Extracted from:

Metaprogramming Ruby

The Pragmatic Programmers

Metaprogramming Ruby

Paolo Perrotta

Edited by Jill Steinberg

The Facets of Ruby Series

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://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-47-6

ISBN-13: 978-1-934356-47-0

Printed on acid-free paper.

P1.0 printing, January 2010

Version: 2010-1-29

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 fashionable 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 of the 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 be best 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 remove duplication from your Ruby program at a level that Java programmers can only dream of. Let's say you have twenty methods in a class, and they 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 what it looks like.

The "M" Word

You're probably expecting a definition of metaprogramming right from the start. Here's one for you:

Metaprogramming is writing code that writes code.

We'll get to a more precise definition in a short while, but this one will do for now. What do I mean by "code that writes code," and how is that useful in your daily work? Before I answer those questions, let's take a step back and look at programming languages themselves.

Ghost Towns and Marketplaces

Think of your source code as a world teeming with vibrant citizens: variables, classes, methods, and so on. If you want to get technical, you can call these citizens *language constructs*.

In many programming languages, language constructs behave more like ghosts than fleshed-out citizens: you can see them in your source code, but they disappear before the program runs. Take C++, for example. Once the compiler has finished its job, things like *variable* and *method* have lost their concreteness; they are just locations in memory. You can't ask a class for its instance methods, because by the time you ask the question, the class has faded away. In languages like C++, runtime is an eerily quiet place—a ghost town.

In other languages, such as Ruby, runtime looks more like a busy marketplace. Most language constructs are still there, buzzing all around. You can even walk up to a construct and ask it questions about itself. This is called *introspection*. Let's watch introspection in action.

Code Generators and Compilers

In metaprogramming, you write code that writes code. But isn't that what code generators and compilers do? For example, you can write annotated Java code and then use a code generator to output XML configuration files. In a broad sense, this XML generation is an example of metaprogramming. In fact, many people think about code generation when the "m" word comes up.

This particular brand of metaprogramming implies that you use a program to generate or otherwise manipulate a second, distinct program—and then you run the second program. After you run the code generator, you can actually read the generated code and (if you want to test your tolerance for pain) even modify it by hand before you finally run it. This is also what happens under the hood with C++ templates: the compiler turns your templates into a regular C++ program before compiling them, and then you run the compiled program.

In this book, I'll stick to a different meaning of *metaprogramming*, focusing on code that manipulates itself at runtime. Only a few languages can do that effectively, and Ruby is one of them. You can think of this as *dynamic* metaprogramming to distinguish it from the *static* metaprogramming of code generators and compilers.

Introspection

Take a look at this code:

Download introduction/introspection.rb

```
class Greeting
  def initialize(text)
    @text = text
  end

  def welcome
    @text
  end
end

my_object = Greeting.new("Hello")
```

I defined a Greeting class and created a Greeting object. I can now turn to the language constructs and ask them questions.

```
my_object.class                              # => Greeting
my_object.class.instance_methods(false)      # => [:welcome]
my_object.instance_variables                 # => [:@text]
```

I asked my_object about its class, and it replied in no uncertain terms: "I'm a Greeting." Then I asked the class for a list of its instance methods. (The false argument means, "List only instance methods you defined yourself, not those ones you inherited.") The class answered with an array containing a single method name: welcome(). I also peeked into the object itself, asking for its instance variables. Again, the object's reply was loud and clear. Since objects and classes are first-class citizens in Ruby, you can get a lot of information out of running code.

However, this is only half the picture. Sure, you can read language constructs at runtime, but what about *writing* them? What if you want to add new instance methods to Greeting, alongside welcome(), while the program is running? You might be wondering why on Earth anyone would want to do that. Allow me to explain by telling a story.

The Story of Bob, Metaprogrammer

Bob, a Java coder who's just starting to learn Ruby, has a grand plan: he'll write the biggest Internet social network ever for movie buffs. To do that, he needs a database of movies and movie reviews. Bob makes it a practice to write reusable code, so he decides to build a simple library to persist objects in the database.

Bob's First Attempt

Bob's library maps each class to a database table and each object to a record. When Bob creates an object or accesses its attributes, the object generates a string of SQL and sends it to the database. All this functionality is wrapped in a base class:

Download introduction/orm.rb

```
class Entity
  attr_reader :table, :ident

  def initialize(table, ident)
    @table = table
    @ident = ident
    Database.sql "INSERT INTO #{@table} (id) VALUES (#{@ident})"
  end

  def set(col, val)
    Database.sql "UPDATE #{@table} SET #{col}='#{val}' WHERE id=#{@ident}"
  end
```

```
  def get(col)
    Database.sql("SELECT #{col} FROM #{@table} WHERE id=#{@ident}")[0][0]
  end
end
```

In Bob's database, each table has an id column. Each Entity stores the content of this column and the name of the table to which it refers. When Bob creates an Entity, the Entity saves itself to the database. Entity#set() generates SQL that updates the value of a column, and Entity#get() generates SQL that returns the value of a column. (In case you care, Bob's Database class returns record sets as arrays of arrays.)

