CHAPTER 25

Create Self-Modifying Classes

In the last chapter we looked at how you can take advantage of Ruby's open classes to do monkey patching, to add to, modify, or even subtract from an existing class. We saw how you can get a lot of programming mileage out of this technique in the form of enhanced or repaired classes.

In this chapter we're going to continue our look at the dynamic nature of Ruby classes, and we'll start by asking a very fundamental question: How do Ruby classes get defined in the first place? Not only will the answer to this question illuminate how monkey patching works, but it will also enable us to build classes that can programmatically modify themselves. We will also learn how to cope with some of the common mistakes you can make when doing this kind of programming, and then we'll check out some examples from the Ruby code base.

Open Classes, Again

So far we've seen how you can change a Ruby class definition by simply repeating the class definition. If the `Widget` class is already defined and you come back a second time to say `class Widget`, then your widgets are going to change. Something we rather took for granted in that discussion was the question of how classes get defined in the first place. To see what I mean, take a look at this rather strange-looking class definition:

```
class MostlyEmpty
  puts "hello from inside the class"
end
```

Instead of methods and attributes and constants, the `MostlyEmpty` class lives up to its name by consisting of a single `puts` statement. Run the three lines above and you will see this output:

```
hello from inside the class
```

Along with the output—and the new, mostly empty class—we can pull something very important from this little demonstration—that Ruby class definitions are executable. That `puts` statement went off because when the Ruby interpreter hits a class declaration, it executes the code between the `class` and the `end`.[1]

We can use this little discovery to prove something you've been told but perhaps have never really seen before your eyes: the value of `self` inside a class definition is the class that you are defining. Run this:

```
class MostlyEmpty
  puts "The value of self is #{self}"
end
```

And there is your confirmation:

```
The value of self is #<Class:0x873698c>::MostlyEmpty
```

We can also use this trick to see *when* methods get defined. All we need is a class with an actual method along and some additional instrumentation:

```
class LessEmpty
  pp instance_methods(false)

  def do_something
    puts "I'm doing something!"
  end
```

1. Well, technically the interpreter reads in the whole class first and then executes the class body.

```
  pp instance_methods(false)
end
```

Again, we have a class with some ordinary code to execute, this time two `pp` statements. Using the `instance_methods` method, those `pp` statements print the names of all the instance methods defined on the `LessEmpty` class.[2] Sandwiched between the `pp`s is a method definition. The idea is to find out when the `do_something` method gets defined. When you run this code you will see the following:

```
[]
[:do_something]
```

This output is trying to tell us that the `do_something` method is not defined at the top of the class—before the `def do_something`—but it is defined at the bottom, after the method definition. We have just discovered that Ruby classes are defined piecemeal, one step—or method—at a time. When Ruby sees that initial `class LessEmpty`, it creates a new and completely empty class. It then executes the class body, the code between the `class` statement and the final `end`. Whatever is inside the class definition—be it an `if` or a `puts` or a method defining `def`—simply gets executed in turn.

Knowing all this gives us some interesting insight into how Ruby classes really operate. For example, given what we've learned so far, it's now no surprise that if you define the same method twice, like this:

```
class TheSameMethodTwice

  def do_something
    puts "first version"
  end

  # In between method definitions

  def do_something
    puts "second version"
  end
end
```

2. Passing false to `instance_methods` says that we don't want to see any inherited methods, only the methods defined directly by `LessEmpty`.

```
twice = TheSameMethodTwice.new
twice.do_something
```

The first version of the method actually springs into existence, briefly, in the space between the two definitions, only to be snubbed out by the second definition. The second one wins, as you can see from the output:

```
second version
```

Now, given that you can reopen Ruby classes, the code above is only a shade different from:

```
class TheSameMethodTwice
  def do_something
    puts "first version"
  end
end

class TheSameMethodTwice
  def do_something
    puts "second version"
  end
end
```

This latest version is also a classic example of monkey patching. We started with the `TheSameMethodTwice` class with the first version of the method, and then we reopened the class and replaced it with the second rendition of the method.

