Extracted from:

Metaprogramming Ruby

The Pragmatic Programmers

Metaprogramming Ruby

Paolo Perrotta

Edited by Jill Steinberg

The Facets of Ruby Series

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

Chapter 2

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*.

As you know, the objects in your code talk to each other all the time. Some languages—Java, for one—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*. So, 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.

Admittedly, that's one advantage of static type checking: the compiler can spot some of your mistakes before the code runs. Before you ask the obvious question, realize that this protectiveness comes at a high price. Static languages force you to write lots of tedious, repetitive methods—these are the so-called boilerplate methods—just to make the compiler happy. (If you're a Java programmer, just think of all the "get" and "set" methods you've written in your life or the innumerable 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 home in on those techniques.

2.1 A Duplication Problem

Where you and Bill have a problem with duplicated code.

Your boss is happy with the job that you and Bill did yesterday. Today, she gives the two of you a more serious integration assignment.

To give you a bit of history, some folks in the purchasing department are concerned that developers are spending oodles of company money on computing gear. To make sure things don't get out of hand, they're requesting a system that automatically flags expenses more than $99. (You read that right: *ninety-nine*. The purchasing department isn't fooling around.)

Before you and Bill, some developers 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, the two of 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"):

Download methods/computer/data_source.rb

```ruby
class DS
  def initialize # connect to data source...
  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_cpu_info(workstation_id) # ...
  def get_cpu_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. The output is in the form of strings, with prices expressed as integer numbers rounded to the nearest dollar. With Bill standing by to offer moral support, you quickly try the class in irb:

```
ds = DS.new
ds.get_cpu_info(42)      # => 2.16 Ghz
ds.get_cpu_price(42)     # => 150
ds.get_mouse_info(42)    # => Dual Optical
ds.get_mouse_price(42)   # => 40
```

It looks like workstation number 42 has a 2.16GHz CPU and a luxurious $40 dual optical mouse. This is enough data to get you started.

Double, Treble. . . Trouble

You and Bill 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 describing both the component and its price. Remember that price limit the purchasing department set? 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:

Download methods/computer/boring.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. This is getting boring fast—not to mention painful.

Bill is right there with you, verbalizing precisely what's going through your head: "This is just the same method again and again, with some minor changes." You turn to each other and ask simultaneously, as if on cue, "How can we refactor it?"

Bill's Plan

"I can think of not one but *two* different ways to remove this duplication," Bill brags. He suggests using either Dynamic Methods or a special method called method_missing(). By trying both solutions, you and Bill can decide which one you like better. You agree to start with Dynamic Methods and get to method_missing() after that.

2.2 Dynamic Methods

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

"As I mentioned, we can remove the duplication in our code with either Dynamic Methods or method_missing()," Bill recalls. "Forget about method_missing() for now—we'll get to that this afternoon. To introduce Dynamic Methods, allow me to tell you a story from my youth," he says.

"When I was a young developer learning C++," Bill muses, "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. Of course, if I'd been using Ruby back then, that notion of sending messages would have come more naturally to me." Bill launches into a mini-presentation.

Calling Methods Dynamically

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

Download methods/dynamic_call.rb

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

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

Bill demonstrates how you can also 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, the name of a method. You can use a string or a symbol, but symbols are considered more kosher (see the sidebar on the next page). Any remaining arguments (and the block, if one exists) are simply passed on to the method.

"Wait a minute," you interject. "Why on Earth would I use send() instead of the plain old dot notation?" Bill is glad you asked, pointing out that this is one of the cool things about Ruby. 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, Bill shows you a couple of real-life examples.

Spell: Dynamic Dispatch

The Camping Example

One example of Dynamic Dispatch comes from Camping, a minimalist Ruby web framework. A Camping application stores its configuration parameters as key-value pairs in a file created with YAML, a simple and very popular serialization format.[1]

1. Camping, a framework written by "_why the lucky stiff," can be installed with gem install camping. YAML stands for "Yaml Ain't Markup Language," and you can learn more about it at http://www.yaml.org.

Symbols

If you prefix any sequence of characters with a colon (actually, any sequence that would make a legal variable name), it becomes a *symbol*:

```
x = :this_is_a_symbol
```

Symbols and strings are not related, and they belong to entirely different classes. Nevertheless, symbols are similar enough to strings that most Ruby beginners are confused by them. "What's the point of having symbols at all? Why can't I just use regular strings everywhere?" they ask.

Different people will provide different answers to these questions. Some might point out that symbols are different from strings because symbols are *immutable*: you can change the characters inside a string, but you can't do that for symbols. Also, some operations (such as comparisons) are faster on symbols than they are on strings. But, choosing between symbols and strings basically comes down to conventions. In most cases, symbols are used as names of things—in particular, names of metaprogramming-related things such as methods.

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 appropriate:

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

Regardless, you can easily convert a string to a symbol (by calling either String#to_sym() or String#intern()) or back (by calling either Symbol#to_s() or Symbol#id2name()).

The configuration file for a blog application might look like this:

```
admin : Bill
title : Rubyland
topic : Ruby and more
```

Camping copies keys and values from the file into its own configuration object. (This object is an OpenStruct. You can read more about this class in Section 2.3, *The OpenStruct Example*, on page 77.) Assume that you store your application's configuration in a conf object. In an ideal world, the configuration code for the blog application would look like this:

```
conf.admin = 'Bill'
conf.title = 'Rubyland'
conf.topic = 'Ruby and more'
```

The sad fact is, in real life, Camping's source can't contain this kind of code. That's because it can't know in advance which keys you need in your specific application—so it can't know which methods it's supposed to call. It can discover the keys you need only at runtime, by parsing the YAML file. For this reason, Camping resorts to Dynamic Dispatch. For each key-value pair, it composes the name of an assignment method, such as admin=(), and sends the method to conf:

Download gems/camping-1.5/bin/camping

```
# Load configuration if any
if conf.rc and File.exists?( conf.rc )
  YAML.load_file(conf.rc).each do |k,v|
    conf.send("#{k}=", v)
  end
end
```

Neat, huh?

The Test::Unit Example

Another example of *Dynamic Dispatch* (*66*) comes from the Test::Unit standard library. Test::Unit uses a naming convention to decide which methods are tests. A TestCase looks inside its own public methods and selects the methods that have names starting with *test*:

```
method_names = public_instance_methods(true)
tests = method_names.delete_if {|method_name| method_name !~ /^test./}
```

Now TestCase has an array of all test methods. Later, it uses send() to call each method in the array.[2] This particular flavor of Dynamic Dispatch

2. To nitpick, TestCase uses a synonym of send() named __send__(). You'll find out why in the sidebar on page 89.

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. Short of using a *Context Probe* (107), this is the easiest way to peek into an object's private matters.

Some Rubyists think that send() makes it too easy to unwillingly break encapsulation. Ruby 1.9 experimented with changing send()'s behavior, but the changes were ultimately reverted. As of Ruby 1.9.1, send() can still call private methods—and many libraries use it just for that purpose. On the other hand, you have a new public_send() method that respects the receiver's privacy.

is sometimes called *Pattern Dispatch*, because it filters methods based on a pattern in their names.

Spell: Pattern Dispatch

Bill leans back in his chair. "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. I'll show you 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:

Download methods/dynamic_definition.rb

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

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

define_method() is executed within MyClass, so my_method() is defined as an instance method of MyClass.[3] This technique of defining a method at runtime is called a *Dynamic Method*.

Spell: Dynamic Method

You learned how to use Module#define_method() in place of the **def** keyword to define a method and how to use send() in place of the dot notation to call a method. Now you can go back to your and Bill's original problem and put this knowledge to work.

Refactoring the Computer Class

Recall the code that pulled you and Bill into this dynamic discussion:

Download methods/computer/boring.rb