Bob can now subclass Entity to map to a specific table. For example, class Movie maps to a database table named movies:

```
class Movie < Entity
  def initialize(ident)
    super("movies", ident)
  end

  def title
    get("title")
  end

  def title=(value)
    set("title", value)
  end

  def director
    get("director")
  end

  def director=(value)
    set("director", value)
  end
end
```

A Movie has two methods for each field: a reader such as Movie#title() and a writer such as Movie#title=(). Bob can now load a new movie into the database by firing up the Ruby command-line interpreter and typing the following:

```
movie = Movie.new(1)
movie.title = "Doctor Strangelove"
movie.director = "Stanley Kubrick"
```

This code creates a new record in movies, which has values 1, Doctor Strangelove, and Stanley Kubrick for the fields id, title, and director, respectively.[1]

Proud of himself, Bob shows the code to his older, more experienced colleague Bill. Bill stares at the screen for a few seconds and proceeds to shatter Bob's pride into tiny little pieces. "There's a lot of duplicated code here," Bill says. "You have a movies table with a title column in the database, and you have a Movie class with a @title field in the code. You also have a title() method, a title=() method, and two "title" string constants. You can solve this problem with way less code if you sprinkle some metaprogramming magic over it."

Enter Metaprogramming

At the suggestion of his expert-coder friend, Bob looks for a metaprogramming-based solution. He finds that very thing in the ActiveRecord library, a popular Ruby library that maps objects to database tables.[2] After a short tutorial, Bob is able to write the ActiveRecord version of the Movie class:

```
class Movie < ActiveRecord::Base
end
```

Yes, it's as simple as that. Bob just subclassed the ActiveRecord::Base class. He didn't have to specify a table to map Movies to. Even better, he didn't have to write boring, almost identical methods such as title() and director(). Everything just works:

```
movie = Movie.create
movie.title = "Doctor Strangelove"
movie.title                          # => "Doctor Strangelove"
```

The previous code creates a Movie object that wraps a record in the movies table, then accesses the record's title field by calling Movie#title() and Movie#title=(). But these methods are nowhere to be found in the source code. How can title() and title=() exist, if they're not defined anywhere? You can find out by looking at how ActiveRecord works.

The table name part is straightforward: ActiveRecord looks at the name of the class through introspection and then applies some simple con-

1. You probably know this already, but it doesn't hurt to refresh your memory: in Ruby, movie.title = "Doctor Strangelove" is actually a disguised call to the title=() method—the same as movie.title=("Doctor Strangelove").
2. ActiveRecord is part of Rails, the quintessential Ruby framework. You'll read more about Rails and ActiveRecord in Chapter 7, *The Design of ActiveRecord*, on page 192.

ventions. Since the class is named Movie, ActiveRecord maps it to a table named movies. (This library knows how to find plurals for English words.)

What about methods like title=() and title(), which access object attributes (*accessor methods* for short)? This is where metaprogramming comes in: Bob doesn't have to write those methods. ActiveRecord defines them automatically, after inferring their names from the database schema. ActiveRecord::Base reads the schema at runtime, discovers that the movies table has two columns named title and director, and defines accessor methods for two attributes of the same name. This means that ActiveRecord defines methods such as Movie#title() and Movie#director=() out of thin air while the program runs![3]

This is the "yang" to the introspection "yin": rather than just reading from the language constructs, you're writing into them. If you think this is an extremely powerful feature, well, you would be right.

The "M" Word Again

Now you have a more formal definition of metaprogramming:

Metaprogramming is writing code that manipulates language constructs at runtime.

How did the authors of ActiveRecord apply this concept? Instead of writing accessor methods for each class's attributes, they wrote code that defines those methods at runtime for *any* class that inherits from ActiveRecord::Base. This is what I meant when I talked about "writing code that writes code."

You might think that this is exotic, seldom-used stuff, but if you look at Ruby, as we're about to do, you'll see that it's used all around the place.

Metaprogramming and Ruby

Remember our earlier talk about ghost towns and marketplaces? If you want to "manipulate language constructs," those constructs must exist at runtime. In this respect, some languages are definitely better than others. Take a quick glance at a few languages and how much control they give you at runtime.

3. The real implementation of accessors in ActiveRecord is a bit more subtle than I describe here, as you'll see in Chapter 8, *Inside ActiveRecord*, on page 208.

A program written in C spans two different worlds: compile time, where you have language constructs such as variables and functions, and runtime, where you just have a bunch of machine code. Since most information from compile time is lost at runtime, C doesn't support metaprogramming or introspection. In C++, some language constructs do survive compilation, and that's why you can ask a C++ object for its class. In Java, the distinction between compile time and runtime is even fuzzier. You have enough introspection available to list the methods of a class or climb up a chain of superclasses.

Ruby is arguably the most metaprogramming-friendly of the current fashionable languages. It has no compile time at all, and most constructs in a Ruby program are available at runtime. You don't come up against a brick wall dividing the code that you're writing from the code that your computer executes when you run the program. There is just one world.

