open-source libraries. Some of these are standard Ruby libraries, so you should already have them. You can install the others with the `gem` command. For example, if I show you a piece of code from Builder 3.2.2, and you want to install the entire library to explore its source by yourself, then you can use `gem install builder -v 3.2.2`. Be aware of the version, because the code might have changed in more recent versions of Builder.

To avoid clutter (and make the code easier to understand in isolation), I'll sometimes take the liberty of editing the original code slightly. However, I'll do my best to keep the spirit of the original source intact.

Unit Tests

This book follows two developers as they go about their day-to-day work. As the story unfolds, you may notice that these two characters rarely write tests. Does this book condone untested code?

Please rest assured that it doesn't. In fact, the original draft of this book included unit tests for all code examples. In the end, I found that those tests distracted from the metaprogramming techniques that are the meat of the book, so the tests fell on the cutting-room floor. This doesn't mean you shouldn't write tests for your own metaprogramming endeavors.

On those occasions where I did show test code in this book, I used the test-unit library. Until Ruby 2.1, test-unit was a standard library. From Ruby 2.2 onward, you need to install it as a gem, with the command gem install test-unit.

Ruby Versions

Ruby is continuously changing and improving. However, this very fluidity can be problematic when you try a piece of code on the latest version of the language, only to find that it doesn't work anymore. This is not overly common, but it can happen with metaprogramming, which pushes Ruby to its limits.

This book is written for Ruby 2. As I write, Ruby 2.1 is the most recent stable version of the language, and it's mostly compatible with Ruby 2.0. Some people still run older versions of Ruby, which miss a few important features from 2.x—notably, Refinements and Module#prepend. In the text, I'll refer to Ruby 2.x, and I'll tell you which features were introduced either in Ruby 2.1 or in Ruby 2.0.

When I talk about Ruby versions, I'm talking about the "official" interpreter (sometimes called MRI for *Matz's Ruby Interpreter*[1]). There are many alternate Ruby implementations. Two of the most popular ones are JRuby, which runs on the Java Virtual Machine,[2] and Rubinius.[3] Alternate implementations usually take a few versions to catch up with MRI — so if you use one of them, be aware that some of the examples in this book might not yet work on your interpreter.

Book Editions

The first edition of this book focused on Ruby 1.8, which has since been deprecated. I updated the text to reflect the new features in Ruby, especially the ones that have been introduced by Ruby 2.x.

The chapters in Part II use the Rails source code as a source of examples. Rails has changed a lot since the first edition, so these chapters are almost a complete rewrite of the first edition's content.

1. http://www.ruby-lang.org
2. http://jruby.codehaus.org
3. http://rubini.us/

Apart from the changes in the language and the libraries, some of my personal opinions also changed since the first edition of this book. I learned to be wary of some techniques, such as *Ghost Methods (?)*, and fonder of others, such as *Dynamic Methods (?)*. Parts of the new text reflect these changes of heart.

Finally, this second edition is a general cleanup of the first edition's text. I updated many examples that were using gems and source code that have been forgotten or changed since the previous book; I added a few spells and removed a few others that don't seem very relevant anymore; I toned down the "story" in the text when it was adding too many words to long technical explanations; and I went through every sentence again, fixing things that needed fixing and addressing errata and suggestions from the readers. Whether you're a new reader or a fan of the first edition, I hope you like the result.

About You

Most people consider metaprogramming an advanced topic. To play with the constructs of a Ruby program, you have to know how these constructs work in the first place. How do you know whether you're enough of an "advanced" Rubyist to deal with metaprogramming? Well, if you can understand the code in the very first chapter without much trouble, then you are well equipped to move forward.

If you're not confident about your skills, you can take a simple self-test. Which kind of code would you write to iterate over an array? If you thought about the `each` method, then you know enough Ruby to follow the ensuing text. If you thought about the `for` keyword, then you're probably new to Ruby. In the second case, you can still embark on this metaprogramming adventure—just take an introductory Ruby text along with you, or take the excellent interactive tutorial at the *Try Ruby!* site.[4]

Are you on board, then? Great! Let's start.

4. http://tryruby.org