Extracted from:

Metaprogramming Ruby 2

Program Like the Ruby Pros

The Pragmatic Bookshelf

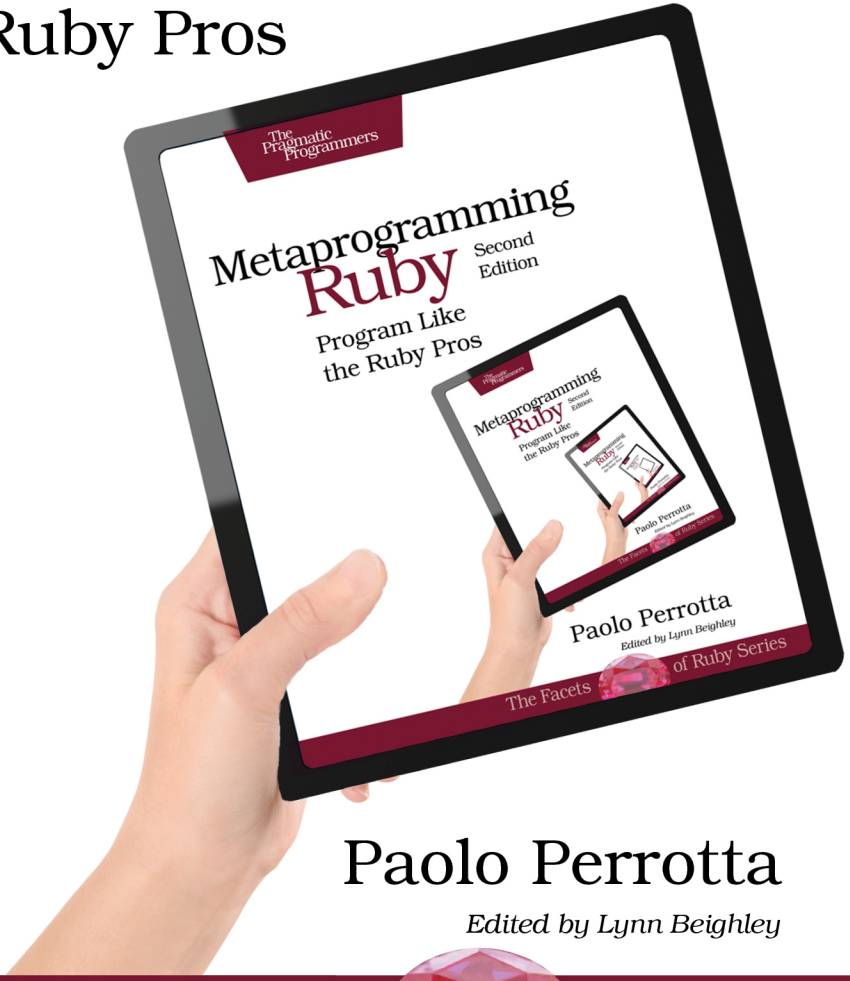
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Metaprogramming Ruby

Second Edition

Program Like the Ruby Pros



Paolo Perrotta

Edited by Lynn Beighley

The Facets of Ruby Series

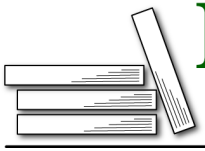
Metaprogramming Ruby 2

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.

CHAPTER 3

Tuesday: Methods

Yesterday you learned about the Ruby object model and how to make Ruby classes sing and dance for you. Today you're holding all calls to focus on *methods*.

The objects in your code talk to each other all the time. Some languages—such as Java and C—feature a compiler that presides over this chatting. For every method call, the compiler checks to see that the receiving object has a matching method. This is called *static type checking*, and the languages that adopt it are called *static languages*. For example, if you call `talk_simple` on a `Lawyer` object that has no such method, the compiler protests loudly.

Dynamic languages—such as Python and Ruby—don't have a compiler policing method calls. As a consequence, you can start a program that calls `talk_simple` on a `Lawyer`, and everything works just fine—that is, until that specific line of code is executed. Only then does the `Lawyer` complain that it doesn't understand that call.

That's an important advantage of static type checking: the compiler can spot some of your mistakes before the code runs. This protectiveness, however, comes at a price. Static languages often require you to write lots of tedious, repetitive methods—the so-called *boilerplate methods*—just to make the compiler happy. (For example, `get` and `set` methods to access an object's properties, or scores of methods that do nothing but delegate to some other object.)

In Ruby, boilerplate methods aren't a problem, because you can easily avoid them with techniques that would be impractical or just plain impossible in a static language. In this chapter, we'll focus on those techniques.

A Duplication Problem

Where you and Bill face a problem with duplicated code.

Today, your boss asked you to work on a program for the accounting department. They want a system that flags expenses greater than $99 for computer gear, so they can crack down on developers splurging with company money. (You read that right: *$99*. The purchasing department isn't fooling around.)