```ruby
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
```

3. There is also an Object#define_method() that defines a *Singleton Method* (135).

Now that you know about send() and define_method(), you and Bill can get to work and remove the duplication in Computer. Time to refactor!

Step 1: Adding Dynamic Dispatches

You and Bill start, extracting the duplicated code into its own message-sending method:

Download methods/computer/send.rb

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

▶   def mouse
▶     component :mouse
▶   end
▶
▶   def cpu
▶     component :cpu
▶   end
▶
▶   def keyboard
▶     component :keyboard
▶   end
▶
▶   def component(name)
▶     info = @data_source.send "get_#{name}_info", @id
▶     price = @data_source.send "get_#{name}_price", @id
▶     result = "#{name.to_s.capitalize}: #{info} ($#{price})"
▶     return "* #{result}" if price >= 100
▶     result
▶   end
end
```

A call to mouse() is delegated to component(), which in turn calls DS#get_mouse_info() and DS#get_mouse_price(). The call also writes the capitalized name of the component in the resulting string. (Since component() expects the name as a symbol, it converts the symbol to a string with Symbol#to_s().) You open an irb session and smoke-test the new Computer:

```
my_computer = Computer.new(42, DS.new)
my_computer.cpu    # => * Cpu: 2.16 Ghz ($220)
```

This new version of Computer is a step forward, because it contains far fewer duplicated lines, but you still have to write dozens of similar methods. To avoid writing all those methods, use define_method().

Step 2: Generating Methods Dynamically

You and Bill refactor Computer to use define_method():

Download methods/computer/dynamic.rb

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

  def self.define_component(name)
    define_method(name) {
      info = @data_source.send "get_#{name}_info", @id
      price = @data_source.send "get_#{name}_price", @id
      result = "#{name.to_s.capitalize}: #{info} ($#{price})"
      return "* #{result}" if price >= 100
      result
    }
  end

  define_component :mouse
  define_component :cpu
  define_component :keyboard
end
```

Note that define_method() is executed inside the definition of Computer, where Computer is the implicit **self**.[4] This means you're calling define_component() on Computer, so it must be a class method.

You quickly test the slimmed-down Computer class in irb and discover that it still works. This is great news!

Step 3: Sprinkling the Code with Introspection

The latest Computer contains minimal duplication, but you can push it even further and remove the duplication altogether. How? By getting rid of all the define_component() methods. You can do that by introspecting the data_source argument and extracting the names of all components:

Download methods/computer/more_dynamic.rb

```
class Computer
  def initialize(computer_id, data_source)
    @id = computer_id
    @data_source = data_source
    data_source.methods.grep(/^get_(.*)_info$/) { Computer.define_component $1 }
  end
```

4. See Section 1.5, *Discovering self*, on page 55.

```
  def self.define_component(name)
    define_method(name) {
      info = @data_source.send "get_#{name}_info", @id
      price = @data_source.send "get_#{name}_price", @id
      result = "#{name.capitalize}: #{info} ($#{price})"
      return "* #{result}" if price >= 100
      result
    }
  end
end
```

The new line in initialize() is where the magic happens. To understand it, you need to know a couple of things. First, if you pass a block to String#grep(), the block is evaluated for each element that matches the regular expression. Second, the string matching the parenthesized part of the regular expression is stored in the global variable $1. So, if data_source has methods named get_cpu_info() and get_mouse_info(), this code ultimately calls Computer.define_component() twice, with the strings "cpu" and "mouse". Note that you're calling define_component() with a string rather than a symbol, so you don't need to convert the argument to string.

The duplicated code is finally gone for good. As a bonus, you don't even have to write or maintain the list of components. If someone adds a new component to DS, the Computer class will support it automatically. Wonderful!

Let's Try That Again!

Your refactoring was a resounding success, but Bill is not willing to stop here. "We said that we were going to try *two* different solutions to this problem, remember? We've only found one, involving *Dynamic Dispatch* (*66*) and *Dynamic Methods* (*70*)." Although it has served the two of you well, to be fair, you need to give the other solution a chance.

"For this second solution," Bill continues, "we need to talk about some strange methods that are not really methods and a very special method named method_missing()."

2.3 method_missing()

Where you listen to spooky stories about Ghost Methods and dynamic proxies and you try a second way to remove duplicated code.