In this one world, metaprogramming is everywhere. In fact, metaprogramming is so deeply entrenched in the Ruby language that it's not even sharply separated from "regular" programming. You can't look at a Ruby program and say, "This part here is metaprogramming, while this other part is not." In a sense, metaprogramming is a routine part of every Ruby programmer's job.

To be clear, metaprogramming isn't an obscure art reserved for Ruby gurus, and it's also not a bolt-on power feature that's useful only for building something as sophisticated as ActiveRecord. If you want to take the path to advanced Ruby coding, you'll find metaprogramming at every step. Even if you're happy with the amount of Ruby you already know and use, you're still likely to stumble on metaprogramming in your coding travels: in the source of popular frameworks, in your favorite library, and even in small examples from random blogs. Until you master metaprogramming, you won't be able to tap into the full power of the Ruby language.

There is also another, less obvious reason why you might want to learn metaprogramming. As simple as Ruby looks at first, you can quickly become overwhelmed by its subtleties. Sooner or later, you'll be asking yourself questions such as "Can an object call a **private** method on another object of the same class?" or "How can you define class methods by importing a module?" Ultimately, all of Ruby's seemingly complicated behaviors derive from a few simple rules. Through metaprogram-

ming, you can get an intimate look at the language, learn those rules, and get answers to your nagging questions.

Now that you know what metaprogramming is about, you're ready to dive in this book.

About This Book

Part I, *Metaprogramming Ruby*, is the core of the book. It tells the story of your week in the office, paired with Bill, an experienced Ruby coder:

- Ruby's object model is the land in which metaprogramming lives. Chapter 1, *Monday: The Object Model*, on page 29 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 2, *Tuesday: Methods*, on page 62.
- Methods are just one member of a larger family also including entities such as blocks and lambdas. Chapter 3, *Wednesday: Blocks*, on page 93, 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's gaining popularity in today's development community. And, of course, this chapter 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 4, *Thursday: Class Definitions*, on page 124, talks about this scope and introduces you to some of the most powerful weapons in a metaprogrammer's arsenal. It also introduces *eigenclasses* (also known as *singleton classes*), the last concept you need to make sense of Ruby's most perplexing features.
- Finally, Chapter 5, *Friday: Code That Writes Code*, on page 162 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 object model events.

Part II of the book, *Metaprogramming in Rails*, is a case study in metaprogramming. It contains three 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.

Before you get down to reading this book, you should know about the three appendixes. Appendix A, on page 242, describes some common techniques that you'll probably find useful even if they're not, strictly speaking, metaprogramming. Appendix B, on page 254, is a look at domain-specific languages. Appendix C, on page 258, is a quick reference to 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 by using the convention *Blank Slate (86)* or *String of Code (165)*, for example. The number in parentheses is the page where the spell receives a name. If you need a quick reference to a spell, in Appendix C, on page 258, you'll find a complete spell book.

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 just because they're fun.

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

Notation Conventions

Throughout this book, I use a typewriter-like font for 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 when trying to differentiate class methods and instance methods. Second, I use a hash prefix to identify eigenclasses (#MyEigenclass).

Some of the code in this book comes straight from existing open source libraries. To avoid clutter (or to 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 the developers rarely write unit 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! In fact, you'll find specific advice on testing metaprogramming code in Chapter 9, *Metaprogramming Safely*, on page 226.

Ruby Versions

One of the joys of Ruby is that it's 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.

As I write this text, the latest stable release of Ruby is 1.9.1 and is labeled a "developer" version. Developer versions are meant as test beds for new language features, but Ruby 1.9 is generally considered stable enough for real production work—so I used it to write this book. You can stick with Ruby 1.8 if you prefer. Throughout the text, I'll tell you which features behave differently on the two versions of Ruby.

The next production version of Ruby is going to be Ruby 2.0, which will likely introduce some big changes. At the time of writing this book, this version is still too far away to either worry or rejoice about. Once 2.0 comes out, I'll update the text.

When I talk about Ruby versions, I'm talking about the "official" interpreter (sometimes called MRI for *Matz's Ruby Interpreter*[4]). To add to all the excitement (and the confusion) around Ruby, some people are also developing alternate versions of the language, like JRuby, which runs on the Java Virtual Machine,[5] or IronRuby, which runs on the Microsoft .NET platform.[6] As I sit here writing, most of these alternate Ruby implementations are progressing nicely, but be aware that some of the examples in this book might not work on some of these alternate implementations.

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 understood the code in the previous sections without much trouble, 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

4. http://www.ruby-lang.org
5. http://jruby.codehaus.org
6. http://www.ironruby.net

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![7]

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

7. I suggest the seminal *Pickaxe* [TFH08] book. You can also find an excellent interactive introduction in the *Try Ruby!* tutorial on http://tryruby.sophrinix.com.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Metaprogramming Ruby's Home Page
http://pragprog.com/titles/ppmetr
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/ppmetr.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)