Some other developers already took a stab at the project, coding a report that lists all the components of each computer in the company and how much each component costs. To date, they haven't plugged in any real data. Here's where you and Bill come in.

The Legacy System

Right from the start, you have a challenge on your hands: the data you need to load into the already established program is stored in a legacy system stuck behind an awkwardly coded class named `DS` (for "data source"):

methods/computer/data_source.rb

```
class DS
  def initialize # connect to data source...
  def get_cpu_info(workstation_id) # ...
  def get_cpu_price(workstation_id) # ...
  def get_mouse_info(workstation_id) # ...
  def get_mouse_price(workstation_id) # ...
  def get_keyboard_info(workstation_id) # ...
  def get_keyboard_price(workstation_id) # ...
  def get_display_info(workstation_id) # ...
  def get_display_price(workstation_id) # ...
  # ...and so on
```

`DS#initialize` connects to the data system when you create a new `DS` object. The other methods—and there are dozens of them—take a workstation identifier and return descriptions and prices for the computer's components. With Bill standing by to offer moral support, you quickly try the class in irb:

```
ds = DS.new
ds.get_cpu_info(42)     # => "2.9 Ghz quad-core"
ds.get_cpu_price(42)    # => 120
ds.get_mouse_info(42)   # => "Wireless Touch"
ds.get_mouse_price(42)  # => 60
```

It looks like workstation number 42 has a 2.9GHz CPU and a luxurious $60 mouse. This is enough data to get you started.

Double, Treble… Trouble

You have to wrap DS into an object that fits the reporting application. This means each Computer must be an object. This object has a single method for each component, returning a string that describes both the component and its price. Remember that price limit set by the purchasing department? Keeping this requirement in mind, you know that if the component costs $100 or more, the string must begin with an asterisk to draw people's attention.

You kick off development by writing the first three methods in the Computer class:

methods/computer/duplicated.rb

```
class Computer
  def initialize(computer_id, data_source)
    @id = computer_id
    @data_source = data_source
  end

  def mouse
    info = @data_source.get_mouse_info(@id)
    price = @data_source.get_mouse_price(@id)
    result = "Mouse: #{info} ($#{price})"
    return "* #{result}" if price >= 100
    result
  end

  def cpu
    info = @data_source.get_cpu_info(@id)
    price = @data_source.get_cpu_price(@id)
    result = "Cpu: #{info} ($#{price})"
    return "* #{result}" if price >= 100
    result
  end

  def keyboard
    info = @data_source.get_keyboard_info(@id)
    price = @data_source.get_keyboard_price(@id)
    result = "Keyboard: #{info} ($#{price})"
    return "* #{result}" if price >= 100
    result
  end

  # ...
end
```

At this point in the development of Computer, you find yourself bogged down in a swampland of repetitive copy and paste. You have a long list of methods

left to deal with, and you should also write tests for each and every method, because it's easy to make mistakes in duplicated code.

"I can think of two different ways to remove this duplication," Bill says. "One is a spell called Dynamic Methods. The other is a special method called method_missing. We can try both solutions and decide which one we like better." You agree to start with Dynamic Methods and get to method_missing after that.

Dynamic Methods

Where you learn how to call and define methods dynamically, and you remove the duplicated code.

"When I was a young developer learning C++," Bill says, "my mentors told me that when you call a method, you're actually sending a message to an object. It took me a while to get used to that concept. If I'd been using Ruby back then, that notion of sending messages would have come more naturally to me."

Calling Methods Dynamically

When you call a method, you usually do so using the standard dot notation:

methods/dynamic_call.rb

```
class MyClass
  def my_method(my_arg)
    my_arg * 2
  end
end

obj = MyClass.new
obj.my_method(3)  # => 6
```

You also have an alternative: call MyClass#my_method using Object#send in place of the dot notation:

```
obj.send(:my_method, 3)   # => 6
```

The previous code still calls my_method, but it does so through send. The first argument to send is the message that you're sending to the object—that is, a symbol or a string representing the name of a method. (See *Method Names and Symbols*, on page 11.) Any remaining arguments (and the block, if one exists) are simply passed on to the method.

Why would you use send instead of the plain old dot notation? Because with send, the name of the method that you want to call becomes just a regular argument. You can wait literally until the very last moment to decide which

method to call, *while* the code is running. This technique is called *Dynamic Dispatch*, and you'll find it wildly useful. To help reveal its magic, let's look at a couple of real-life examples.

Spell: Dynamic Dispatch

Method Names and Symbols

People who are new to the language are sometimes confused by Ruby's symbols. Symbols and strings belong to two separate and unrelated classes:

```
:x.class  # => Symbol
"x".class # => String
```

Nevertheless, symbols are similar enough to strings that you might wonder what's the point of having symbols at all. Can't you just use regular strings everywhere?