Put Programming Logic in Your Classes

Beyond helping us understand how monkey patching works, all of this theory has some real, and useful, consequences. Being able to embed code in your classes means that your classes can make run-time decisions about what methods to define and the code that those methods will contain. For example, we might imagine that all the applications using the `Document` class want to be able to save their documents to a file, but only some of those applications need to encrypt the documents as they are saved. If there was a constant that turned the encryption off and on, then it's fairly easy to build a class that configures itself accordingly:

```
class Document

  # Lots of code omitted...

  def save( path )
    File.open( path, 'w' ) do |f|
      f.puts( encrypt( @title ))
      f.puts( encrypt( @author ) )
      f.puts( encrypt( @content ))
    end
  end

  if ENCRYPTION_ENABLED
    def encrypt( string )
      string.tr( 'a-zA-Z', 'm-za-lM-ZA-L')
    end
  else
    def encrypt( string )
      string
    end
  end
end
```

This code starts out with a mundane `save` method, which writes the data in the document out to a file, after running it through the `encrypt` method. The question is, which `encrypt` method? It's the class-level logic that makes the decision. If the `ENCRYPTION_ENABLED` constant is true, we end up with an `encrypt` method that does indeed shuffle the contents of the string. On the other hand, if `ENCRYPTION_ENABLED` isn't true, we get an `encrypt` method that does nothing. Critically, the `ENCRYPTION_ENABLED` logic runs exactly once, when the class is loaded.

Class Methods That Change Their Class

The code that executes inside a class definition has something in common with a class method: They both execute with `self` set to the class. This suggests that we can use class methods to make the same kind of structural changes that we have done so far with class-level logic, and so we can. Here is our encryption example again, this time wrapped in a class method:

```
ENCRYPTION_ENABLED = true

class Document

   # Most of the class left behind...

  def self.enable_encryption( enabled )
    if enabled
      def encrypt_string( string )
        string.tr( 'a-zA-Z', 'm-za-lM-ZA-L')
      end
    else
      def encrypt_string( string )
        string
      end
    end
  end

  enable_encryption( ENCRYPTION_ENABLED )
end
```

This code does the very recursive trick of defining a class method, `enable_encryption`, which itself defines an instance method. Actually, it defines one of two versions of the `encrypt_string` method, depending on whether the `enabled` parameter is passed in as true or false. The last line of the class definition calls `enable_encryption`, passing in true, thereby starting us off with encrypting turned on. A handy side effect of this latest implementation is that we can toggle encryption off and on by calling the class method from outside. Thus, if we wanted to be sure that we were going to write encryption-free documents we could run:

```
Document.enable_encryption( false )
```

And we would have the "do nothing" version of the `encrypt_string` method.

In the Wild

Executable class definitions are wonderfully useful when you need to write code that will work in different environments. Take the transition that the Ruby world is going

through right now, from Ruby version 1.8.X to version 1.9. Progress is a great thing, but it can also be a pain in the neck. It's great that version 1.9 adds all sorts of new methods to the built-in classes, but what if you need to write code that will run in both Ruby versions?

For example, one of the differences between 1.8 and 1.9 lies in the `String` class. Take this string:

```
name = 'Robert Jordon'
```

In Ruby 1.9, `name[2]` will evaluate to a string containing one character, in this case `'b'`. In Ruby 1.8, `name[2]` will give you 98, the number that lurks behind the letter b. Now think about the problems this will cause if we wanted to have a `char_at` method on our `Document` class, a method that always returns a one-character string. It's trivial in Ruby 1.9:

```
class Document
  # Ruby 1.9 version

  def char_at( index )
    @content[ index ]
  end
end
```

But in Ruby 1.8 we need to convert the integer back into a string with a call to the `chr` method:

```
class Document
  # Ruby 1.8 version

  def char_at( index )
    @content[ index ].chr
  end
end
```

The problem is, which version of `char_at` should we use at any given moment? One easy way to deal with this kind of problem is to simply put an `if` statement right at the class level, an `if` statement that will pick the right method:

```
class Document
  # Lots of stuff omitted...

  if RUBY_VERSION >= '1.9'
    def char_at( index )
      @content[ index ]
    end
  else
    def char_at( index )
      @content[ index ].chr
    end
  end
end
```

This code uses the built in `RUBY_VERSION` constant to figure out which Ruby version it's running on and works from there. The key thing to understand about the `if` statement in the code here is that it executes exactly once as the class is being defined.

You can find a spectacular example of on-the-fly class modification in the delightful habit that Rails has of picking up your code changes on the fly. One thing that makes using Rails such a delight is that once you have a basic project set up, you simply crank up the Rails server and start hacking. With a few exceptions, Rails will magically pick up your changes and incorporate them into the running application, without so much as a hiccup let alone a server restart. Clearly there is something interesting going on inside of Rails; the question is what?