There are a few different reasons to use symbols in place of regular strings, but in the end the choice boils down to conventions. In most cases, symbols are used as names of things—in particular, names of metaprogramming-related things such as methods. Symbols are a good fit for such names because they are *immutable*: you can change the characters inside a string, but you can't do that for symbols. You wouldn't expect the name of a method to change, so it makes sense to use a symbol when you refer to a method name.

For example, when you call `Object#send`, you need to pass it the name of a method as a first argument. Although `send` accepts this name as either a symbol or a string, symbols are usually considered more kosher:

```
# rather than: 1.send("+", 2)
1.send(:+, 2)   # => 3
```

Regardless, you can easily convert from string to symbol and back:

```
"abc".to_sym    #=> :abc
:abc.to_s       #=> "abc"
```

The Pry Example

One example of Dynamic Dispatch comes from Pry. Pry is a popular alternative to irb, Ruby's command-line interpreter. A `Pry` object stores the interpreter's configuration into its own attributes, such as `memory_size` and `quiet`:

methods/pry_example.rb

```
require "pry"

pry = Pry.new
pry.memory_size = 101
pry.memory_size        # => 101
pry.quiet = true
```

For each instance method like Pry#memory_size, there is a corresponding class method (Pry.memory_size) that returns the default value of the attribute:

```
Pry.memory_size       # => 100
```

Let's look a little deeper inside the Pry source code. To configure a Pry instance, you can call a method named Pry#refresh. This method takes a hash that maps attribute names to their new values:

```
pry.refresh(:memory_size => 99, :quiet => false)
pry.memory_size       # => 99
pry.quiet             # => false
```

Pry#refresh has a lot of work to do: it needs to go through each attribute (such as self.memory_size); initialize the attribute with its default value (such as Pry.memory_size); and finally check whether the hash argument contains a new value for the same attribute, and if it does, set the new value. Pry#refresh could do all of those steps with code like this:

```
def refresh(options={})
  defaults[:memory_size] = Pry.memory_size
  self.memory_size = options[:memory_size] if options[:memory_size]

  defaults[:quiet] = Pry.quiet
  self.quiet = options[:quiet] if options[:quiet]
  # same for all the other attributes...
end
```

Those two lines of code would have to be repeated for each and every attribute. That's a lot of duplicated code. Pry#refresh manages to avoid that duplication, and instead uses *Dynamic Dispatch (11)* to set all the attributes with just a few lines of code:

gems/pry-0.9.12.2/lib/pry/pry_instance.rb

```
def refresh(options={})
  defaults   = {}
  attributes = [ :input, :output, :commands, :print, :quiet,
                 :exception_handler, :hooks, :custom_completions,
                 :prompt, :memory_size, :extra_sticky_locals ]

  attributes.each do |attribute|
    defaults[attribute] = Pry.send attribute
  end
  # ...
  defaults.merge!(options).each do |key, value|
    send("#{key}=", value) if respond_to?("#{key}=")
  end

  true
end
```

The code above uses send to read the default attribute values into a hash, merges this hash with the options hash, and finally uses send again to call attribute accessors such as memory_size=. The Kernel#respond_to? method returns true if methods such as Pry#memory_size= actually exist, so that any key in options that doesn't match an existing attribute will be ignored. Neat, huh?

Privacy Matters

Remember what Spiderman's uncle used to say? "With great power comes great responsibility." The Object#send method is very powerful—perhaps *too* powerful. In particular, you can call any method with send, including private methods.

If that kind of breaching of encapsulation makes you uneasy, you can use public_send instead. It's like send, but it makes a point of respecting the receiver's privacy. Be prepared, however, for the fact that Ruby code in the wild rarely bothers with this concern. If anything, a lot of Ruby programmers use send exactly *because* it allows calling private methods, not in spite of that.

Now you know about send and Dynamic Dispatch—but there is more to Dynamic Methods than that. You're not limited to calling methods dynamically. You can also *define* methods dynamically. It's time to see how.

Defining Methods Dynamically

You can define a method on the spot with Module#define_method. You just need to provide a method name and a block, which becomes the method body:

methods/dynamic_definition.rb

```
class MyClass
  define_method :my_method do |my_arg|
    my_arg * 3
  end
end

obj = MyClass.new
obj.my_method(2)  # => 6

require_relative '../test/assertions'
assert_equals 6, obj.my_method(2)
```

define_method is executed within MyClass, so my_method is defined as an instance method of MyClass. This technique of defining a method at runtime is called a *Dynamic Method*.

Spell: Dynamic Method

There is one important reason to use define_method over the more familiar def keyword: define_method allows you to decide the name of the defined method

at runtime. To see an example of this technique, look back at your original refactoring problem.