To see whether we can glean the answer, let's modify `Document` to make it reloadable. What we need is a `Document` class method that will sync the code stored in the file with the code that is actually running. A naive approach is to have the `reload` method simply reread the `Document` source code in a recursive act of monkey patching:

```
class Document
  def self.reload
    load( __FILE__ )
  end

  # Rest of the class omitted...
end
```

There is a surprising amount happening in the one-line `reload` class method shown here. First, it uses the Ruby-supplied `load` method to reread its own source.

The `load` method is similar to the more familiar `require`. The difference between `load` and `require` is that `require` keeps track of which files are already loaded so that it doesn't load the same file twice. The `load` method just loads the file, no questions asked. Since loading the same file twice is exactly what we're after here, we'll use the dumber `load`. The `reload` method also uses `__FILE__`. `__FILE__` is also supplied via the magic of Ruby and is always set to the path of the source file of the current class, which is just what we need here.

The `reload` method above takes us a long way toward our goal of a Rails-like reloadable class. Calling `reload` in the example above will ensure that any new methods added to the source file will show up in the running system. It will also catch any changes to existing methods. This first version of `reload` falls short because it can't get rid of deleted methods. If you edit `document.rb` and remove a method and then reload the source file, the original method will stubbornly stay in place in the running Ruby interpreter. The solution is to remove all the methods from the class *before* reloading it:

```
class Document
  def self.reload
    remove_instance_methods
    load( __FILE__ )
  end

  def self.remove_instance_methods
    instance_methods(false).each do |method|
      remove_method(method)
    end
  end

  # Rest of the class omitted...
end
```

The additional `remove_instance_methods` method in this last example prepares the `Document` class for reloading by systematically removing all of the instance methods from it, leaving behind an empty shell that is repopulated when the source file is reloaded.[3]

3. Not quite as empty a shell as we might like. It doesn't clear out any class methods, class variables, and class instance variables. One of the best things about writing books is that I get to leave something like that as an exercise for the reader.

Staying Out of Trouble

Let's face it: Where there is code, there will be bugs. Write enough of the kind of class-level logic discussed in this chapter and eventually you'll make all of the traditional bone-headed mistakes that have plagued engineers since the first `hello_world`. The difference is that the same old mistakes can have some interesting consequences when they occur in the middle of making structural changes to your classes. For example, take a look at this broken version of our encrypting document class:

```
ENCRYPTION_ENABLED = true

# Broken!!

class Document

   # Most of the class left behind...

  def self.enable_encryption( enabled )
    if enabled
      def encrypt_string( string )
        string.tr( 'a-zA-Z', 'm-za-lM-ZA-L')
      end
    else
      def encrypt_string( string )
        string
      end
    end
  end
end
```

See the problem? We left out the call to `enable_encryption` at the bottom of the class, so this version of `Document` will start out with no `encrypt_string` method at all. This will give us an unpleasant shock when we try to save a document.

In the same vein, if you make a mistake in one branch of the class-level logic, it will not show up until you actually use the branch. Take a look at this unfortunate bit of code:

```
def self.enable_encryption( enabled )
  if enabled
```

```
      def encrypt_string( string )
        string.tr( 'a-zA-Z', 'm-za-lM-ZA-L')
      end
    else
      def incrypt_string( string )
        string
      end
    end
  end
```

With this code, trying to turn encryption off will only succeed in defining a second method with the very odd name of `incrypt_string`. The moral here is simple and is spelled R-S-P-E-C:

```
describe Document do
  before :each do
    @doc = Document.new( "test", "tester", "this is a test" )
  end

  it "should encrypt if encryption is enabled" do
    Document.enable_encryption( true )
    @doc.encrypt_string( 'abc' ).should_not == 'abc'
  end

  it "should not encrypt if encryption is disabled" do
    Document.enable_encryption( false )
    @doc.encrypt_string( 'abc' ).should == 'abc'
  end
end
```

While regular code needs unit tests, metaprogramming code absolutely cries out for them!

Wrapping Up

If this chapter did its job, you are walking away with two big ideas: The first is that Ruby classes are executable. During the process of defining a class, Ruby executes the code between the `class` and the `end`, modifying the class as it goes. The second is that

since class definitions are executable, you can insert logic in your class definitions, logic that will determine exactly what the class will look like. And actually there is a third big idea, not a new idea but one that bears repeating: Programs—especially metaprograms—that lack automated tests are probably not going to work.